A Study of Application-awareness in Software-defined Data Center Networks

Chui-Hui Chiu
Louisiana State University and Agricultural and Mechanical College, chiuchuihui@gmail.com

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_dissertations

Part of the Computer and Systems Architecture Commons, Data Storage Systems Commons, and the Digital Communications and Networking Commons

https://digitalcommons.lsu.edu/gradschool_dissertations/4138

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Doctoral Dissertations by an authorized graduate school editor of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
A STUDY OF APPLICATION-AWARENESS
IN SOFTWARE-DEFINED DATA CENTER NETWORKS

A Dissertation
Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

School of Electrical Engineering and Computer Science

by
Chui-Hui Chiu
B.S., Taipei Municipal University of Education, 2006
M.S., Louisiana State University, 2011
December 2017
ACKNOWLEDGMENTS

First, I would like to express my sincere appreciation to my PhD. advisory committee. As my advisor and the chair of the committee, Dr. Seung-Jong Park brought numerous enlightening pieces of advice on computer networking research. He also highly trusted and supported me to lead the BIC-LSU service construction. Dr. Qingyang Wang helped me perfectly apply my knowledge in computer networking to cloud computing for conducting an inter-disciplinary research. Guidances from Dr. Konstantin Busch and Dr. Gabriele Piccoli are also very precious.

Second, I am grateful to my colleagues who developed the BIC-LSU service infrastructure with me. Georgi Stoyanov and Nathan Lewis constructed the physical spine-leaf fat-tree network topology of the high-speed storage area network. Nathan Lewis configured OpenFlow settings on all OpenFlow switches and implemented a fundamental OpenFlow Controller. Dipak Kumar Singh helped integrating the web portal and the BIC-LSU Command Service. Collaborating with highly skilled experts from multiple areas in a group was an exciting and pleasant experience.

Third, I am appreciative to my labmates with whom we constructed our data center testbed together. Mohammad Mussadiq Jalazai with advanced knowledge in production network device operation and administration configured all switches in the control network. Dipak Kumar Singh helped mount server machines and wire cables.

Last, my thanks go to anyone who has helped me during my study but not explicitly mentioned above. Every little bit of help is a necessity for completing my works.
Table of Contents

ACKNOWLEDGEMENTS ................................................... ii

LIST OF TABLES ............................................................ v

LIST OF FIGURES ............................................................ ix

ABSTRACT ................................................................. x

CHAPTER 1: INTRODUCTION AND MOTIVATION .............. 1
  1.1 Development of Data Center Network and Motivation .... 1
  1.2 Summary of Contribution ........................................ 5
  1.3 Outline of Dissertation .......................................... 7

CHAPTER 2: BACKGROUND ............................................. 8
  2.1 Data Center Network ............................................. 8
  2.2 Software-defined Networking .................................... 14
  2.3 Application-aware Network ...................................... 19

CHAPTER 3: APPLICATION-AWARENESS IN DATA CENTER
  NETWORK USING OPENFLOW .................................. 22
  3.1 BIC-LSU Project: Introduction ................................ 24
  3.2 BIC-LSU: Framework Design ................................... 27
  3.3 BIC-LSU: Evaluation ............................................. 44
  3.4 BIC-LSU: Related Work ........................................ 48
  3.5 BIC-LSU: Conclusion ............................................ 49
  3.6 MinCOF Project: Minimal Coflow Routing and Scheduling in OpenFlow-based Cloud Storage Area Networks ........................................ 49
  3.7 MinCOF: Motivation .............................................. 52
  3.8 MinCOF: Framework Design ..................................... 53
  3.9 MinCOF: Evaluation ............................................. 64
  3.10 MinCOF: Related Work .......................................... 69
  3.11 MinCOF: Conclusion and Future Works ....................... 70

CHAPTER 4: APPLICATION-AWARENESS IN DATA CENTER
  NETWORK USING FAST PACKET PROCESSING LIBRARY ........ 72
  4.1 Coflourish Project: Introduction ............................. 72
  4.2 Motivation .......................................................... 75
  4.3 Framework Design ................................................ 77
  4.4 Simulation Evaluation ........................................... 89
4.5 Extended Framework for Virtualized DCN .................... 95
4.6 Data Center Network Testbed Construction ................ 97
4.7 Experiment Evaluation ....................................... 100
4.8 Related Work .................................................. 108
4.9 Conclusion ...................................................... 109

CHAPTER 5: CONCLUSION AND FUTURE WORK .......... 111

REFERENCES ...................................................... 113

VITA ................................................................. 122
List of Tables

3.1 Hardware specification of a cache storage server. . . . . . . . . . . . . 30
3.2 Comparison of features of coflow scheduling/routing frameworks. . . 51
3.3 Statistics of coflow inter-arrival time in the Facebook traffic trace. . 66
4.1 Symbols used in algorithms . . . . . . . . . . . . . . . . . . . . . . . . 80
4.2 The PathRem which stores bi-directional available bottleneck band-
widths at the scheduler. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
4.3 Default Framework Parameter Configuration for Simulation. . . . . 92
4.4 Default Framework Parameter Configuration for Experiment. . . . . 102
List of Figures

2.1 FT(m,n) ......................................................... 9
2.2 SUBFT(m,ℓ) .................................................. 10
2.3 SUBFT(m,1) .................................................. 10
2.4 Fat-tree(4,2). ................................................ 10
2.5 Fat-tree(4,3). ................................................ 11
2.6 Fat-tree provides full bisection bandwidth. ................. 12
2.7 DCN with high-bandwidth switches ......................... 12
2.8 DCN with commodity switches in fat-tree topology. ....... 13
2.9 Alternative paths between 2 hosts in fat-tree. ............... 13
2.10 scatter-gather and work-seeks-bandwidth traffic patterns in 10 sec. . 15
2.11 OpenFlow Architecture. .................................. 16
2.12 OpenFlow processing pipeline. ............................ 17
2.13 The packet processing workflow using the fast packet processing library (using DPDK as an example). ................. 18
2.14 The packet processing workflow using the Linux kernel. ....... 19
2.15 The MapReduce cluster computing programming model. Arrows within 1 red circle are flows in 1 coflow. ....................... 21
3.1 The existing network stack and network packet generation. ....... 23
3.2 BIC-LSU software-defined network architecture. ............ 29
3.3 BIC-LSU control framework architecture. .................... 32
3.4 BIC-LSU SDN Maximal Matching path allocation example. .... 36
3.5 The Entity-Relationship Diagram of BIC-LSU Management DB. . . . 39
3.6 The object-oriented architecture of the BIC-LSU Command Service. 40
3.7 Write performance of the RAID 0 on the cache storage server. 2 through 8 unbuffered parallel writes achieve the maximum throughput of 42.4Gb/s.

3.8 Performance of transfer applications. BBCP in unbuffered I/O mode is the overall best. FDT is the best cross-platform choice.

3.9 Performance of BIC-LSU WebIO in the regular campus network.

3.10 Performance of BIC-LSU SDN scheduler.

3.11 (a) The overall architecture of MinCOF. Dashed links carry control messages. To save space, hosts connecting to leaf switches other than $L_n$ are omitted. (b) The functions on the leaf OpenFlow Switches. (c) The functions on the hosts. (d) The slices in links from leaf OpenFlow Switches to spines for prioritizing coflows.

3.12 2 flows must fairly share one path without per-flow rate-limiting. Avg. FCT = $(2+3)/2 = 2.5$.

3.13 2 flows optimally share one path with per-flow rate-limiting. Flows can transfer in arbitrary order. Avg. FCT = $(1+3)/2 = 2$.

3.14 Random schedule of 2 coflows. Avg. CCT = $(2+3)/2 = 2.5$.

3.15 Optimal coflow schedule. Avg. CCT = $(1+3)/2 = 2$.

3.16 Random routing of 2 coflows via 2 alternative paths. Avg. CCT = $(4+5)/2 = 4.5$.


3.18 The worst impact of switching paths on the congestion window size of a TCP flow. The flow re-routes to a faster path at Time $W_o$. Transmission capacity loss because of TCP re-ordering (shaded area) and gain (slashed area) are marked.

3.19 The spine-leaf fat-tree experiment DCN consists of 4 spines, 4 leaves, and 16 hosts. Each link is 100Mb/s.

3.20 The downscaled file size distribution of Data Oasis.

3.21 Impact of Coflow Width. Shorter CCT is better.

3.22 Impact of Coflow Size. Shorter CCT is better.
3.23 Impact of Network Blocking Ratio. Shorter CCT is better.

3.24 Performance under Background Traffic. Shorter CCT is better.

4.1 A spine-leaf fat-tree DCN with 2 spine switches and 3 leaf switches. Each switch in the leaf layer connects to the same number of host machines.

4.2 CCT of the ideal scheduling framework normalized to Varys’.

4.3 Switch feedback mechanism to machine $H_{nh+h-1}$. Phase (1): spine layer feedbacks to leaf layer (solid arrow). Phase (2): leaf layer feedbacks to host layer (dashed arrow).

4.4 The inaccuracy of Coflourish normalized to Varys’. Lower inaccuracy is better.

4.5 The CCT of Coflourish normalized to Varys’. Lower CCT is better.

4.6 The impact of coflow sizes on the CCT of Coflourish normalized to Varys’. Lower CCT is better.

4.7 The impact of feedback intervals on the CCT of Coflourish normalized to Varys’. Lower CCT is better.

4.8 The impact of available bandwidth variation thresholds on the CCT of Coflourish normalized to Varys’. Lower CCT is better.

4.9 Targeted DCN for the extended Coflourish. A physical DCN consists of spines, leaves, and hosts. A virtualized DCN consists of spines, leaves, virtual switches, and virtual machines.

4.10 The 1Gb/s DCN testbed topology in CRON-DC.

4.11 The 10Gb/s DCN testbed topology in CRON-DC.

4.12 The inaccuracy of Coflourish normalized to Varys’. Lower inaccuracy is better.

4.13 The average CCT of Coflourish normalized to Varys’. Lower CCT is better.

4.14 The CCTs of Coflourish under various traffic loads normalized to Varys’. Lower CCT is better.
4.15 The impact of feedback intervals on the CCT of Coflourish normalized to Varys'. Lower CCT is better. . . . . . . . . . . . . . . . 107

4.16 The impact of available bandwidth variation thresholds on the CCT of Coflourish normalized to Varys'. Lower CCT is better. . . . . . 108
ABSTRACT

A data center (DC) has been a fundamental infrastructure for academia and industry for many years. Applications in DC have diverse requirements on communication. There are huge demands on data center network (DCN) control frameworks (CFs) for coordinating communication traffic. Simultaneously satisfying all demands is difficult and inefficient using existing traditional network devices and protocols. Recently, the agile software-defined Networking (SDN) is introduced to DCN for speeding up the development of the DCNCF. Application-awareness preserves the application semantics including the collective goals of communications. Previous works have illustrated that application-aware DCNCFs can much more efficiently allocate network resources by explicitly considering applications needs.

A transfer application task level application-aware software-defined DCNCF (SDDCNCF) for OpenFlow software-defined DCN (SDDCN) for big data exchange is designed. The SDDCNCF achieves application-aware load balancing, short average transfer application task completion time, and high link utilization. The SDDCNCF is immediately deployable on SDDCN which consists of OpenFlow 1.3 switches. The Big Data Research Integration with Cyberinfrastructure for LSU (BIC-LSU) project adopts the SDDCNCF to construct a 40Gb/s high-speed storage area network to efficiently transfer big data for accelerating big data related researches at Louisiana State University.

On the basis of the success of BIC-LSU, a coflow level application-aware SD-DCNCF for OpenFlow-based storage area networks, MinCOF, is designed. MinCOF
incorporates all desirable features of existing coflow scheduling and routing frameworks and requires minimal changes on hosts.

To avoid the architectural limitation of the OpenFlow SDN implementation, a coflow level application-aware SDDCNCF using fast packet processing library, Coflourish, is designed. Coflourish exploits congestion feedback assistances from SDN switches in the DCN to schedule coflows and can smoothly co-exist with arbitrary applications in a shared DCN. Coflourish is implemented using the fast packet processing library on an SDN switch, Open vSwitch with DPDK. Simulation and experiment results indicate that Coflourish effectively shortens average application completion time.
CHAPTER 1
INTRODUCTION AND MOTIVATION

In this chapter, we introduce the development of DCN driven by the advance of the SDN and the communication characteristics from the applications running on DCN. This missing features in existing DCN-based service motivate us to make up for the deficiencies. We summarize our contributions in this study.

1.1 Development of Data Center Network and Motivation

The Data center (DC) is an indispensable infrastructure both in the academia and the industry. DC possesses humongous computing and storage resources which are desirable for researchers to tackle complex problems or for entrepreneur to discover a new business strategy to profit more. People spend lots of effort on researching and optimizing DC to squeeze out any possible throughput. Data center network (DCN) is the most important communication medium that every machine heavily relies on to coordinate with each other. The researches on improving the performance of DCN is always in the spot light. The intense research areas in DCN include load balancing [1, 2], flow completion time shortening [3], controlling in network delay [4], application completion time shortening [5, 6], and increasing network link utilization [7, 8]. Achieving the goals of these rapidly evolving researches using existing network devices is complicated, even not possible yet.

The emerging Software-defined Networking (SDN) allows network engineers to programmatically control the behavior of a network [9]. SDN quickly becomes the ideal alternative for DCN researchers to implement their innovations and conduct experiments on physical network hardware. SDN has different implementations with different granularities and different maturities. For example, the OpenFlow-
based SDN [10] is good at processing network flows and has been deployed in production networks. In contrast, the OpenDataPlane-based SDN [11] is capable of efficiently process each individual low-level transmission unit but is still under construction. Researchers and network engineers have to carefully select the appropriate implementations to adopt according to the target DCN features.

The concept of application-aware network scheduling is recently thriving in DCN [5, 6]. The coflow abstraction [12] is widely adopted to model applications’ communication characteristics. By allocating network resources according to communication characteristics of applications, both average application completion time and 95th percentile application completion time obviously decrease. The legacy design of network protocol stack overlooks the correlations between network flows from applications, such as collaborating parallel flows and flows with a specific finishing order are treated equally as independent irrelevant individual flows. The network devices forwarding individual flows according to local information is likely to result in a sub-optimal network utilization. Application-awareness feature provides a mechanism for applications to inform the network scheduler of the important but missing correlation information. The scheduler has a better chance to produce a schedule close to the global optimum.

With the increasing popularity of big data related research, the demands on the infrastructures for storing, moving, and analysing big data are rising [13]. Government officials also allocate big budget for supporting the development of such infrastructures [14, 15, 16, 17]. This type of big data research infrastructure typically consists of high-performance storage systems at research venues for storing experiment data, high-performance storage systems at high-performance comput-
ing facilities for caching data for analysis, and high-speed storage network for moving data between the former two storage systems. Many research institutes already have the storage systems but the storage network. Their institutional networks are overwhelmed while the storage systems exchange big data sets.

We design a transfer application task level application-aware software-defined DCN(SDDCN) and its corresponding SDDCN control framework (SDDCNCF) for satisfying the demands on the big data research infrastructure. The SDDCN and SDDCNCF are designed to utilize the OpenFlow SDN implementation. We sophisticatedly choose the practically implementable desirable features from state-of-the-art DCN researches and realize them in our DCNCF. The selected features are application-aware load balancing, short transfer application completion time, high link utilization, and compatibility with background traffic. Our design is deployed as the big data research infrastructure “Big Data Research Integration with Cyberinfrastructure for LSU” (BIC-LSU) \[18, 19\] with 40Gb/s storage area network (SAN) link capacity at Louisiana State University (LSU). We also integrate high-performance shared storage systems for researchers who cannot access a private one to profit from the new infrastructure. We develop a user-friendly interface for researchers to launch big data analysis applications on LSU’s high-performance computing facilities.

On the basis of the success of BIC-LSU, we design, MinCOF, a coflow level application-aware SDDCNCF for SANs. MinCOF incorporates all desirable features of existing coflow scheduling and routing frameworks and require minimal changes on hosts. MinCOF is the prototype of the SDDCNCF for the next generation BIC-LSU SAN.
After we recognize the deficiency of instant reaction to rapid traffic variations of the OpenFlow SDN is resulted in by the inevitable communication delay between the OpenFlow switch and OpenFlow controller [10]. This flaw is critical to some desirable features in general DCNs. For example, find-grained application-aware load balancing for regular size traffic flows. We search for an alternative SDN implementation that does not have OpenFlow’s limitation for constructing an advanced revision of our OpenFlow SDDCNCF and find the implementation using fast packet processing library such as Intel Data Plane Development Kit (DPDK) [20]. SDN using fast packet processing library allows SDN switches to have complicated control logics for instantly making forwarding decisions at local. We then search for existing non-OpenFlow SDDCNCFs which accommodate the most features as our OpenFlow SDDCNCF does. Varys [5] and its successor Aalo [6] provide coflow-based [12] short average application completion time, and high link utilization. The coflow is an abstraction which preserves the cooperative relationship of network flows from applications. However, both SDDCNCFs are not compatible with background traffic. They require all hosts in DCN to be under their control. Our simulations show that the interference from background traffic may degrade the performance of the existing coflow-based SDDCNCF up to 80%.

We design an advanced coflow level application-aware SDDCNCF, Coflourish, which can co-exist with background traffic. Coflourish takes congestion feedbacks from SDN-enabled switches in the fat-tree-based DCN [21, 22] for more accurate estimation of available network bandwidth against the interference from the background traffic. Every switch keeps track of the congestion information of every
port. A feedback mechanism periodically aggregates the congestion information from each fat-tree layer for each host machine. A daemon process on each machine participating in the framework converts its collected congestion information into available bandwidth from(to) itself to(from) each fat-tree layer and report the available bandwidths to a logically centralized scheduler. The scheduler can easily synthesize the reported bandwidths and use the smallest bottleneck bandwidth as the overall available machine-to-machine bandwidth. Simulations show that Coflourish is much more resilient to background traffic than the existing coflow scheduling frameworks are. Coflourish is the first coflow scheduling framework which provides detailed algorithms of the SDN-assisted available bandwidth estimation. We implement Coflourish on the Open vSwitch with Intel DPDK fast packet processing library and evaluate it with experiments.

1.2 Summary of Contribution

Our contributions in this study are designing application-aware SDDCNCFs using various SDN implementations in various application-awareness levels. Major contributions are summarized.

1.2.1 Application-aware SDDCNCFs using OpenFlow.

(a) We design a transfer application task level application-aware SDDCNCF to shorten average data transmission time. Its application-awareness feature targets transfer application tasks between a single source transfer application and a single destination transfer application. Simulation results illustrate that our SDDCNCF effectively reduces average data transmission time.
We integrate our SDDCNCF into a campus SAN. Our incidental contributions in the production environment are emphasized as follows.

- We build the BIC-LSU service for accelerating big data related research at LSU. Researchers manipulate big data storages, transmission networks, and analysis facilities via a user-friendly social networking interface.

- We construct a highly extensible software control framework, BIC-LSU Command Service, for controlling widely used big data storages, transmission networks, and analysis facilities. Other research institutes can adapt it and easily build their own cyberinfrastructure.

(b) We further design an advanced coflow level application-aware SDDCNCF for SANs, MinCOF. Its application-awareness feature targets coflows between groups of machines in DCN. Objectives are shortening average coflow completion time and avoiding dependencies on proprietary tools. MinCOF only relies on hardware which conforms open standards and open source software. Experiment results reveal that MinCOF outperforms existing coflow routing/scheduling frameworks.

2. 1 Application-aware SDDCNCF using fast packet processing library.

We design a coflow level application-aware SDDCNCF, Coflourish, to shorten average coflow completion time in a DCN with traffic from some not controlled hosts. The switches in Coflourish coordinate and help estimate available bandwidth in DCN. The switch in Coflourish is implemented on Open
vSwitch with Intel DPDK fast packet processing library. Simulation and experiment results illustrate that Coflourish shortens average coflow completion time.

1.3 Outline of Dissertation

The rest of this dissertation is organized as follows.

In Chapter 2, we provide important background knowledge of DC, DCN, SDN, and application-aware network.

In Chapter 3, we elaborate the transfer application task level application-aware SDDCNCF using OpenFlow and its successive SDDCNCF, MinCOF. The BIC-LSU service built on our transfer application task level application-aware SDDCNCF is also introduced in details.

In Chapter 4, we elaborate the coflow level application-aware SDDCNCF, Coflourish, using fast packet processing library.

In Chapter 5, we conclude our works and suggest future development directions.
CHAPTER 2
BACKGROUND

In this chapter, we supply important background knowledge which is referred to in later chapters.

2.1 Data Center Network

A data center (DC) is a set of clustered computing or storage resources interconnect with a communication network[13]. A modern DC provides enormous computing power and storage space by aggregating a large amount of commodity servers. The communication network in a DC is a data center network (DCN). The applications running in a DC mostly consists of multiple software components running on multiple servers. The software components coordinate by exchanging a large amount of information across DCN.

2.1.1 Topology

Since all servers in DC rely on DCN to communicate, the topology of a DCN must provide high bandwidth for fast data transmission, high availability for reliable transmission, and good scalability for easy increment of servers. Mohammad et al [23] brings the fat-tree from the classical Clos topology [24] family in the telecommunications network to the modern data center to achieve 3 targets altogether. Fat-tree has been widely deployed in DCNs.

The fat-tree network is a recursive structure built with identical commodity switches. We adopt the notations for fat-tree structure explanation from [25], fat-tree FT(m,n) and sub-fat-tree SUBFT(m,ℓ) where m denotes the number of ports on each switch and n,ℓ denotes the number of levels of switches in the tree. Each link has the same bandwidth. Each link is bi-directional. Guidelines for constructing
a fat-tree are (1) Every FT and SUBFT has switches in the root level to connect all sub-trees and (2) Every sub-tree in FT and SUBFT connects to every root-level switch in the next higher level tree. On a root-level switch of FT, all ports are used to connect sub-trees. Figure 2.1 illustrates the recursive construction of a FT(m,n). On a root-level switch of SUBFT, half of the ports are used to connect sub-trees and the other half are used to connect the root-level switches in the next higher level tree. Figure 2.2 presents the recursive construction of a SUBFT(m,\ell).

SUBFT(m,1) shown in Figure 2.3 uses \( \frac{m}{2} \) ports to connect \( \frac{m}{2} \) hosts and the other \( \frac{m}{2} \) ports to connect to higher level.

Typically, a fat-tree DCN does not exceed 3 levels in height to keep the network diameter low for lowering the host-to-host delay. A 2-level fat-tree topology has 2 layers of switches, core and edge (or spine and leaf, also known as spine-leaf topology). A FT(4,2) is in Figure 2.4. A 3-level fat-tree topology has 3 layers of switches, core, aggregate, and edge. A FT(4,3) is in Figure 2.5.
FIGURE 2.2: SUBFT(m, ℓ)[25].

FIGURE 2.3: SUBFT(m, 1)[25].

FIGURE 2.4: Fat-tree(4,2).
For achieving high bandwidth, the fat-tree aggregates bandwidth from multiple physical paths between switches and hosts to form a logical high bandwidth links. There are \( \binom{m}{2} \) physical shortest paths between 2 hosts residing in one common smallest FT(m,i) or SUBFT(m,i) of FT(m,n). Thus, each switch can be an inexpensive commodity switch with low bandwidth but still achieve the lowest oversubscription as 1:1. The oversubscription is the ratio of the worst-case achievable aggregate bandwidth among the end hosts to the total bisection bandwidth of a particular communication topology. Figure 2.6 shows paths used when a half of the hosts communicate at full bandwidth with the other half. 4 colors represent the paths used by 4 pairs of communicating hosts. The cost of the fat-tree DCN is much lower compared to DCNs with single high-bandwidth carrier-grade switches. Figure 2.7 shows a DCN constructed with high-bandwidth switches and Figure 2.8 shows fat-tree DCN having equal bandwidth between hosts with low-bandwidth commodity switches.
Figure 2.6: Fat-tree provides full bisection bandwidth.

Figure 2.7: DCN with high-bandwidth switches[25].

For achieving high availability, the fat-tree intuitively takes advantage of the multiple physical paths to provide redundant alternative paths between switches. Figure 2.9 highlights all alternative paths between 2 hosts with dashed lines.

For achieving good scalability, the fat-tree adds sub-trees or expands its core level and recursively add sub-trees if the old fat-tree is full. An FT(m,n) fat-tree connects at most $m^{(n-1)}$ hosts.

In production DCNs, some modifications and relaxations are applied to the fat-tree topology for practical concerns: (1) Links between switches have higher bandwidth than links between switches and hosts so that the number of actual switches and cables are reduced much. The ease of maintenance increases; (2) DCN oversubscription is increased to between 2.5:1 to 10:1 [23, 26] to greatly reduce
FIGURE 2.8: DCN with commodity switches in fat-tree topology[25].

FIGURE 2.9: Alternative paths between 2 hosts in fat-tree.

cost; (3) The spine-leaf topology is most widely used because of its simplicity and low host-to-host delay. The latter parts of this study focus on the spine-leaf topology.

2.1.2 Traffic Characteristics

Previous observations [27, 28] identify the following traffic characteristics in DCNs and implies many DCNCF design principles.

- **80% flows are smaller than 10KB** A DCNCF must adopt scheduling algorithms that profit small flows. For example, shortest job first and least attained first. However, with more bandwidth-hungry services such as high-definition video streaming [29] migrate to DCN, the weight of huge flows increases.
• traffic is bursty and has ON/OFF pattern A DCNCF must provide load balancing to evenly distribute traffic load to alternative paths in DCN fabric especially during the ON period. Flowlet is the ideal granularity. Flowlet is a burst of packets which is at least one Round Trip Time (RTT) behind the previous burst.

• 99% traffic is TCP A DCNCF must avoid events that decrease TCP’s throughput when scheduling. For example, transferring segments in order to avoid reordering, and alleviating congestion early to avoid timeout.

• Regular traffic patterns are scatter-gather and work-seeks-bandwidth

A DCNCF must schedule traffic according to the correlation of individual communications between host pairs. Application-aware scheduling may improve the throughput. The diagonal in Figure 2.10 represents the work-seeks-bandwidth. Hosts communicate with hosts nearby to utilize high bandwidth and low delay. The horizontal and vertical lines in Figure 2.10 represent the scatter-gather. A subset of hosts communicates with another subset for collaborations. However, with more customized applications running in DCN, these patterns are less obvious.

2.2 Software-defined Networking
2.2.1 Definition of Software-defined Networking

“An architecture decouples the network control and forwarding functions enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services.” [9]

Network engineers are able to impose their own control logics on the abstracted network by writing a control program. The tedious waiting for the vendors to
implement a desired feature on their proprietary network devices no longer hinders evolution of the network architecture. The control program is also portable to any SDNs that support the same abstraction interface.

There are several implementations of the SDN concept. We only introduce the open source ones in each category.

2.2.2 Complete Implementation

In this category, OpenFlow has both the network control and forwarding function in the SDN definition. The network control is implemented as an OpenFlow controller program running on a remote host. The forwarding function is implemented as an OpenFlow switch. The OpenFlow switch and controller communicate with the OpenFlow protocol on a secure OpenFlow channel. Figure 2.11 presents the overview of OpenFlow architecture.

The definition of a flow is “A series of packet with identical values in specific header fields within a certain amount of time”. The workflow of packet processing
inside an OpenFlow switch is described in Figure 2.12. If there is not any action instruction to execute at the end of the pipeline in Figure 2.12(a) or the action instruction in the matching flow entry in Figure 2.12(b) is “send packet to the port to the controller”, the current packet is sent to the OpenFlow controller via the secure OpenFlow channel. The controller decides the action instruction to execute and notify the switch through the OpenFlow channel again. The switch executes the action instruction to process the current packet. The controller can install the header fileds and the action instruction as a new flow table entry to the switch for later matching.

The OpenFlow SDN is flexible for creating new features but lack of existing building blocks for composing production-grade network functions. The applicability of OpenFlow is also restricted to fixed set of header fields supported by OpenFlow switch hardware. The following 2 advanced extensions, Open Network Operating System(ONOS)[30] and P4[31], emerges and largely improve the usability of the OpenFlow-based SDN.
ONOS is a distributed SDN applications platform with fault tolerance and numerous network functions in existing networks as dynamically loadable applications. It controls underlying network devices using SDN implementations such as OpenFlow. Network engineers can efficiently build robust and versatile network control functions for SDN.

P4 is a language for programming protocol-independent packet processors. It is capable of defining the format a switch uses to parse a packet, and the logic a switch uses to process a packet [32]. Network engineers can design a totally customized network forwarding device using P4 and realize it on a switch that supports the P4 compilation output.
2.2.3 Fast Packet Processing Library Implementation

In this category, only the forwarding function is implemented. In plain English, the fast packet processing library is used as the building block to create an SDN switch. The library bypasses the network stack in the operating system kernel to process network packets in a user application. The bypassing avoids the overhead of the execution mode switching between the kernel mode and the user mode. Versatile other libraries in the user space can be used. There are 3 implementations, Intel Data Plane Development Kit (DPDK), netmap, and OpenDataPlane. Figure 2.13 presents the packet processing using a fast processing library (using DPDK as an example). Figure 2.14 presents the conventional packet processing in the Linux kernel.

![Diagram](image)

FIGURE 2.13: The packet processing workflow using the fast packet processing library (using DPDK as an example) [33].

netmap is a framework for fast packet I/O similar. It memory-maps network packets directly from kernel-space to user-space for customized processing [34]. Any user-space libraries can be used to enhance packet processing.
Intel Data Plane Development Kit is similar to netmap [20] but includes more network related helper functions such as hash table and longest prefix matching. The widely used Open vSwitch [35] has a branch built on DPDK named “Open vSwitch with DPDK” [36].

OpenDataPlane (ODP) is a standard interface for packet processing libraries on network devices. It converts the interface to the native packet processing libraries to a standardized abstract interface [11]. The underlying implementation may be DPDK or netmap. An application compatible with ODP is portable to any network device that provides ODP interface.

2.3 Application-aware Network

Applications generate communication flows to coordinate with each other to accomplish some logical objective. To reduce communication time, applications usually generate parallel communication flows to exploit concurrency. In the conventional computer network environment, the application’s logical objective is hidden.
from any intermediate network devices between the source and destination. Parallel flows for a collective logical objective are scheduled as independent entity in the network and optimized individually. However, optimal scheduling of individual flows may lead to a sub-optimal scheduling for their collective objective. A network that is able to perceive the collective objective is application-aware network. Application-aware networks have deeper understanding of applications requirements so that it is able to more efficiently shorten applications’ communication time.

Coflow [12] is an abstraction of application’s communication behavior. It is used for precisely keeping track of all flows for one logical objective and the source/destination of each flow. Figure 2.15 illustrates the flows and coflows in an application using the MapReduce cluster computing programming model [37]. All arrows within 1 red circle are the communication flows of 1 communication stage. The MapReduce application continues to the right only when all flows in the current communication stage finish. All flows in one communication stage are modeled as flows of 1 coflow. A coflow scheduler should schedule network resources to finish all flows at close instants of time. Previous works show that network scheduling based-on coflow is more efficient than existing flow-based scheduling in DCN.
Background: Cluster Computing

Frameworks

Cluster computing frameworks greatly simplify cluster application development (e.g., MapReduce).

Communication time contributes up to 70% of application completion time [Orchestra, 2011].

FIGURE 2.15: The MapReduce cluster computing programming model[38]. Arrows within 1 red circle are flows in 1 coflow.
CHAPTER 3
APPLICATION-AWARENESS IN DATA CENTER NETWORK USING OPENFLOW

This chapter focuses on developing application-aware SDDCNCFs for DCNs built using OpenFlow SDN. OpenFlow is the most mature SDN implementation. OpenFlow SDN has been deployed in production DCNs [39, 40] for its easeiness in directing flows among multiple alternative paths.

Applications in DCNs typically utilize parallel communication flows to increase throughput and reliability. However, the underlying network switch is not able to identify different groups of parallel flows of different applications because information required for differentiating the affiliation of flows is not carried in existing network packet headers. The existing network stack was originally created for the Internet following the layered approach. The network layers below the application layer mainly target carrying a communication flow from the source application process to the destination application process. Figure 3.1 illustrates the existing layered network stack, the function of each layer, and the network packet generation process. The existing network stack does not provide an interface for the application to insert the relationships between its parallel flows and collective communication requirements into lower layer network packet header fields. Partial application semantics are sealed in the application data and then in the Transport Layer (L4) Payload. OpenFlow distinguishes flows only by matching on existing network packet header fields, Link Layer (L2) header field, Network Layer (L3) header field, and L4 header field, so that OpenFlow SDN is intrinsically only partially application-aware by inferring application semantics using information exposed in the packet header fields.
FIGURE 3.1: The existing network stack and network packet generation.

To achieve fully application-aware network scheduling for applications with parallel flows using OpenFlow, applications have to explicitly provide additional information through some out-of-band channels. For example, an application notifies the OpenFlow Controller of all parallel flows’ header field definitions by directly sending a specialized message for a network resource schedule which considers all flows as the basic unit.

There is no previous work which achieves application-awareness of applications with parallel flows in an OpenFlow-based DCN because previous works solely use the network packet header fields to manage flows thus overlook the logical relationships between flows in the application.

We develop 2 SDDCNCFs which achieves application-awareness of applications with parallel flows by tightly integrating applications and network in 2 different granularities. One is at the transfer application task level 3.1 and the other,
named MinCOF, is at the coflow level 3.6. The former SDDCNCF is adopted by the Louisiana State University to provide the BIC-LSU service. The MinCOF will be the foundation of the next generation of the BIC-LSU service.

The contents from Section 3.1 through 3.5 are from our publication “BIC-LSU: Big Data Research Integration with Cyberinfrastructure for LSU” [18]. The contents from Section 3.6 through 3.11 are from our publication “Minimal Coflow Routing and Scheduling in OpenFlow-based Cloud Storage Area Networks” [41].

3.1 BIC-LSU Project: Introduction

Big data technologies recently have gained enormous popularity in academia because they can help researchers explore invaluable information hidden in big data that is difficult to be observed with traditional research methods. Even though big data technologies are promising in many research fields, researchers conducting big data analysis on campus typically face two significant obstacles: 1) inadequate network infrastructure for transferring big data and 2) non-trivial user interface for launching big data analysis applications. A big data analysis application typically starts with transferring a huge amount of data from a local storage system to a high-performance computing (HPC) facility. For example, a coastal research group uploads a large-scale data set collected from its seashore observatory to an XSEDE HPC cluster [42] to predict the erosion of the coastline. Such a big data transfer can impose a significant load on the network infrastructure on campus. In addition, given that inter-disciplinary research collaboration is widespread in research universities, researchers often need to exchange big data among geographically distant research groups through the campus network for conducting big data analysis in various research domains. For example, in a collaborative genome sequencing
research project, a biology research group with a next-generation sequencing machine generates a massive genome data set and then transfers it to a statistics research group for error correction and filtering. Next, the statistics research group transfers the filtered data set to a computer science research group for sequence reconstruction using sequencing applications such as ABySS [43] and BLAST [44]. Such frequent big data exchanges among different research groups are putting more pressure on the campus network infrastructure.

The existing network infrastructures in a large portion of research institutes are not designed for frequent big data transmission. Transferring big data via a regular network is likely to suffer from network congestion events, and big data analysis process is stalled at the beginning. To alleviate this limitation, a new high-performance network infrastructure has to be constructed [14]. Achieving high-performance big data transmission involves four factors: (1) a network which transfers bits at high bandwidth, (2) a network scheduler which efficiently allocates network bandwidth to data transmission tasks, (3) storage devices on end hosts which can send and receive at full network bandwidth, (4) transfer applications on end hosts which keep up both with the bandwidths of the network and the storage devices. However, each existing IT infrastructure that supports big data transmission has at least one weak point within the above four factors. For example, the transfer application in the widely used Globus Toolkit [45], GridFTP, cannot saturate a single 10Gb/s network link from a single host [46]. Detailed performance evaluations are in Section 3.3.

The second step of the big data analysis is launching big data analysis applications on the high-performance computing (HPC) facility to process the data. The
standard user interface that most HPC facilities provide is the primitive command line interface (CLI). For simply starting an application, a researcher must have some level of proficiency for using the primitive interface to (1) prepare big data in the appropriate format and location and (2) submit a job scheduling request script to the job scheduling sub-system to acquire computing resource to run the application. To gain such required proficiency, researchers need to spend much precious time on training, instead of focusing on their research works [47].

There are several portal systems that greatly reduce the required training for using the HPC facilities. For example, XSEDE incorporates many science gateways which have comprehensive web interface to help researchers in specific fields of science easily launch big data analysis applications [48]. Most portal systems have fixed targeted applications thus cannot be deployed as a generalized solution for most use cases. Fortunately, the DiaGrid[49] makes a big step forward to allow researchers to add their own web user interface for an application to the portal system and share with others. However, there are numerous existing job scheduling request scripts at work, and converting each of them into a web user interface using the toolkit provided by the DiaGrid requires a long manual process[50]. A swift way to convert existing job scheduling request scripts into web user interfaces is still lacking.

We construct a new IT infrastructure for big data migration and analysis in the project titled “Big Data Research Integration with Cyberinfrastructure for LSU” (BIC-LSU) at Louisiana State University (LSU). The BIC-LSU project synthesizes the preferable features of the existing big data transmission and analysis infrastructures. For example, Single Sign-On (SSO), comprehensive web user interface,
and convergence of control on heterogeneous systems are provided in BIC-LSU. The BIC-LSU project makes up the deficiencies of the existing big data transmission infrastructures and the HPC user interfaces mentioned above. BIC-LSU is designed to be highly extensible and can easily be augmented to existing IT infrastructures.

3.2 BIC-LSU: Framework Design

The design principles of BIC-LSU are as follows:

- **High-performance** The BIC-LSU infrastructure must be able to consistently utilize the hardware at nearly maximal performance.

- **Efficient network utilization** Several network related researches [7, 51, 5] indicate that adding simple Shortest Job First-like scheduling greatly shortens network communication time. The BIC-LSU infrastructure should take advantage of these findings.

- **Least intrusiveness** BIC-LSU infrastructure is designed to easily augment to existing IT infrastructures. BIC-LSU software should have least interference with and dependency requirements on integrated computer systems.

- **Extensibility** BIC-LSU network should easily expand with the growing number of connected computer systems. BIC-LSU software should be easily expandable for supporting emerging storage and HPC systems.

- **User-friendliness** BIC-LSU infrastructure should provide a user interface that is intuitive and comprehensive to effectively hide the complexity of the background systems from researchers.
The size of the big data that BIC-LSU targets is a synthesis of previous research work [52], publicly accessible big data sources [53, 54, 55], and a survey from researchers who conduct big data analysis at LSU. The targeted big data size scatters from 10GB to 128GB.

The BIC-LSU system components we deploy and develop include multiple high-performance shared storage servers, a high-bandwidth OpenFlow Software-defined Network (SDN), customized high-performance transfer applications which closely coordinate with the SDN for optimized network utilization, a light-weight control framework, BIC-LSU Command Service, that controls all control interfaces of all software systems which involve in the BIC-LSU, and a user-friendly social networking web portal.

3.2.1 Customized High-performance Transfer Applications

We customize two popular transfer applications, BBCP [56] and Fast Data Transfer (FDT) [57], to keep the I/O performance of BIC-LSU at maximum and enable the network resource scheduling optimization in the OpenFlow SDN. These two transfer applications also have the least dependency requirements and can run on most operating systems (OSs) after simply being downloaded. They do not interfere with the operation of the existing software systems on the host so that they are suitable for rapid large-scale deployment on existing heterogeneous hosts.

For maximizing I/O performance, we adopt the existing optimization designs which the 2 applications successfully provide including unbuffered I/O for avoiding the virtual memory buffering overhead and creating an exclusive thread for each storage device for avoiding the context switching overhead.
FIGURE 3.2: BIC-LSU software-defined network architecture.

For optimizing network resource scheduling, we enhance the transfer applications to inform the SDN controller of all TCP connections used in each data transmission task so that the controller can schedule them as a logical unit. When a researcher starts a data transmission task via the web portal of BIC-LSU, the transfer application at the destination wraps the information of all connections in the current task, the size of the task, and a security validation code in one UDP request packet and sends it to the SDN controller. The controller schedules exclusive network resources only for tasks with valid requests.

3.2.2 High-performance Shared Cache Storage Server

To allow any researchers who intend to explore big data analysis but do not have competent computer hardware to store and transfer big data, the researchers who are in the “long tail of science” [58], we deploy multiple shared high-performance storage server, each is equipped all solid-state drive (SSD) RAID 0, as cache servers. In the rest of this paper, we refer to this type of storage servers as “cache storage server”. The hardware specification is in Table 3.1. Each cache storage server
TABLE 3.1: Hardware specification of a cache storage server.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2x Intel Xeon E5-2670 2.6GHz 8-core 16-HT</td>
</tr>
<tr>
<td>Memory</td>
<td>16x 16GB DDR3 1600 MHz</td>
</tr>
<tr>
<td>RAID 0</td>
<td>1x PMC Adaptec RAID 71605</td>
</tr>
<tr>
<td></td>
<td>12x Samsung 840EVO 1TB Solid-state Drive</td>
</tr>
<tr>
<td>NIC 1</td>
<td>1x Intel XL710 1x 40Gb/s port</td>
</tr>
<tr>
<td>NIC 2</td>
<td>1x SuperMicro AOC-STG-I2t 2x 10Gb/s port</td>
</tr>
</tbody>
</table>

connect to the OpenFlow SDN with the 40Gb/s network interface card (NIC) to migrate big data between itself and other big data storage systems. The other 10Gb/s NIC is used for transferring data between researchers’ personal computers and cache storage servers via the regular campus network. To avoid network throughput degradation resulted in by the TCP synchronization [59, 60] in a shared network link carrying less than 500 TCP flows, we deploy the Desynchronized Multi-channel TCP (DMTCP) [60, 61] on each cache storage server.

We deploy an Intelligent Rule-oriented Data System [62] (iRODS) on the cache servers to manage all data as they are on a single logical storage space. Locating a file in iRODS is much easier than locating the file on multiple independent cache storage servers. iRODS is popular in many research institutes such as NASA Atmospheric Science Data Center, The National Institute for Computational Sciences, and Welcome Trust Sanger Institute[63] so that it has a matured mechanism for federation between different autonomous iRODS systems, in the term of iRODS, “Zones”. Currently, LSU Baton Rouge campus is a zone consisting of the cache storage servers. If some LSU researchers want to share big experiment data with another research institute which also has iRODS, BIC-LSU can easily federate with it and both sides just see the data in one shared simple logical storage space.
Although iRODS is ideal for managing data on our cache storage servers, we find that it cannot fully utilize the bandwidth of the RAID 0 on each server when moving big files. Thus, we adopt our more efficient customized transfer application, BBCP, for the task. BBCP is able to fully utilize the bandwidths, 40Gb/s, of our dedicated OpenFlow SDN and RAID 0 on cache storage servers.

Another deficiency of iRODS is at its client for uploading/downloading. Its default client application interrupts the SSO experience that the BIC-LSU provides and cannot fully utilize the bandwidth of the regular campus network. We implement our own Java Web Start client application, BIC-LSU WebIO, which can transfer data using SFTP or FDT directly between a researcher’s personal computer and the cache storage server which runs iRODS. When using SFTP, the BIC-LSU WebIO transfer files with its built-in JSch[64] SFTP. When using FDT, the researcher has to download FDT, generate the command line for launching FDT with the help from BIC-LSU WebIO, and launch the FDT with the generate command line to transfer files. Our client preserves the SSO experience and much more efficiently utilizes the campus network. The upper left portion of Figure 3.3 illustrates the procedure of uploading/downloading using the BIC-LSU WebIO.

3.2.3 High-bandwidth OpenFlow Software-defined Network

**Physical Network Construction:** We use OpenFlow switches to construct a new dedicated high-bandwidth OpenFlow SDN for big data migration. OpenFlow is an open standard for programmatic switch control so we can develop our own OpenFlow controller program to manage any OpenFlow switches. By operating an OpenFlow-based network, we do not have to worry about our new network being locked down to a specific switch manufacturer.
We build a 40Gb/s 2-level Clos[65] topology with 2 core switches and 6 edge switches. The core switch has 32 40Gb/s ports, and the edge switch has 48 10Gb/s ports and 4 40Gb/s ports. They all comply with the OpenFlow 1.3 standard. All OpenFlow switches are controlled by a logically centralized OpenFlow controller at LSU Information Technology Services (ITS). The 6 edge switches are deployed to buildings where big data research groups locate. Each edge switch uses one 40Gb/s port to connect to one cache storage server for all researchers in the building. The OpenFlow SDN is constructed as illustrated in Figure 3.2. Each edge switch still has 2 40Gb/s ports and 48 10Gb/s ports available for researchers’ private storage servers, storage clusters, computing clusters, etc. In the rest of this paper, we refer to any private system which connects to BIC-LSU SDN as a partner system. To make the network scheduling less complicated, the requirement for a partner...
system is that it must have 10Gb/s connectivity. The BIC-LSU SDN leaves small network traffic to the regular campus network.

The advantages of constructing a Clos network are the extensibility and availability. For the extensibility, if the Clos topology is extended to full, it can connect 1440 private systems with 10Gb/s connectivity and 60 partner systems with 40Gb/s connectivity. LSU does not have to worry about the network capacity for big data migration in the near future. For the availability, there are always 2 physical paths between any pair of edge switches. No single point of failure can unexpectedly disable the network.

The SDN connect to LSU HPC facilities through the gateway router of LSU HPC clusters. For example, the SuperMIC, an XSEDE computing resource, and SuperMike II, etc. The SDN connects to the Internet2 through the LSU border router and the Louisiana Optic Network Initiative (LONI)[66] network. All network traffic for inter-institutional federation goes through this border router.

OpenFlow Controller Development: The BIC-LSU OpenFlow controller provides 4 desirable features: (1) minimal maintenance requirement, (2) controller high-availability, (3) transfer application task level load balancing, and (4) short average transfer application task completion time. We adopt the Ryu SDN framework as the basis of the controller to exploit the rapidity of prototyping using Python.

The BIC-LSU OpenFlow controller achieves minimal maintenance requirement using the OpenFlow Discovery Protocol (OFDP)[67]. When the network is switched on or experiences any topology changes, the controller automatically re-discovers the latest topology of the SDN using the OFDP. The controller builds a network
graph which reflects the network topology. Then, the controller constructs a Link Layer (L2) spanning tree to all switches, cache storage servers, and all partner systems. After that, the controller is ready to serve any traffic requests. Administrators do not have to be involved.

We implement the controller high-availability by duplicating any OpenFlow messages and other packets for the primary controller to a standby controller. Each controller independently updates its state. Only the primary actually sends OpenFlow replies to switches. The primary controller periodically sends a heartbeat to the standby controller. If the heartbeat stops for more than 5 intervals, the standby controller elevates itself to be the primary and starts to reply.

The BIC-LSU SDN provides transfer application task level load balancing and short average transfer application task completion time. They are coupled and achieved at the same time by the controller. These two features heavily rely on the customized transfer application. To provide the two features, we perform the following pre-configurations. We pre-slice each physical link in the SDN into a fixed number of small logical channels with equal minimal bandwidth guarantees. We define 2 types of flows, unmanaged and managed. A managed flow belongs to a task which is scheduled by the controller. An unmanaged flow belongs to a task which is not yet scheduled. Only one small channel in each physical link is used to carry unmanaged flows. Thus, managed flows can utilize most bandwidth of the network.

When a transfer application task starts by the customized transfer application, its information is reported to the controller as a scheduling request. The controller classifies requests based on total data size into 3 queues, \( \leq 30\text{GB} \), \( \leq 60\text{GB} \), and
> 60GB according to statistics of use cases at LSU. Each queue sorts requests by their arrival times. All flows belong to the current task are unmanaged flows and go through the single small channel for unmanaged flow along the L2 spanning tree. At the same time, the controller iteratively dequeues scheduling requests using a Deficit Round Robin (DRR)[68] algorithm. We dictate the controller to maintain a 3:2:1 ratio when it dequeues requests. If a task can be given a path across the source edge switch, core switch, and destination edge switch, it is scheduled. Otherwise, it is given a higher priority to be scheduled the next round to prevent starvation. This bounded shortest job first scheduling ensures short average transfer application task completion time. All flows in a scheduled task are assigned to a path which consists of exclusive small channels across the network. The controller can efficiently keep track of the exclusive small channels from hosts to an edge switch to ensure that traffic load from hosts does not exceed the capacity from the edge switch up to the core switches with Maximal Matching[69]. The controller then chooses the least used path to balance the load in the core part of the network. After the entire path from the source host to the destination host is determined for the transfer application task, the controller adds the flows of the task to every switch using the Transport Layer flow definition. At this moment, the task is scheduled and its flows are managed. The detailed algorithm is presented in Algorithm 3.2.1. Line 2 and 3 make sure that source and destination have the capacity to accommodate the transfer application task. Line 4 balances load on switches in the network core. Line 5 to Line 8 give higher priority at the next round of scheduling to the task which cannot be scheduled at this round to
prevent starvation. After one round of scheduling, the controller adds actual flows to OpenFlow switches according to the reservations by Line 9.

**Algorithm 3.2.1** BIC-LSU SDN path allocation.

1. for all $t$ in not promoted task requests in DRR order do
2. Find a channel from src host to src edge switch
3. Find a channel from dst edge switch to dst host
4. Find a least used core switch to connect src to dst
5. if any channel not found then
6. Promote $t$ to next higher priority request queue
7. go to next iteration
8. end if
9. Reserve all channels found
10. end for

Figure 3.4 gives an example of the matching scheduling algorithm. For simplicity, the example network has a 2-level Clos topology with core switches $C_1$ and $C_2$, edge switches $E_1$, $E_2$, and $E_3$, and hosts $H_{11}$, $H_{21}$, $H_{22}$, $H_{23}$, $H_{31}$, and $H_{32}$. The solid lines are the channels for managed flows. The network is idle. 3 requests arrive at the 3 DRR queues respectively: $H_{11} \Rightarrow H_{21}$, $H_{31} \Rightarrow H_{22}$, and $H_{32} \Rightarrow H_{23}$. $H_{11} \Rightarrow H_{21}$ gets the path $H_{11} \Rightarrow$ rectangle route $\Rightarrow C_1 \Rightarrow$ circle route $\Rightarrow H_{21}$. Then, $H_{31} \Rightarrow H_{22}$ gets the path $\Rightarrow H_{31} \Rightarrow$ diamond route $\Rightarrow C_2 \Rightarrow$ triangle route $\Rightarrow H_{21}$.
$H_{32} \Rightarrow H_{23}$ cannot be scheduled because Line 4 fails to find a core switch that connects $E_2$ to $E_3$. $H_{32} \Rightarrow H_{23}$ is inserted back to the next higher priority queue for the next round of scheduling. At last, the 2 allocated paths are established on the switches along the paths.

3.2.4 Light-weight Control Framework

After physically connecting all diverse partner systems, we develop a control framework to manipulate each of them. Because of the heterogeneity of the partner systems, the control framework must have least dependency requirements and can easily be extended to adapt to any new type of partner systems.

The light-weight control framework also maintains its own user identities and group affiliations and accordingly enforces fundamental directory-level access controls. Directories on different partner systems can easily and flexibly be shared.

To avoid depending on special interfacing utilities be pre-installed on partner systems, BIC-LSU manipulates partner systems with their own native user interface. A light-weight control framework, BIC-LSU Command Service, which exactly knows the most appropriate approach to access each partner system is carefully constructed. The design philosophy is “BIC-LSU Command Service directly manipulates partner systems as their remote clients. If a remote client is lacking, BIC-LSU Command Service indirectly connects to an external access host and manipulates from there with the partner system’s native CLI”. While BIC-LSU Command Service has to indirectly use the CLI of a partner system, the BIC-LSU Command Service takes advantage of the widely available SSH to establish a secure communication channel. BIC-LSU Command Service proactively logs in partner systems using SSH Certificate authentication. Partner systems do not have
to alter system configurations. A sophisticated management database, BIC-LSU Management DB, shown in Figure 3.5 is designed to keep track of the procedure for invoking the most appropriate remote client for each partner system. Figure 3.3 illustrates the architecture of the BIC-LSU control framework. The names above the dashed elbow connectors are the native interfaces adopted by the BIC-LSU Command Service. Please note that the SSD icons represent the cache storage servers. To save the space, only 3 out of the 6 cache storage servers are presented in the figure.
FIGURE 3.5: The Entity-Relationship Diagram of BIC-LSU Management DB.
FIGURE 3.6: The object-oriented architecture of the BIC-LSU Command Service.
The architecture of BIC-LSU Command Service is designed for extension to adapt to new partner systems. It strictly follows object-oriented design. 5 base classes accommodate the core logical services required by BIC-LSU, which are the immediate children of the root, POSIX Shell File Operation, Job Control, Remote Login, and Authentication. The logical services are realized in derived classes using a more specific and efficient control operations. The class hierarchy UML diagram is shown in Figure 3.6. The immediate children of the root other than the 5 core logical services are utilities used by the cores.

To enforce access controls of various partner systems for flexible resource sharing among federated researchers, the BIC-LSU Command Service also maintains its own user identity, as BIC-LSU User, and group, as BIC-LSU Group, affiliation and enforces a directory-level access control. A BIC-LSU User has multiple resources on various partner systems. Federated BIC-LSU Users forms BIC-LSU Groups. A BIC-LSU User can share a directory on a partner system to other BIC-LSU Users or Groups by configuring the access control list (ACL) which exists only in BIC-LSU Command Service not in any partner systems. BIC-LSU Command Service serves an access request by checking the request against the access privilege on the ACL and accessing the shared resource using the native user identity of the owner on the partner system if the request is legal. BIC-LSU Management DB preserves all access control regulations.

It is trivial that BIC-LSU consisting of many partner systems is a distributed system. Concurrent operations may interleave and result in catastrophe such as data inconsistency and data corruption. BIC-LSU introduces a logically centralized coordinator to organize all operations in the system. After requested by users via
the web portal, every operation is associated with a task ID. Every task has to receive a grant from the coordinator before starting. The coordinator keeps track of all tasks and their target path. Tasks are granted in a first come first serve order. If two tasks conflict, the early task is granted. The properties of transaction, atomicity, consistency, isolation, and durability (ACID) are preserved.

3.2.5 Social Networking Web Portal

The user interface of BIC-LSU is a web portal built on an open source social networking engine, Elgg[70]. The Elgg is customized to seamlessly inter-operate with the myLSU campus service portal[71]. It allows access from any researcher who is authenticated by the LSU Shibboleth[72] SSO mechanism. We synchronize the user and group on Elgg with the BIC-LSU User and BIC-LSU Group on the BIC-LSU Command Service to provide access control using social networking user identities. We develop two Elgg plug-in modules, the “Big Data Manipulation” module and the “Big Data Analysis” module to provide all big data related functions.

To provide access control using social networking user identities, we tightly synchronize every user and group related operation with its counterpart operation in BIC-LSU Command Service on BIC-LSU User and BIC-LSU Group. When a researcher shares resource with other Elgg users or groups, this access control change is mapped to an ACL entry change in the BIC-LSU Command Service.

The Big Data Manipulation module provides a universal web file browser to access any storage space on partner systems. The module calls the BIC-LSU Command Service to retrieve the content of the storage space on a storage space or make changes to the content. The relationship between the web portal and the BIC-LSU Command Service is shown in Figure 3.3. All operations are actually
performed by the BIC-LSU Command Service. The web portal just converts a researcher’s request to a proper BIC-LSU Command Service call.

The Big Data Analysis module provides a universal job management interface to all supported job scheduling systems. Researchers supply their parameters for launching big data analysis applications via comprehensive web page interfaces, the parameters are inserted into corresponding job scheduling request scripts in the back-end, and the request scripts are submitted to the job scheduling systems to start the analysis applications. This module follows the same design principle as the Big Data Manipulation module. The web portal just displays. All operations are done by the BIC-LSU Command Service.

In addition to generating the job scheduling request script from the web page interface, our Big Data Analysis module provides the reverse as well. Generating the web page interface from a job scheduling request script is desirable because there are many existing job scheduling request scripts currently working well at HPC facilities. It is preferable that a converter can translate those request scripts into web pages so that many users who are not proficient at writing scripts can easily reuse them. However, the converter is currently lacking.

We design a set of simple conversion directives and a corresponding parser to help researchers easily convert an existing job scheduling request script to an HTML form that asks for parameters as form inputs. A directive explicitly specifies what HTML form element to use for asking for a specific parameter. Researchers simply replace the actual values of parameters with the appropriate directives in their existing job scheduling request scripts, upload the scripts to the web portal, run the parser on the modified request scripts, and HTML forms are generated for anyone
to submit the same jobs with their own customized parameters. An example job scheduling request script for a specific application looks like the following.

```plaintext
cpu_core=5
libraries="/path/to/lib1 /path/to/lib2"
```

The creator can replace the actual values of parameters with our directives like

```plaintext
cpu_core=<INPUT>
libraries=<LIST <BROWSER><BROWSER>>
```

An HTML form with one text input box followed by a web file browser followed by two coupled web file browsers is generated for any researcher who wants to launch the same application. The web file browsers have the same function as the universal web file browser provided by the Big Data Manipulation module.

3.3 BIC-LSU: Evaluation

In this section, we evaluate the performance of each component of BIC-LSU with experiment or simulation.

3.3.1 Cache Storage Server

We evaluate the performance of the all-SSD RAID 0 on the cache storage server. We evaluate both the buffered and unbuffered I/O modes. The workloads vary from 1 I/O operation to 64 parallel I/O operations. The throughput of the read operation always exceeds 50Gb/s. The throughput of the write operation is presented in Figure 3.7. The buffered I/O writes slower than the unbuffered I/O because the virtual memory (VM) module in the OS kernel cannot write data from main memory to the RAID 0 at a sufficient high bandwidth. The unbuffered I/O bypasses the VM module and writes at the full bandwidth of the RAID 0. To achieve the maximum write throughput, 42.4Gb/s, of the RAID 0, the transfer application must
FIGURE 3.7: Write performance of the RAID 0 on the cache storage server. 2 through 8 unbuffered parallel writes achieve the maximum throughput of 42.4Gb/s. Use the unbuffered I/O mode and the number of parallel write operations should be between two and eight. The BIC-LSU Command Service generates parameters to start transfer applications according to this observation.

3.3.2 Transfer Application

We evaluate transfer applications from three categories (1) the commonly deployed transfer applications, OpenSSH scp, HPN-SSH scp, and Globus Toolkit GridFTP, (2) the iRODS built-in transfer application, icp, and (3) the customized high-performance transfer application by BIC-LSU, BBCP and FDT. Receive(write) is the operation of interest. Data is transferred between two cache storage servers through the BIC-LSU SDN. Figure 3.8 shows the result. The number appending to each application is the number of parallel streams in use to achieve the maximum throughput. BBCP achieves almost the full network bandwidth, 38.6Gb/s, using four streams in the unbuffered I/O mode. FDT is the best cross-platform application which achieves 14.8Gb/s. GridFTP achieves only 6.5Gb/s. The iRODS icp achieves 18.8Gb/s but uses 64 streams which overwhelm the RAID 0 so that applications cannot profit from the unbuffered I/O mode. The BIC-LSU’s default
FIGURE 3.8: Performance of transfer applications. BBCP in unbuffered I/O mode is the overall best. FDT is the best cross-platform choice.

transfer applications, BBCP and FDT, strike a good balance between achieving high throughput and using few streams to avoid slowing down other transfer applications. BIC-LSU deploys BBCP on all supported UNIX-like partner systems and deploys FDT on the other supported partner systems.

3.3.3 Client Application

We evaluate the performance of the Java Web Start BIC-LSU WebIO client application for uploading data to and downloading data from the cache storage server on the regular campus network. On the campus WiFi network, BIC-LSU WebIO using SFTP increases throughput by 1.25x while BIC-LSU WebIO using FDT increases throughput by 1.4x compared to the commonly deployed OpenSSH scp. On the campus wired network, BIC-LSU WebIO using SFTP increases throughput by 1.33x while BIC-LSU WebIO using FDT increases throughput by 2x compared to OpenSSH scp. Figure 3.9 shows the result.
3.3.4 OpenFlow Controller

**Transer Application Task Level Application-aware Scheduling** We evaluate the performance of the BIC LSU SDN scheduling algorithm by isolating it from the SDN controller and supply various number of task requests. The algorithm is able to efficiently schedule 128 task requests in 0.1 second which is far shorter than the time for flow installation time on the physical OpenFlow switches. The scheduling algorithm is not the bottleneck of the SDN for sure. The algorithm also demonstrates a nearly linear scalability. Figure 3.10 shows the result.
**Bounded Shortest Job First Scheduling** Since BIC-LSU’s schedule for production is August 2016, a very limited number of partner systems connect to the BIC-LSU SDN at the time of the writing of this paper. There are not sufficient senders and receivers to emulate the traffic dynamics of the SDN in production. Thus, we create a flow-level simulation to evaluate the impact of the bounded shortest job first scheduling. Requests are generated from the statistics model. 128 requests are in a Poisson Process with arrival intensity 1.0. The data size in each request is chosen among the targeted big data size range of BIC-LSU. The simulation result shows that our bounded shortest job first scheduling reduces task completion time by 11.7%.

3.4 BIC-LSU: Related Work

High-performance transfer applications for big data movement such as GridFTP [45] and BBCP [56] typically use parallel TCP connections to accelerate. The transmission starts when the first TCP connection starts and finishes when the last finishes. Progresses of TCP connections should be coordinated to finish at close time instants and yield bandwidth to other applications. Existing transfer applications leave parallel TCP connections uncontrolled.

Hedera [73] balances load in OpenFlow-based DCN using Global First Fit and Simulated Annealing. The algorithm evenly distributes individual flows among equal cost links. Logical relationships between flows are ignored.

DANCES [74] develops an OpenFlow-based SDN and allows users to specify traffic from applications to be scheduled. DANCES adopts the 4-tuple for distinguishing TCP connections, *source IP Address, destination IP Address, source port,*
and destination port, to identify applications. It only schedules individual TCP connections.

Hesham et al. [75] uses header fields supported by OpenFlow to identify applications and enforce network policies. The OpenFlow network does not know the correlation between individual flows. The collective goal of the parallel flows is overlooked while flows being scheduled.

Pongsakorn et al. [76] allows users to select routing preferences for applications in OpenFlow-based SDN. Their system adopts the 4-tuple for distinguishing TCP connections to identify applications. It only schedules individual TCP connections.

3.5 BIC-LSU: Conclusion

BIC-LSU is a campus cyberinfrastructure for big data migration and analysis. Its features include 40Gb/s disk-to-disk transmission, short average transfer application task completion time, a universal light-weight control framework for coordinating heterogeneous partner systems, and a comprehensive social networking user interface. Its Clos SDN architecture is designed to be scalable and reliable. Its software architecture is designed to be extensible and have least dependencies. Other research institutes can easily customize BIC-LSU and augment it to their current IT infrastructures.

3.6 MinCOF Project: Minimal Coflow Routing and Scheduling in OpenFlow-based Cloud Storage Area Networks

Nowadays, researchers widely utilize the enormous computational power of computer clusters to analyze big data. Many cluster computing frameworks [37, 77] emerge to unify and simplify the development of the analytic cluster applications. These frameworks transform application logics into an iteration of alternating and mutual blocking computing and communication stages (e.g., MapReduce jobs). In
the communication stage, typically, multiple hosts exchange the result from the
previous computing stage with multiple other hosts using multiple communication
flows. The computing framework suspends the next computing stage until the all
communication flows finish.

Coflow scheduling [12, 5] is introduced to optimize the network usage and help
shorten the completion time of the communication stage. It bundles communica-
tion flows into coflows according to their logical semantics in the application
programs and sophisticatedly allocates network resources to coflows for shortening
the overall completion time of all associated communication flows. Shortening the
coflow completion time (CCT), from the beginning of the first associated flow to the
ending of the last associated flow, shortens the communication stage. If a cluster
application spends a substantial portion of time in communication stages [78] (e.g.,
intermediate data sizes of a MapReduce job is large), coflow scheduling shortens
the application completion time. Successive researches incorporate network routing
to coflow scheduling [79]. The outcome coflow scheduling and routing framework
has even better performance.

By investigating existing coflow scheduling/routing frameworks, we summarize
3 common critical correlated features as follows.

1. Coflow Scheduling: Determining the order of coflows and the rate of each
   flow in coflows to transmit according to the coflow information synchronized
   from all hosts.

2. Coflow Routing: Determining the path for each flow in coflows to transmit.
3. **Per-flow Rate-limiting**: Enforcing the rate allocation to each flow by the coflow scheduler to prevent individual flows from aggressively seizing network bandwidth.

Some existing frameworks provide the 3 features by customizing operating systems (OSs) on hosts to synchronize coflow information to the coflow scheduler and limit flows’ rates such as Varys [5] and Rapier [79]. This strategy may prevent the adoption of coflow scheduling/routing in a DCN if the hosts require another customized OS which results in incompatibility issues. Other existing frameworks rely on external commercial monitoring tools on switches to synchronize coflow information such as Tailor [80] which requires the proprietary sFlow-RT [81]. However, deploying commercial tools in a production environment is likely to cost additional budget. Tab. 3.2 summarizes the features of existing frameworks. Details are included in Sec. 3.10.

To avoid all the obstacles to deploying coflow scheduling/routing in production, we design, **MinCOF**, a coflow scheduling and routing framework which imposes minimal requirements on hosts and thoroughly takes advantage of the standardized mature commodity OpenFlow Switches. **MinCOF** migrates all unnecessary requirements away from the switch/host and efficiently provide them using the OpenFlow
1.3 Switch and our coflow scheduler (3.8). **MinCOF** is especially suitable for cloud SANs storing big data. For example, the scientific cloud object storage [82, 83] involves massive coflow communication scenarios (3.7), and its big data transmissions which utilize few concurrent long flows can be processed by current OpenFlow SDN implementations. Widely used transfer applications in cloud SANs only have to augment small segments of code to their original connection establishment and progress reporting functions to be integrated.

We measure the performance of **MinCOF** on our DCN testbed with physical software OpenFlow Switches and hosts. The results illustrate that **MinCOF** largely outperforms the existing framework without the routing feature (**Varys**) and further shortens the average CCT by up to 12.94% compared to the existing framework without rate limiting (**Tailor**). **MinCOF** also maintains the lead under various workload compositions and network topologies.

3.7 **MinCOF**: Motivation

Two observations motivate us to create a new coflow scheduling and routing framework for OpenFlow-based cloud SANs, (1) the cloud SAN has high similarity to the DCN for cluster computing in which existing coflow scheduling/routing frameworks operate, (2) the OpenFlow SDN is suitable for processing the traffic load intensity of the cloud SAN.

For the similarity of cloud SAN and DCN for cluster computing, both networks have similar underlying DCN topologies and similar communication patterns. The improvement brought by coflow scheduling/routing on cluster computing is likely to be reproducible in cloud SAN. Both networks are built on topologies with multiple alternative paths between a pair of hosts such as the fat-tree [21]. Both networks
carry communication patterns involving a group of hosts concurrently communicating with another group [12]. The communication stage in cluster computing and data exchange/replication in cloud SAN all prefer multiple communication flows in the same logical operation finishing at close time instants. An example in cloud SAN is that a file transmission completes fast only if all parallel transferring flows complete fast because the slowest flow determines the overall file completion time.

For processing the traffic load with the OpenFlow SDN, the cloud SAN typically serves limited number of concurrent flows and experiences less frequent flow arrival/completion compared to DCN for cluster computing. The transfer applications in cloud SANs usually establish few parallel flows for data exchange. The flow table capacity of a commodity OpenFlow Switch is able to contain the number of concurrent flows [84]. When the cloud SAN stores big data, total files is relatively few and individual files are typically large so that parallel flows carrying files are long-lived. There are few flow arrivals and completions. The undesirable overheads of processing those events in the OpenFlow SDN are infrequently encountered.

The 2 observations suggest the feasibility of porting coflow scheduling and routing to OpenFlow-based cloud SANs, thus we design MinCOF.

3.8 MinCOF: Framework Design

We design a coflow scheduling and routing framework, MinCOF, which is expected to achieve the following objectives in cloud SANs.

1. **Short Average CCT:** Our framework should minimize the average CCT of transfer applications to increase the throughput of the object storage. Shortening the CCT of the transfer application implies shortening the turnaround time of the storage I/O operation.
2. **Starvation Free Scheduling:** Our framework should prevent a coflow from being delayed for an uncertain amount of time.

3. **Work Conserving Scheduling:** Our framework should allocate any available network resource if the resource trigers a coflow to progress.

4. **Backward Compatibility:** Our framework should efficiently operate with flows without coflow affiliation.

5. **Proprietary System Avoidance:** Our framework should use mostly open source components to avoid licensing costs when deployed on production environments.

6. **Immediate Deployability:** Our framework should impose least change on existing software and be built on commodity hardware.

The designs which ensure these objectives are highlighted in the following sub-sections.

We first give a high-level description of the procedure for the framework components to cooperate and dive into details in each subsequent sub-sections.

### 3.8.1 Framework Workflow

Fig. 3.11 illustrates the targeted spine-leaf cloud SAN topology and all framework components of MinCOF.

The framework initialization steps are

1. Each link from a leaf switch to a spine is sliced into queues for prioritizing coflows (Fig. 3.11(d)).
FIGURE 3.11: (a) The overall architecture of MinCOF. Dashed links carry control messages. To save space, hosts connecting to leaf switches other than $L_n$ are omitted. (b) The functions on the leaf OpenFlow Switches. (c) The functions on the hosts. (d) The slices in links from leaf OpenFlow Switches to spines for prioritizing coflows.

2. Coflow-aware transfer application is deployed on each host (Fig. 3.11(c.c1) and (c.c2)). (3.8.3)

The initialization steps for each coflow are

3. Unique coflow IDs are given to start transfer applications on hosts. Transfer applications report per flow information in coflows to the coflow scheduler (Fig. 3.11(c.c1) and (a.c1)). (3.8.3)
4. The scheduler schedules the new coflows. (3.8.4)

The repetative steps during the live time of each coflow are

5. Transfer applications update remaining data size of each flow to the scheduler (Fig. 3.11(c.c2) and (a.c1)). (3.8.3)

6. The scheduler calculates a new coflow schedule at coflow arrival and finishing. The scheduler releases per flow route, per flow queue schedule, and per flow rate to the OpenFlow Controller (Fig. 3.11 (a.c2)) to achieve short average CCT, starvation free, and work conserving objectives. (3.8.4)

7. The controller enforces the new per flow properties on leaf OpenFlow Switches (Fig. 3.11 (a.c3) and (b)).

3.8.2 OpenFlow 1.3 Switch

MinCOF requires all leaf switches to be OpenFlow 1.3 Switches. Coflow schedules, coflow routing, and per-flow rate-limiting are enforced on OpenFlow Switches.

For enforcing coflow schedules, MinCOF takes advantage of slicing links from each leaf switch to spines for different classes of traffic. The slicing is achieved by dividing each physical egress port into 3 queues, Scheduled queue(50% port capacity), Starved queue(20%), and Best Effort queue(30%) shown in Fig. 3.11(d). When coflow scheduler pass a new coflow schedule, coflows with exclusive bandwidth allocations are placed in Scheduled queue with rate limiting configured, and the others are moved into the Starved queue. To ensure starvation free scheduling, Starved queue has identical priority to transfer data as the Scheduled queue. To ensure work conserving scheduling, any traffic exceeding the
capacity of the Scheduled queue or Starved queue are moved into the Best Ef
fort queue. Flows in the Starved queue and Best Effort queue freely compete for
bandwidth. If fair share among flows is preferred, active queuing disciplines such
as AFCD[85, 86] and FaLL[87] should be deployed. The best effort queue length is
set to 5% Bandwidth-delay Product (BDP) to keep queuing delay low. To provide
backward compatibility, flows which do not have coflow affiliations are placed
in the best effort queue.

For coflow routing, MinCOF adopts MPLS segment routing (SR) (Fig. 3.11(b.d1)).
Each scheduled flow is attached with an MPLS label assigned by the scheduler when
first reaching a leaf switch. Routing between leaf switches is based on MPLS labels.
The benefit of using MPLS SR is that any conventional flow matching using the
5-tuple, <Protocol, Src. IP Addr., Dst. IP Addr., Src. Port, Dst. Port>, is reduced
to a matching using only the MPLS label. Spine switches can be regular switches
which are able to forward packets according to MPLS labels.

For per-flow rate-limiting, MinCOF utilizes the OpenFlow Meter. The coflow
scheduler passes the corresponding rate limit of each flow in scheduled coflows
in every new coflow schedule. The rate limit is configured on the OpenFlow Meter
table to control the transmission speed of flows.

3.8.3 Coflow-aware Transfer Application

To avoid using proprietary systems, we create the coflow-aware transfer ap-
lication by making a regular transfer application synchronize its own coflow infor-
mation to our coflow scheduler. No proprietary flow monitoring tool is needed. We
develop a compact 1-way egress synchronization protocol from the transfer appli-
cation to the coflow scheduler. The benefit of our 1-way protocol is the simplicity.
The 2-way coflow synchronization in existing frameworks requires a server which accepts incoming traffic on each host. Much more security and system management policy configurations have to be changed. **To provide immediate deployability**, our protocol only requires a regular UDP socket.

Our protocol can easily be integrated into the typical workflow of popular transfer applications as follows.

The typical workflow of a transfer application is

1. Receiving command line arguments.

2. Creating parallel TCP flows and evenly distributing data to TCP flows.

3. Periodically collecting statistics from all flows and reporting aggregated transmission progress.

The workflow after integrating our protocol is

1. Receiving command line arguments.

2. Receiving coflow ID and coflow scheduler location.

3. Creating parallel TCP flows and evenly distributing data to TCP flows.

4. Initially synchronizing the flows in the coflow information to coflow scheduler using 1 egress UDP segment.

5. Periodically collecting statistics from all flows and reporting aggregated transmission progress.

6. Synchronizing flows in the coflow information to the coflow scheduler using 1 egress UDP segment.
An example integrated transfer application is the customized BBCP of the BIC-LSU big data storage area network[18].

3.8.4 Coflow Scheduler and OpenFlow Controller

The core control components of MinCOF are a logically centralized coflow scheduler and an OpenFlow Controller. The coflow scheduler sophisticatedly allocates network resources to ensure that design objectives are achieved. OpenFlow Controller enforces the allocations by coflow scheduler on corresponding OpenFlow switches.

Our coflow scheduler is equipped with the 3 critical desirable features, coflow scheduling, coflow routing, and per-flow rate-limiting, of coflow scheduling/routing frameworks so that it can produce efficient or even optimal network resource allocations. We use 3 comprehensive examples to illustrate the importance of the critical features in shortening average CCT or flow completion time (FCT) by applying the shortest job first heuristic. All paths in examples have the bandwidth \( \frac{1}{\text{unit time}} \) and all flows/coflows arrive at Time 0. **First,** we introduce the per-flow rate-limiting. In Fig. 3.12, 2 flows, \( F_1 \) (Size 1) and \( F_2 \) (Size 2) share 1 path. The optimal schedule is achieved in Fig 3.13. Per-flow rate-limiting can arbitrarily change the order of flow transmissions and enables the other 2 features. **Second,** we explain the coflow scheduling. In Fig. 3.14, 2 coflows, \( C_1 \) with Flow \( C_{1,1} \) (Size 1) and Flow \( C_{1,2} \) (Size 1) and \( C_2 \) with Flow \( C_{2,1} \) (Size 2) and Flow \( C_{2,2} \) (Size 1) share 2 independent paths. Coflow scheduling schedules all flows in one coflow as a logical unit. The optimal schedule is achieved in Fig 3.15. **Third,** we elaborate the coflow routing. In Fig. 3.16, 2 coflows, \( C_1 \) with Flow \( C_{1,1} \) (Size 3) and Flow \( C_{1,2} \) (Size 1) and \( C_2 \) with Flow \( C_{2,1} \) (Size 2) and Flow \( C_{2,2} \) (Size 3) share 2 alternative paths. Coflow routing finds
FIGURE 3.12: 2 flows must fairly share one path without per-flow rate-limiting. Avg. FCT = (2+3)/2 = 2.5.

FIGURE 3.13: 2 flows optimally share one path with per-flow rate-limiting. Flows can transfer in arbitrary order. Avg. FCT = (1+3)/2 = 2.

FIGURE 3.14: Random schedule of 2 coflows. Avg. CCT = (2+3)/2 = 2.5.

FIGURE 3.15: Optimal coflow schedule. Avg. CCT = (1+3)/2 = 2.

the best route all flows in one coflow as a logical unit. The optimal routing decision is achieved in Fig 3.17.

Our scheduler maintains a Data Structure Path to avoid using proprietary systems for collecting coflow information. Path tracks the bandwidth usage in the Scheduled queue in each leaf-to-leaf path. The per-flow rate-limiting ensures that the usage information in Path is precise. The total path tracked is the product of total leaves by total spines. In Fig. 3.11, the total path tracked is $s \cdot \ell$. As the blocking ratio (“total bandwidth to hosts” to “total bandwidth to spines” ratio on the leaf switch.) in the cloud SAN is typically high (10:1), the size of Path is limited and can be efficiently managed using hash table. Function PathRem($i,j,p,h$)
returns the remaining available bandwidth from Host $i$ to Host $j$ via Path $p$. Each leaf switch connects to $h$ hosts.

Our coflow scheduler is able to estimate the capacity gain of re-routing a TCP flow. We assume that every host uses the default Ethernet Maximum Transmission Unit (MTU) size 1500 bytes. Due to the stably short queue at each switch port (3.8.2), BDP can be estimated, so can the TCP’s congestion window size ($C_{wnd}$) using the theoretical model in [88], $C_{wnd} \approx \text{BDP}/\text{MTU}$. The duration of a re-routed flow is reasonably assumed to be the inter-coflow arrival time. Fig. 3.18 illustrates the worst impact of re-routing. The area below the curve is the effective capacity (BDP) of the flow. $C_{O}(C_{N})$ is the effective capacity on the old(new) after re-routing and is calculated in Eq. 3.2(Eq. 3.4) respectively. The overall capacity gain of re-routing a TCP flow to a faster path is calculated by $C_{N} - C_{O}$.

Our coflow scheduler executes Alg. 3.8.1 to serve coflows at each coflow arrival/completion by invoking serveCoflow. The new coflows first are routed through DCN for maintaining load balancing (Line 17-23). A new coflow schedule is generated for all coflows (Line 24). We define the bottleneck of a coflow to be the completion time of its slowest flow, which is calculated using Eq. 4.4. The coflow with the smallest bottleneck is re-routed if there are paths through which the coflow
can earlier complete (Line 2-5) measured by the estimation mechanism explained in the previous paragraph. **To shorten average CCT**, coflows receive bandwidth allocation in the shortest bottleneck first (SBF) order (Line 6-12). Coflows with all flows which receive sufficient exclusive bandwidth allocation to continue are categorized as Scheduled Coflows ($C_{sch}$). The others are categorized to as Starved Coflows ($C_{str}$). The Scheduled Coflows in the latest coflow schedule are placed in the Scheduled queue on their paths (Line 25-30) to receive a large portion of link bandwidth. **To ensure starvation free scheduling**, the Starved Coflows are moved to the Starved queue on their paths (Line 31-35) to share a small portion of link bandwidth (3.8.2).

The OpenFlow Controller configures OpenFlow Switches to enforce coflow scheduling, coflow routing, and per-flow rate-limiting using the OpenFlow protocol. **To provide immediate deployability**, MinCOF only includes hardware features which are provided by commodity OpenFlow Switches and scheduler functions which can be implemented on any up-to-date OpenFlow Controller frameworks.

\[ \Gamma = \max_i (\max_j (\max_p d_{ij} / \text{PathRem}(i, j, p, h))) \] (3.1)

where $d_{ij}$ is the summation of the remaining data size of the current coflow from Host $i$ to Host $j$.

\[ C_O = D \frac{W_O}{2} + \left( D \ \text{div} \ \frac{W_O}{2} \right) \frac{W_O^2}{4} + \frac{(D \ \text{mod} \ \frac{W_O}{2})^2}{2} \] (3.2)

\[ D_h = D - \frac{W_N}{2} \] (3.3)

\[ C_N = \frac{W_N^2}{4} + \frac{W_N}{2} D_h + \left( D_h \ \text{div} \ \frac{W_N}{2} \right) \frac{W_N^2}{4} + \frac{(D_h \ \text{mod} \ \frac{W_N}{2})^2}{2} \] (3.4)
Algorithm 3.8.1 Coflow Scheduling & Routing in OpenFlow-based spine-leaf DCN.

0: function SCHEDULE(Coflows $\gamma$, PathRem(.))
0: $C_{sch} =$ Sort all Coflows in $\gamma$ in SBF order
0: if Rebalancing shortens 1st coflow’s CCT in $C_{sch}$ using Eq. 4.4, Eq. 3.2, and Eq. 3.4 then
0: Rebalance flows in 1st coflow.
0: end if
0: for all coflow $C$ in $C_{sch}$ do
0: allocate BW.
0: Calculate $\Gamma$ using Eq. 4.4
0: for all flow in $C$ do
0: \( rate \leftarrow (\text{flow’s remaining size}) / \Gamma \)
0: Update PathRem(src host,dst,p,h) with \( rate \)
0: end for
0: end for
0: return $C_{sch}, C_{str}$
0: end function

0: procedure SERVECOFLOW(Coflows $\gamma$, PathRem(.))
0: if new coflow arrival then
0: for all flow $\in$ new coflow do
0: if flow traverses across leaf SWs then
0: Place flow on least used inter-leaf path.
0: end if
0: end for
0: end if
0: $C_{sch}, C_{str} =$ SCHEDULE($\gamma$, PathRem(.))
0: for all flow in $C_{sch}$ do
0: place scheduled flows on src leaf SWs.
0: Configure new segment routing path
0: Migrate flow to Scheduled queue at egress port
0: Configure flow’s OpenFlow Meter entry
0: end for
0: for all flow in $C_{str}$ do
0: place starved flows on src leaf SWs.
0: Migrate flow to Starved queue at egress port
0: Remove flow’s OpenFlow Meter entry
0: end for
0: end procedure
FIGURE 3.18: The worst impact of switching paths on the congestion window size of a TCP flow. The flow re-routes to a faster path at Time $W_O$. Transmission capacity loss because of TCP re-ordering (shaded area) and gain (slashed area) are marked.

where $\text{div}$ calculates the quotient of integer division and $\text{mod}$ calculates the remainder of integer division.

3.9 MinCOF: Evaluation

To evaluate the improvement of MinCOF, we compare the results of running MinCOF and previous works, Varys with ECMP routing and Tailor, on a DCN testbed with synthesized SAN workloads. Due to the complexity, we do not include the Rapier in our evaluation. Each result is the average of 5 runs and normalized to the result of Varys with ECMP routing.

3.9.1 Testbed Environment

All scheduling frameworks run on a physical DCN testbed with the spine-leaf fat-tree topology. There are 4 spine switches, 4 leaf switches, and 16 hosts on the testbed shown in Fig. 3.19. Commodity servers are used to emulate both switches and hosts. Each switch and host has 8 Intel Xeon 2.33GHz CPU cores, 8GB main memory. Each switch has 2 Intel I350-T4 network interface cards. All servers
run the Ubuntu Linux 16.04. For the simplicity of configuration, we deploy Open vSwitch (OVS) OpenFlow Switch [35] on every switch even though MinCOF does not require the spine switches to be OpenFlow Switches. The coflow scheduler is written in Python, and the OpenFlow Controller is built on the Ryu framework [89]. However, the OpenFlow Meter is not implemented on OVS. We create a primitive control framework for the OpenFlow Controller to manipulate the tc command on each switch to mimic the behavior of the OpenFlow Meter using the Linux kernel packet scheduler. All network interfaces are throttled to 100Mb/s to ensure that the processing power of the CPUs on the switch is competent for forwarding network traffic. The RTT at a host is around 350µs.

3.9.2 Workload

To the best of our knowledge, there are very few publicly available statistics of data size distribution in cloud storage. Since the design of MinCOF targets the big data cloud storage, we adopt the scientific big data size distribution in the Data Oasis storage cluster [90]. We scale down the file size by multiplying the testbed bandwidth to Data Oasis’ SAN bandwidth ratio, \( \frac{\text{Exp. Net BW}}{\text{Data Oasis Net BW}} = \frac{100\text{Mb/s}}{10\text{Gb/s}} = 0.01 \). The scaled file size distribution is summarized in Fig. 3.20. To increase scalability,
FIGURE 3.20: The downscaled file size distribution of Data Oasis.

TABLE 3.3: Statistics of coflow inter-arrival time in the Facebook traffic trace.

<table>
<thead>
<tr>
<th>Mean</th>
<th>Median</th>
<th>Mode</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.91s</td>
<td>3.06s</td>
<td>1.58s</td>
<td>9.76s</td>
</tr>
</tbody>
</table>

MinCOF schedules only at coflow arrival/finishing and only considers the coflows carrying the large files (>2.5MB) which already account for around 70% of total data size. We generate 64 coflows with the arrival times distribution from the Facebook trace in [5]. Each coflow involves 3 through 12 normally distributed host-to-host file transmissions. Each file transmission uses 4 through 16 flows.

3.9.3 Impact of Coflow Width

In this scenario, we test coflow scheduling/routing frameworks under various coflow width, number of flows in coflows. We intentionally generate coflows all with only 4, 8, 12, or 16 flows. The result is provided in Fig. 3.21. MinCOF and Tailor both more effectively shorten the average CCT compared to Varys. MinCOF further improves by at most 9.81% compared to Tailor in the case of 8 flows. The trend for saturation with larger number of flows is because of the overhead of our workaround implementation of the OpenFlow Meter. Each flow becomes shorter and faster finishes when the coflow width increases. The coflow scheduler is not able to instantaneously adjust the OpenFlow Meter entries to the most adequate values. We conjecture that the saturation can be alleviated if MinCOF runs on dedicated hardware OpenFlow Switches because the OpenFlow protocol message format and message passing mechanism is standardized and highly optimized.
FIGURE 3.21: Impact of Coflow Width. Shorter CCT is better.

3.9.4 Impact of Coflow Size

We vary coflow size, the aggregated size of all associated flows, in the experiment. We divide the file size range into 3 categories, <40MB(4x10MB), <160MB(4x40MB), and <200MB(4x50MB). The last range ends at 50MB because we simulate the scaled down file size upper limit in the OpenStack Swift cloud object storage, 5GB. Fig. 3.22 presents the result. MinCOF demonstrates more advantage with larger coflow size, up to 12.94% compared to Tailor, if the coflow can be as large as 200MB. The overhead problem mentioned in Sec. 3.9.3, again, results in the indistinction between MinCOF and Tailor in the case of small coflow.

3.9.5 Impact of Network Blocking Ratio

In this experiment, we observe the performance of coflow scheduling/routing frameworks in the spine-leaf fat-tree DCN with various blocking ratios. A higher blocking ratio reflects that communications between hosts connecting to different leaf switches are more difficult. The result is summarized in Fig. 3.23. With high block-
ing ratio, the severe congestion in the network lessens the improvement of routing so that 3 frameworks have similar performance. With low blocking ratio, MinCOF effectively routes flows among multiple available paths so that achieves the largest improvement of 11.33% compared to Tailor.

3.9.6 Impact of Background Traffic

We randomly select coflows to not synchronize with the scheduler as background traffic to examine the resilience and backward compatibility of the coflow scheduling/routing frameworks. Background traffic in practice is from applications which are not yet integrated with the coflow scheduling/routing framework. The percentage of background traffic gradually increases from 10% through 50%. The result is shown in Fig. 3.24. Tailor and MinCOF outperform Varys because of their routing feature. However, Tailor saturates with 40% background traffic. MinCOF profits from the sliced leaf-to-spine links (3.8.2) which confine the bandwidth consumed.
by background traffic and still improves under heavy background traffic. The maximum improvement is 15.21% with 50% background traffic.

3.10 MinCOF: Related Work

Varys is the coflow scheduling framework which totally operates on hosts and is not aware of the network condition. Varys requires a customized OS on each host for a 2-way interact to a logically centralized coflow scheduler to monitor network interface usage, synchronize coflows’ information, and enforce coflow scheduling decisions. The critical coflow scheduling features of Varys include coflow scheduling and per-flow rate-limiting.

Rapier inherits all features from Varys and incorporates coflow routing while generating coflow schedules. Rapier also needs a customized OS to enforce per-flow rate-limiting.

Tailor is a simple coflow scheduling and routing framework built upon OpenFlow-based DCN. Limited coflow scheduling and coflow routing are included. Tailor
adopts proprietary sFlow-RT to collect coflows’ information and network condition for trimming the completion time of the slowest flow. Per-flow rate-limiting is not considered. Tailor does not require any features provided by a customized OS. Applications can profit from Tailor coflow scheduling and routing with simple modification.

**MinCOF** synthesizes all desirable critical features of all previous works including coflow scheduling, simplified coflow routing, and per-flow rate-limiting. **MinCOF** also gets rid of undesirable requirements such as customized OS and proprietary system component. An OpenFlow-based DCN and extended transfer applications for the 1-way coflow information synchronization are the only requirements.

3.11 MinCOF: Conclusion and Future Works

Existing coflow scheduling/routing frameworks require customized OSs on hosts or proprietary network monitoring tools to fully function. Those requirements hinder the adoption of coflow scheduling which does effectively reduce the communication
and completion time of applications in DCNs. We design MinCOF, a coflow scheduling and routing framework which imposes only minimal modifications to applications and accommodates all critical features of coflow scheduling/routing targeting the OpenFlow-based cloud SANs. Being built upon open source software and standardized commodity OpenFlow Switches, MinCOF is easily deployable in production cloud SANs. Experiment results using software OpenFlow Switches confirm that MinCOF shortens the CCT by up to 12.94% compared to latest OpenFlow-based framework. MinCOF in the cloud SANs with dedicated hardware OpenFlow Switches is expected to perform even better. As this prototype implementation illustrates consistent improvements, we plan to port MinCOF to the BIC-LSU 10/40 Gb/s OpenFlow-based spine-leaf big data SAN.
CHAPTER 4
APPLICATION-AWARENESS IN DATA CENTER NETWORK USING FAST PACKET PROCESSING LIBRARY

The architectural design of separate control plane and data plane for the OpenFlow-based DCN restricts the efficiency of the DCN to instantly react to sophisticated traffic variations that cannot be handle at the switch. Any decision making experiences a delay of event reporting from a switch to the controller, the decision making at the controller, and the decision notification from the controller to the switch. Implementing a custom type of SDN switch which can efficiently locally make decision using the fast packet processing library is an alternative.

We design a coflow level application-aware SDDCNCF, Coflourish. Coflourish requires switches to swiftly coordinate to estimate the instant available bandwidth between any pair of hosts across the DCN for coflow scheduling. A trace-driven flow-level simulation and a software switch implementation is created for performance evaluation. OpenFlow SDN is not able to satisfy the short timing requirement of the coordination. We implement Coflourish on the Open vSwitch with Intel DPDK packet processing library.

The contents of this chapter excluding the experiment related sections are from our publication “Coflourish: An SDN-Assisted Coflow Scheduling Framework for Clouds” [91].

4.1 Coflourish Project: Introduction

The prosperous growth of big data related researches and applications result in the thriving of cluster computing frameworks [37], which greatly simplify the development of applications for big data analysis. An application running on one of these frameworks typically consists of a sequence of dependent steps such as the
Map, Shuffle, and Reduce phases in the MapReduce framework. There are data exchanges among machines between steps. Machines exchange the output data of the previous step to prepare the input data for the next step. The application can resume only when the data exchange completes. In data-intensive applications, data exchange time can contribute to as much as 70% of the application completion time [78]. Shortening the data exchange time reduces a large proportion of the application completion time.

A data exchange mostly involves multiple parallel communication flows between machines. The data exchange completes only when all associated flows finish. Strategies for shortening the data exchange time should consider all associated flows as a logic unit. Coflow [12] is proposed to abstract the collective communication characteristics such as requirement and behavior between two groups of machines. Coflow scheduling frameworks [5, 6] allow programmers to specify the communication characteristics of applications using the coflow abstraction and shorten the average coflow completion time (CCT) by coordinating coflows’ transmission order and transmission rates. Evaluations show encouraging decreasing of CCT. The existing coflow scheduling frameworks such as Varys [5] are designed on the assumption that the underlying network can be seen as an ideal non-blocking switch connecting all machines which are monitored and controlled by the frameworks’ cooperative daemon processes. Making scheduling decisions by considering only the available bandwidth of the network interface card (NIC) on each controlled machine is sufficiently satisfying.

However, the non-blocking switch assumption is likely to not hold when the existing coflow scheduling frameworks are deployed in a cloud environment. A cloud
is typically a cluster of machines connected via a DCN [21, 22] shared by numerous tenants such as the Amazon EC2 [92] and Google Cloud Platform [93]. Tenants are free to deploy customized software environments on their machines. Thus, the existing coflow scheduling frameworks cannot guarantee that their daemons control all machines. Not controlled machines may run various applications which generate complicated communication flows across the cloud network. We refer to this category of communication flows as the background traffic in the rest of this paper. The network bandwidth consumed by the background traffic is transparent to the existing coflow scheduling frameworks. Once the background traffic is huge such as high-definition video streaming [29], the existing coflow scheduling frameworks do not perform as well as expected.

In this project, we study the performance degradation of the existing coflow scheduling frameworks such as Varys in the cloud environment and improve the performance by exploiting feedbacks from the switches in the DCN.

We motivate ourselves by studying the performance loss resulted in by the background traffic. We implement a trace-driven simulation of the Varys. The simulation runs on the workload from the benchmark Facebook traffic probability distribution. We observe up to 82.1% decrease of CCT (4.2). We realize that accurate available network bandwidth information is crucial to resolve the issue.

We propose Coflourish, the first coflow scheduling framework which takes congestion feedbacks from SDN-enabled switches in the fat-tree-based DCN [21, 22] for more accurate estimation of available network bandwidth (4.3). Every switch keeps track of the congestion information of every port. A feedback mechanism periodically aggregates the congestion information from each fat-tree layer for each host.
machine. A daemon process on each machine participating in the framework converts its collected congestion information into available bandwidth from (to) itself to (from) each fat-tree layer and report the available bandwidths to a logically centralized scheduler. The scheduler can easily synthesize the reported bandwidths and use the smallest bottleneck bandwidth as the overall available machine-to-machine bandwidth. Coflourish is the first coflow scheduling framework which provides detailed algorithms of the SDN-assisted available bandwidth estimation.

To evaluate Coflourish, we create a trace-driven simulation and run it on a large variety of DCN workloads in the flow level. In the presence of background traffic, simulation results show that our framework estimates the available network bandwidth 4.1% more accurately than Varys under light background traffic and 78.7% more accurately under heavy background traffic. Simulation results also reveal that our framework shortens the average CCT up to 75.5% compared to Varys (4.4). We also implement Coflourish and evaluate it in experiments.

4.2 Motivation

To understand the impact of the network congestion to the existing coflow scheduling framework in clouds, we construct a trace-driven flow-level simulation of Varys and examine Varys’ performance under various network traffic patterns.

We assume the topology of the underlying DCN to be the de-facto spine-leaf fat-tree [1] shown in Figure 4.1. Detailed simulation configurations are provided in 4.4.2. Two types of traffic loads are generated, regular size coflow traffic load (Regular) and large size coflow traffic load (Large). The Regular load represents the load which consists of traffic from regular cluster applications such as the MapReduce. The Large load represents the load which consists of 50% traffic from
FIGURE 4.1: A spine-leaf fat-tree DCN with 2 spine switches and 3 leaf switches. Each switch in the leaf layer connects to the same number of host machines.

applications that consistently produce large traffic such as HD video streaming and 50% traffic from the regular applications. We convert a certain weight (or percentage) from generated traffic load into the background traffic by removing their coflow related attributes to impose unknown congestions in network. The weight of background traffic varies from 10% through 90% with a 10% interval. We compare the performance of Varys with an ideal scheduling framework which runs identical scheduling algorithm as Varys’ but knows the precise available bandwidth of each network link when making new coflow schedules and immediately reschedules when there is any change of available bandwidth at any network link.

The average CCTs normalized to the Varys’ using Equation 4.1 are shown in Figure 4.2.

The ideal scheduling framework achieves 4.1%(82.1%) decrease in average CCT with 10%(90%) background traffic under the regular size coflow traffic load and achieves 7.3%(80.2%) decrease in average CCT with 10%(90%) background traffic
FIGURE 4.2: CCT of the ideal scheduling framework normalized to Varys’.

under the large size coflow traffic load. By comparing the results of the Varys and
the ideal scheduling framework, we conclude that accurate network congestion
information is critical to an efficient coflow scheduling.

\[
\text{Normalized CCT} = \frac{\text{New Fwk.'s CCT}}{\text{Varys' CCT}}
\] (4.1)

4.3 Framework Design

We propose a new coflow scheduling framework, **Coflourish** which inherits the
preferable features of the existing framework, Varys, and alleviates the perfor-
mance degradation problem resulted in by the inaccuracy of the available network
bandwidth estimation in Varys. The design goals of **Coflourish** are as follows.

- **Switch-assisted Bandwidth Estimation:** Network switches in the DCN
  should report the instant available network bandwidth information to the
  coflow scheduler for more accurate available bandwidth estimation. Compli-
cated computations should be avoided on switches.
• **Average CCT Minimization:** The average CCT should be comparable to existing coflow scheduling frameworks.

• **Starvation Prevention:** Coflows should not be suspended for an unpredictable amount of time because of the lack of necessary resources to continue.

• **Work Conservation:** Any resource which any coflow needs to make progress should be allocated.

We first introduce the overall workflow of all system components and then individually elaborate the detailed design for each.

4.3.1 Framework Workflow

Coflourish requires several components in the framework, network switches, host machines, and a logically centralized coflow scheduler, to tightly coordinate to achieve its design goals. Major steps are as follows.

1. Each switch in the spine layer periodically sends per-port congestion information to each switch in the leaf layer which itself connects to. (4.3.2)

2. Each switch in the leaf layer, upon receiving the congestion information from the spine layer, sends its own per-port congestion information and the aggregated information received from the spine layer to the host machine which it connects to in the host layer. (4.3.2)

3. Each host machine in the host layer, upon receiving the congestion information from switches, calculates the available network bandwidth from itself to each switch layer using the received information or estimates any missing
The host machine sends the available bandwidth information to the coflow scheduler. (4.3.3)

4. The coflow scheduler estimates available bandwidth for each source-destination machine pair and make scheduling decisions to achieve average CCT minimization, starvation prevention, and work conservation. (4.3.4)

4.3.2 SDN-enabled Switch

To accurately estimate the congestion in a DCN, we design custom configuration for network links, and data structures and algorithms for switches. Table 4.1 provides the meaning of symbols used in algorithms in the rest of this paper. Figure 4.3 illustrates the DCN on which all algorithms operate. To save space, we omit the host machines which connect to the leaf switches other than $L_n$. We assume all network links have identical capacities.

For each network link, the link is sliced into two logical channels. One with higher priority exclusively transfers the congestion feedback messages from switches to ensure that critical network congestion information delivery is not delayed by other types of traffic. The other one carries the actual data packets. The slicing can easily be implemented using existing technology such as the Priority Code Point (PCP) in the Ethernet.

For each switch, the switch maintains a congestion table and algorithms for updating the table and feeding back congestion information from every switch layer to the host layer. The spine switch initiates a feedback every $T_{FB}$ interval.

The congestion table tracks the congestion of each switch port with only one register and is updated using the EstCong (in Algorithm 4.3.1). EstCong in-

79
TABLE 4.1: Symbols used in algorithms

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>Network link capacity.</td>
</tr>
<tr>
<td>$CT$</td>
<td>Congestion table on current switch.</td>
</tr>
<tr>
<td>$CT_p$</td>
<td>Congestion value in the $CT$ for Port $p$.</td>
</tr>
<tr>
<td>$Pt(X,Y)$</td>
<td>Port on Switch $X$ connecting to Switch $Y$.</td>
</tr>
<tr>
<td>$AB(X)$</td>
<td>Available bandwidth of Port $X$.</td>
</tr>
<tr>
<td>$P_{out}^i$</td>
<td>Egress port on Machine $i$ ($H_i$).</td>
</tr>
<tr>
<td>$P_{in}^i$</td>
<td>Ingress port on $H_i$.</td>
</tr>
<tr>
<td>$S$</td>
<td>Set of all switches in the spine layer.</td>
</tr>
<tr>
<td>$s$</td>
<td>Total elements in $S$.</td>
</tr>
<tr>
<td>$L$</td>
<td>Set of all switches in the leaf layer.</td>
</tr>
<tr>
<td>$\ell$</td>
<td>Total elements in $L$.</td>
</tr>
<tr>
<td>$H$</td>
<td>Set of all host machines in the host layer.</td>
</tr>
<tr>
<td>$h$</td>
<td>Total machines a leaf switch connects to.</td>
</tr>
<tr>
<td>$X_{y..z}$</td>
<td>Element with Index $y$ through $z$ in set $X$.</td>
</tr>
<tr>
<td>$UL_i$</td>
<td>Avl. BW to leaf layer switch from $H_i$.</td>
</tr>
<tr>
<td>$DL_i$</td>
<td>Avl. BW from leaf layer switch to $H_i$.</td>
</tr>
<tr>
<td>$US_i$</td>
<td>Avl. BW to spine layer switch from $H_i$.</td>
</tr>
<tr>
<td>$DS_i$</td>
<td>Avl. BW from spine layer switch to $H_i$.</td>
</tr>
<tr>
<td>$C_{sel}$</td>
<td>Set of coflows with BW allocation.</td>
</tr>
<tr>
<td>$C_{str}$</td>
<td>Set of coflows without BW allocation.</td>
</tr>
</tbody>
</table>
crements the congestion value by the size of each packet which passes through (Line 2-4) and decrements by multiplying $(1 - \alpha)$ in the range of $(0, 1)$ every $T_{dre}$ interval (Line 5-9). This algorithm is a simplified Discontinuing Rate Estimator (DRE) [1] without computing the congestion metric. The adoption of DRE reveals a potential of Coflourish for integration with lower layer traffic engineering mechanisms in DCN.

The $T_{FB}$, $\alpha$, $\tau$, and $T_{dre}$ are dynamically tunable by the SDN controller.

**Algorithm 4.3.1 Congestion Estimation**

```
0: procedure EstCong(Event E, CT)
0: if E is packet arrival then
0:   $p \leftarrow$ event source switch port
0:   $CT_p \leftarrow CT_p +$ packet size
0: else if E is decrease timeout then
0:   for all $CT_x \in CT$ do
0:     $CT_x \leftarrow (1 - \alpha)CT_x$
0:   end for
0:   Schedule a decrease timeout in $T_{dre}$ interval
0: end if
0: end procedure
```

Each switch layer periodically feeds back its congestion information and the any aggregated congestion information from the next layer which is farer from the host layer to the next layer which is closer to the host layer every $T_{FB}$ interval. In Figure 4.3, each switch in the spine layer feeds back congestion information to the leaf layer, marked as Stage (1), and each switch in the leaf layer feeds back congestion information to the host layer, marked as Stage (2). During Stage (1), each $S_x \in S$ runs $FbSpine$ (in Algorithm 4.3.2). The congestion value of each egress port which connects to Leaf Switch $L_x$ (Line 3-4) is sent to $L_x$ as the from-spine-to-leaf congestion. During Stage (2), each $L_x \in L$ runs $AggFbLeaf$ (in Algorithm 4.3.2).
Each leaf switch sends three pieces of congestion information to Machine $H_x$. First, the congestion value of the egress port which connects the current leaf switch to $H_x$ (Line 15-17) as the from-leaf-to-host congestion. Second, the summation of congestion values of the egress ports which connect the current leaf switch to all spine switches (Line 10-12) as the from-leaf-to-spine congestion. Third, the summation of the from-spine congestion values from all spine switches (Line 9).

We only consider the congestion at egress ports because congestion happens only at egress ports. The ingress port is always not congested because the ingress traffic must be the egress traffic which is shaped by the congestion at some egress port. As revealed in the on-switch algorithms, no complex computation is involved. Major complexity is on the host machine and the scheduler. Coflourish imposes little overhead to switches.
Algorithm 4.3.2 Congestion Feedback & Aggregation

0: procedure FbSpine(CT, L)
  0: for all $L_x \in L$ do
  0: $pe \leftarrow \text{Pt}(\text{current switch}, L_x)$
  0: $fS \leftarrow \text{CT}_{pe} \{\text{S to L cong.}\}$
  0: Send $fS$ to $L_x$
  0: end for
  0: end procedure

0: procedure AggFbLeaf(Feedbacks FB, CT, S, H)
  0: $fS \leftarrow \sum (fS \in FB) \{\text{Agg. S to L cong.}\}$
  0: for all $S_x \in S$ do
  0: $pe \leftarrow \text{Pt}(\text{current switch}, S_x)$
  0: $tS \leftarrow \sum \text{CT}_{pe} \{\text{Agg. L to S cong.}\}$
  0: end for
  0: $n \leftarrow \text{current leaf switch ID}$
  0: for all $H_x \in H_{nh..nh+h-1}$ do
  0: $pe \leftarrow \text{P}(\text{current switch}, H_x)$
  0: $fL \leftarrow \text{CT}_{pe} \{\text{Agg. L to H cong.}\}$
  0: Send $fS, tS, fL$ to $H_x$
  0: end for
  0: end procedure

=0
To reduce the traffic load introduced by the feedback mechanism, each switch send feedback messages using simple transmission protocols such as Point-to-Point Protocol (PPP) and High-level Data Link Control (HDLC).

An SDN-enabled switch is easily be customizable to accommodate this congestion estimation mechanism. In the cloud environment, the SDN-enabled switch can be either physical or software switch on the host machine to connect virtual machines. For example, a physical switch which support the P4 programming language [31] or the Open vSwitch software switch which is already widely deployed in production cloud management systems [39].

4.3.3 Host Machine

The host machine in Coflourish does two tasks, finding the smallest bottleneck available bandwidth from(to) itself to(from) each switch layer for later bandwidth estimation at the scheduler, and pacing the transmission of flows in coflows using the transmission rate given by the scheduler. In this work, we only consider physical machines and switches in one DCN for easy explanations, but algorithms can be further extended to virtualized environment.

For finding the bottleneck available bandwidth, the host machine runs AggrptHost (in Algorithm 4.3.3). According to [1], the congestion value ($C_{ng}$) is proportional to the bandwidth of traffic. The available bandwidth ($A$) of a link can be derived using Equation 4.2.

$$C_{ng} = \lambda(C - A)\tau \Rightarrow \lambda\tau = \frac{C_{ng}}{C - A} \Rightarrow A = C - \frac{C_{ng}}{\beta}$$  \hspace{1cm} (4.2)

where $C$ is the capacity of the link, $\lambda$ is a scaling factor, $\tau$ is a time constant, and $\beta = \lambda\tau$ is a scaling factor. The from-leaf-to-host congestion value is fed back from
leaf layer to the host machine ($fS$ in Line 5). The host machine also knows its own exact available bandwidth from the leaf switch to itself (Line 4). Thus, the $\beta$ in Equation 4.2 is calculated (Line 6). The leaf-to-spine available bandwidth is calculated (Line 7). The $tS$ is an aggregated congestion value of all network links from a leaf switch to all $s$ spine switches so that the link capacity used in Equation 4.2 should also be aggregated as $sC$. The bottleneck available bandwidth from the current machine to spine layer is calculated (Line 8). The bottleneck available bandwidth from spine layer to current machine can be calculated through a similar process (Line 9-10). Finally, The available bottleneck bandwidths from current machine to each switch layer are reported to the Coflourish scheduler using a reliable transmission protocol such as TCP.

**Algorithm 4.3.3 Congestion Aggregation & Report**

0: **procedure** AggRptHost(Feedback $FB$)
0: $i \leftarrow$ current machine ID
0: $UL_i \leftarrow AB(P_{out}^i) \{BW to L.\}$
0: $DL_i \leftarrow AB(P_{in}^i) \{BW from L.\}$
0: $fS, tS, fL \leftarrow fS, tS, fL \in FB$
0: $\beta \leftarrow \frac{fL}{(C - DL_i)}$
0: $LtoSBW \leftarrow sC - \frac{tS}{\beta}$
0: $US_i \leftarrow \min(LtoSBW, UL_i) \{BW to S.\}$
0: $StoLBW \leftarrow sC - \frac{fS}{\beta}$
0: $DS_i \leftarrow \min(StoLBW, DL_i) \{BW from S.\}$
0: Send $US_i, DS_i, UL_i, DL_i$ to scheduler
0: **end procedure**

=0

85
For pacing the transmission of flows in coflows, the cooperative daemon process of Coflourish receives transmission rate of each egress communication flow from the scheduler and throttles each flow using the mechanism similar to Varys’.

The report interval and the number of aggregated links, $s$, can be dynamically changed by the SDN controller.

4.3.4 Coflow Scheduler

The scheduler of Coflourish decides the order in which coflows receive network bandwidth allocations and the amount of allocated bandwidth at which each flow in each coflow sends data to achieve all design goals. We enhance the state-of-the-art existing algorithm in Varys since many of its design goals are identical to ours. In this section, we briefly cover the complete logics of the scheduler and put emphasis on our enhancements. The logic which achieves each design goal is highlighted. The scheduler is also the SDN controller. We explain in details at the end of this section.

To provide switch-assisted bandwidth estimation, we add a data structure, PathRem, shown in Table 4.2 to help the scheduler trace the bi-directional available bottleneck bandwidth from(to) each host machine to(from) each switch layer. The PathRem synthesizes all the available bottleneck bandwidth information sent from the daemon on each host machine. PathRem$(i,j,h)$ represents the smallest available bottleneck bandwidth from the source host Machine $i$ to the destination host Machine $j$. Equation 4.3 illustrates the algorithm used by the PathRem. PathRem (1) calculates the rendezvous switch layer of the traffic from the source machine (up direction) and the traffic to the destination machine (down direction); (2) retrieves the estimated available upward and downward bandwidths;
TABLE 4.2: The PathRem which stores bi-directional available bottleneck bandwidths at the scheduler.

<table>
<thead>
<tr>
<th>Direction</th>
<th>Layer</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up</td>
<td>Leaf</td>
<td>UL_1, UL_2, ..., UL_{\ell h-1}</td>
</tr>
<tr>
<td></td>
<td>Spine</td>
<td>US_1, US_2, ..., US_{\ell h-1}</td>
</tr>
<tr>
<td>Down</td>
<td>Leaf</td>
<td>DL_1, DL_2, ..., DL_{\ell h-1}</td>
</tr>
<tr>
<td></td>
<td>Spine</td>
<td>DS_1, DS_2, ..., DS_{\ell h-1}</td>
</tr>
</tbody>
</table>

(3) calculates the overall available bottleneck bandwidth along the path across the DCN.

\[
PathRem(i, j, h) = \begin{cases} 
\min(UL_i, DL_j) & \text{if } (i \div h) = (j \div h), \\
\min(US_i, DS_j) & \text{otherwise.}
\end{cases}
\]

where \texttt{div} is the integer division operator. \quad (4.3)

Current colfows are scheduled by Algorithm 4.3.4, Varys' algorithm with our enhancements. Only colfows with size greater than 25MB are scheduled for scalability concerns as suggested by Varys. Whenever a colflow arrives/finishes or the variation of the available bandwidth in the DCN exceeds a certain threshold, \texttt{serveCoflow} is invoked and generates a new schedule. \texttt{serveCoflow} contains two stages, CCT minimization and starvation prevention. In the CCT minimization stage, the design goals of average CCT minimization and work conservation are achieved.

To minimize the average CCT, colfows are selected to progress (Line 22) with the Smallest Effective Bottleneck First (SEBF) heuristic (Line 3). The SEBF
in Varys only uses the estimated available bandwidth of the NICs as the end-to-end available bandwidth across DCN to calculate the Effective Bottleneck (EB) for each coflow. This simple method incurs huge error at the presence of the background traffic.

We improve the accuracy of the end-to-end available bandwidth estimation by introducing Equation 4.3. In Case 1, both ends attach to the same leaf switch so that the end-to-end available bandwidth is the minimum of the available bandwidth to the leaf layer from Machine $i$ ($UL_i$) and the available bandwidth from the leaf layer to Machine $j$ ($DL_j$). In Case 2, two ends attach to different leaf switch connected via the spine layer so that the end-to-end available bandwidth is the minimum of the available bandwidth to the spine layer from Machine $i$ ($US_i$) and the available bandwidth to from the spine layer to Machine $j$ ($DS_j$). We develop Equation 4.4 to determine the longest CCT ($\Gamma$) as the bottleneck of a coflow by taking the accurate end-to-end available bandwidth into consideration.

We schedule coflows using the shortest bottleneck first heuristic (SBF).

$$\Gamma = \max_i \left( \max_j \frac{d_{ij}}{PathRem(i, j, h)} \right)$$ (4.4)

where $d_{XY}$ is the summation of the remaining data size of the current coflow from Machine $X$ to Machine $Y$.

To provide work conservation, we distribute the unallocated bandwidth to the coflows which are able to progress in the latest schedule (Line 14). In the starvation prevention stage, the design goal of starvation prevention is achieved. All coflows which cannot acquire any network bandwidth to progress in the former stage receive bandwidth allocations (Line 24-31).
To prevent starvation, the CCT minimization stage and starvation prevention stage execute in turns. The former executes for $T_{sel}$ interval, and the latter executes for $T_{str}$ interval. This strategy guarantees that each coflow progresses for at least $T_{str}$ interval every $T_{sel} + T_{str}$ interval.

To effectively react to available bandwidth change due to the variation of the background traffic, we add a branch in SERVECOFLOW (Line 32-35). If the variation of the available bandwidth exceeds a tunable threshold, $thld_{BW}$, the variation is considered significant, and the current schedule is considered sub-optimal and a new schedule is generated. $thld_{BW}$ determines the sensitivity of Coflourish to available bandwidth change in the DCN.

Being the SDN controller as well, the scheduler is able to dynamically manipulate the behaviors of the congestion feedback mechanism on switches and the available bandwidth calculations on hosts as stated in previous sections. The Coflourish can dynamically adjust its performance as needed. Section 4.4.5 illustrates the impact of some adjustments.

4.4 Simulation Evaluation
4.4.1 Simulation Configuration

We extend the trace-driven flow-level simulation created in 4.2 to evaluate the performance of Coflourish. Each result is an average of 5 runs.

4.4.2 Simulation Environment

Our simulation is similar to that in [5, 79]. Coflow arrival events are inserted according to the workload trace and sorted by time in advance. So are the events of periodical congestion feedback and host report. When processing the current event, the simulation generates future events, updates state variables, or re-calculates flow remaining size according to Coflourish’s algorithms until no event exists.
Algorithm 4.3.4 Scheduling with Switch-assisted BW Estimation

0: **procedure** shareBW(Coflows $\gamma$, PathRem(.))
0:    for all coflow $C$ in $\gamma$ do
0:        Calculate $\Gamma$ using Eq 4.4
0:            for all flow in $C$ do
0:                rate $\leftarrow$ (flow’s remaining size) / $\Gamma$
0:                Update PathRem(src host,dst,h) with rate
0:            end for
0:        end for
0:    end procedure

0: **function** select(Coflows $\gamma$, PathRem(.))
0:    $C_{sel}$ = Sort all Coflows in $\gamma$ in SBF order
0:    shareBW($C_{sel}$, PathRem(.))
0:    Assign remaining BW to coflows in $C_{sel}$
0:    $C_{str}$ = starved coflows in $C_{sel}$
0:    return $C_{sel}$, $C_{str}$
0: end function

0: **procedure** serveCoflow(Coflows $\gamma$, PathRem(.))
0:    if is CCT min. stage then
0:        {CCT min. stage}
0:        Stop coflows in $C_{str}$
0:        $C_{sel}$, $C_{str}$ = select($\gamma$, PathRem(.))
0:        Switch to feeding starved stage in $T_{sel}$ interval
0:    else if is starvation prev. stage then
0:        {starvation prev. stage}
0:        Stop coflows in $C_{sel}$
0:        for all coflows in $C_{str}$ do
0:            Add all flows to one coflow
0:            shareBW(one_coflow, PathRem(.))
0:            Switch to min CCT stage in $T_{str}$ interval
0:        end for
0:    else if net avl. BW change $> thld_{BW}$ then
0:        {Significant avl. BW change}
0:        $C_{sel}$, $C_{str}$ = select($\gamma$, PathRem(.))
0:        go back to interrupted point of execution
0:    end if
0: end procedure
The simulation workload consists of two types of coflows: regular coflows and large coflows. A regular coflow represents the traffic load which follows the benchmark Facebook traffic probability distribution used in Varys [5](Figure 4). A large coflow is a coflow with its size uniformly distributed between 25MB and 1GB. In both Varys and Coflourish, only large coflows will be scheduled by the coflow scheduling framework to reduce the scheduling overhead. We vary the percentage of large coflows in the overall traffic load to evaluate the impact of large coflows on the overall coflow performance (e.g., CCT). Concretely speaking, we test three representative percentages of large coflows: 25%, 50% and 75% in our simulation. Once the overall traffic load is generated, we convert a certain weight (or percentage) from the overall load into the background traffic by removing their coflow related attributes. The purpose is to impose different levels of unknown congestions in network. The weight of background traffic varies from 10% through 90% with a 10% interval. We generate 3000 host machines connected by a spine-leaf fat-tree DCN. Each leaf switch connects to 50 host machines. The oversubscription is 10:1. The total number of coflows in our traffic load is 1000. The coflow arrival is a Poisson process. Other default framework related parameter configuration is presented in Table 4.3.

4.4.3 Improvement of Coflourish

We evaluate Coflourish’s improvement with two criteria, available bandwidth estimation accuracy and average coflow completion time.

First, we compare the available bandwidth estimation accuracies. We use regular traffic load and vary the weight of background traffic. The metric, Normalized
TABLE 4.3: Default Framework Parameter Configuration for Simulation.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{FB}$</td>
<td>200 milliseconds.</td>
</tr>
<tr>
<td>$T_{sel}$</td>
<td>2 seconds.</td>
</tr>
<tr>
<td>$T_{str}$</td>
<td>200 milliseconds.</td>
</tr>
<tr>
<td>$thld_{BW}$</td>
<td>1/8 spine layer network link capacity.</td>
</tr>
<tr>
<td>$\tau$</td>
<td>500 microseconds.</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>0.5.</td>
</tr>
<tr>
<td>$T_{dre}$</td>
<td>250 microseconds.</td>
</tr>
</tbody>
</table>

Inaccuracy (lower is better), is defined in Equation 4.5.

\[
\text{Normalized Inaccuracy} = \frac{|\text{New Fwk.'s Est. Err.}|}{|\text{Varys' Est. Err.}|} \tag{4.5}
\]

where the Err. is the difference between the estimated available bandwidth by a framework and the true available bandwidth in the network. The result is shown in Figure 4.4. The Coflourish's estimation is 4.1% (10% background) through 78.7% (90% background) more accurate than the Varys'. When the weight of background traffic is small, the interference from the background traffic to the Varys is small so that the Varys can estimate available bandwidth with slight error using its simple NIC-based estimation. Varys' estimation is slightly worse than the estimation of Coflourish based on accurate available bandwidth feedback information. When the weight of background traffic is large, the interference from the background traffic to the Varys is severe so that the simple bandwidth estimation of Varys gives largely biased result. Varys’ estimation is much more erroneous than the estimation of Coflourish base on accurate available bandwidth feedback information.

Second, we compare the average CCT. We use regular traffic load and vary the weight of background traffic. The result is shown in Figure 4.5. The Coflourish
shortens the average CCT by 4.3% (10% background) through 75.5% (90% background) compared to Varys’. As explained in the previous analysis, Varys estimates bandwidth with slight error when encountering small amount of background traffic, and Varys estimates bandwidth with huge error when encountering large amount of background traffic. Thus, Varys generates a comparable coflow schedule to Coflourish’s when background traffic is light, and Varys generates a much worse schedule than the Coflourish’s when background traffic is heavy.

4.4.4 Impact of Coflow Size

We generate 3 traffic loads with 25%, 50%, and 75% large coflows. The weight of background traffic varies from 10% through 90% with a 10% interval. The result is shown in Figure 4.6.

With less than 30% of background traffic in network, background traffic impact Varys’ bandwidth estimation little. Varys still generates comparable schedule to Coflourish.
With 30% through 80% of background traffic in network, background traffic heavily interferes the bandwidth estimation of Varys. Average CCT increases much. In contrast, Coflourish still generates good schedule with available bandwidth feedback information. The more the large background traffic, the larger the average CCTs differ between the Varys’ schedule and Coflourish’s schedule.

With more than 80% of background traffic in network, the background traffic still heavily impacts Varys bandwidth estimation, but the network tends to be saturated. The gap of average CCT resulted in by different quality of schedule closes. The average CCT also stops decreasing.

4.4.5 Impact of Feedback Interval and Bandwidth Variation Threshold

We probe the impact of two critical parameters in Coflourish, the feedback interval ($T_{FB}$) and the bandwidth variation threshold ($thld_{BW}$).

We set the $T_{FB}$ to 100, 200, and 400ms. We vary the weight of background traffic. The result is shown in Figure 4.7. A smaller $T_{FB}$ results in more frequent update of
available bandwidth information. The up-to-date available bandwidth information results in a schedule which is closer to the optimal. A larger $T_{FB}$ results in the opposite.

We substitute $1/4$, $1/8$, and $1/16$ the spine layer network link capacity into $thld_{BW}$. We vary the weight of background traffic. The result is shown in Figure 4.8. A smaller $thld_{BW}$ results in more frequent rescheduling with the lastest available bandwidth. Scheduling with up-to-date available bandwidth results in a schedule which is closer to the optimal. A larger $thld_{BW}$ results in the opposite.

4.5 Extended Framework for Virtualized DCN

A considerable portion of DCNs is built to accommodate the communications in the cloud environment. A cloud typically launches multiple virtual machines (VMs) on a physical host to accommodate more concurrent tenants to maximize hardware utilization. To support the DCN for clouds, we extend the Coflourish framework to the virtualized DCN. Figure 4.9 reveals the targeted DCN topology. On the basis
FIGURE 4.7: The impact of feedback intervals on the CCT of Coflourish normalized to Varys’. Lower CCT is better.

FIGURE 4.8: The impact of available bandwidth variation thresholds on the CCT of Coflourish normalized to Varys’. Lower CCT is better.
of the DCN consisting of physical hardware in Figure 4.3, each host runs multiple VMs, connects VMs with 1 virtual switch, and connects the virtual switch to the leaf switch.

The Coflourish’s switch is adapted to the virtualized network topology. The original AggFbLeaf in Algorithm 4.3.2 is extended to AggFbLeaf in Algorithm 4.5.1 for the congestion feedback and aggregation between the leaf switch and VS on host. AggFbVS in Algorithm 4.5.1 is added on VS for the congestion feedback and aggregation between the VS and VMs on host. AggRptVM in Algorithm 4.5.2 replaces AggRptHost in Algorithm 4.3.3 to calculate the available bandwidths from VM to each switch layer and report them to the coflow scheduler.

The data structures and corresponding algorithms in the coflow scheduler are accordingly extended.

4.6 Data Center Network Testbed Construction

In order to do research on SDDCN, we construct a 1Gb/s fat-tree DCN testbed, CRON-DC. Both switches and hosts in the fat-tree are servers with multiple Intel DPDK-capable NICs in the testbed. We are able to implement any proposed algorithm on the servers used as switches and evaluate the performance. We name a server used as a switch a switch server. We name a server used as a host a host server. Our FT(4,2) 1Gb/s DCN testbed is shown in Figure 4.10. We also construct a 10Gb/s fat-tree DCN testbed using high-performance servers. The topology, a variation of fat-tree, is shown in Figure 4.11.

We develop a management system, ClusterManager [94], for CRON-DC. ClusterManager is able to backup and recover the disk content of any server in the testbed using the Clonezilla [95]. While conducting an experiment, we can fine
Algorithm 4.5.1 Extended Congestion Feedback & Aggregation

0: procedure AggFbLeaf(Feedbacks $FB, CT, S, VS$)
0: $fS \leftarrow \sum (fS \in FB) \{ \text{Agg. } S \text{ to } L \text{ cong.} \}$
0: for all $S_x \in S$ do
0: \hspace{1em} $pe \leftarrow Pt(\text{current switch}, S_x)$
0: \hspace{1em} $tS \leftarrow \sum CT_{pe} \{ \text{Agg. } L \text{ to } S \text{ cong.} \}$
0: end for
0: $n \leftarrow \text{current leaf switch ID}$
0: for all $VS_x \in VS_{n\ldots nh+n+1}$ do
0: \hspace{1em} $pe \leftarrow P(\text{current switch}, VS_x)$
0: \hspace{1em} $fL \leftarrow CT_{pe} \{ \text{Agg. } L \text{ to } VS \text{ cong.} \}$
0: \hspace{1em} Send $fS, tS, fL$ to $VS_x$
0: end for
0: end procedure

0: procedure AggFbVS(Feedbacks $FB, CT, L, VM$)
0: $fL \leftarrow fL \in FB \{ \text{Agg. } L \text{ to } VS \text{ cong.} \}$
0: for all $L_x \in L$ do
0: \hspace{1em} $pe \leftarrow Pt(\text{current virtual switch}, L_x)$
0: \hspace{1em} $tL \leftarrow \sum CT_{pe} \{ \text{Agg. } VS \text{ to } L \text{ cong.} \}$
0: end for
0: $v \leftarrow \text{current virtual switch ID}$
0: for all $VM_x \in VS_{vnh+vh\ldots vnh+vh+1}$ do
0: \hspace{1em} $pe \leftarrow P(\text{current virtual switch}, VM_x)$
0: \hspace{1em} $fVS \leftarrow CT_{pe} \{ \text{Agg. } VS \text{ to } VM \text{ cong.} \}$
0: \hspace{1em} Send $fS, tS, fL, tL, fVS$ to $VM_x$
0: end for
0: end procedure
Algorithm 4.5.2 Extended Congestion Aggregation & Report

0: procedure AggrRptVM(Feedback $FB$)
0: $i \leftarrow$ current VM ID
0: $UVS_i \leftarrow AB(P_{out}^i)$ \{BW to VS.\}
0: $DVS_i \leftarrow AB(P_{in}^i)$ \{BW from VS.\}
0: $fS, tS, fL, tL, fVS \leftarrow fS, tS, fL, tL, fVS \in FB$
0: $\beta \leftarrow \frac{fVS}{(C - DVS_i)}$
0: $VStoLBW \leftarrow C - \frac{tL}{\beta}$
0: $UL_i \leftarrow \min(VStoLBW, UVS_i)$ \{BW to L.\}
0: $LtoVSBW \leftarrow C - \frac{fL}{\beta}$
0: $DL_i \leftarrow \min(LtoVSBW, DVS_i)$ \{BW from L.\}
0: $LtoSBW \leftarrow sC - \frac{tS}{\beta}$
0: $US_i \leftarrow \min(LtoSBW, UL_i)$ \{BW to S.\}
0: $StoLBW \leftarrow sC - \frac{fS}{\beta}$
0: $DS_i \leftarrow \min(StoLBW, DL_i)$ \{BW from S.\}
0: Send $US_i, DS_i, UL_i, DL_i, UVS_i, DVS_i$ to scheduler
0: end procedure

=0
tune one server, backup its disk content with ClusterManager, and effortlessly recover the content to all other servers with ClusterManager so that all servers are fine tuned. If we accidentally destroy the operating systems on some servers when we customize the network stacks, we can easily recover some default or check point disk content to those servers and they are immediately back to work.

4.7 Experiment Evaluation
4.7.1 Experiment Configuration

We implement Coflourish and Varys. The Coflourish’s switch is created by integrating the Open vSwitch with DPDK [36] and CONGA’s DRE. The Coflourish and Varys implementations are evaluated using experiments on the 1Gb/s CRON-DC testbed. The experiment DCN has the topology in Figure 4.9 with 4 spine switches, 4 leaf switches, and 16 hosts. To achieve the typical 10:1 oversubscrip-
tion ratio between the spine switches and leaf switches, we set the link bandwidths between hosts and leaf switches to 1Gb/s and limit the link bandwidths between the spine switches and leaf switches to 100Mb/s. Each host runs an virtual switch. Each virtual switch connects to 2 KVM virtual machines (VMs). Each host has 8 Xeon 2.33Hz CPU cores, 16GB main memory, and 1 1Gb/s NIC. Each VM has 2 CPU cores, 2GB main memory, and 1 1Gb/s NIC. The link bandwidths between the VS and VMs are 1Gb/s so that the oversubscription ratio of the VS is 2:1. A physical DCN consists of the spine, leaf, and host. A virtualized DCN consists of the spine, leaf, VS, and VM. While evaluating the Varys and the Coflourish on the physical DCN, all physical and virtual switches are standard Open vSwitches. While evaluating the Coflourish on the virtualized DCN, all physical and virtual switches are our customized Open vSwitch with DPDK for Coflourish. All ex-

FIGURE 4.10: The 1Gb/s DCN testbed topology in CRON-DC.

FIGURE 4.11: The 10Gb/s DCN testbed topology in CRON-DC.
TABLE 4.4: Default Framework Parameter Configuration for Experiment.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{FB}$</td>
<td>200 milliseconds.</td>
</tr>
<tr>
<td>$T_{sel}$</td>
<td>2 seconds.</td>
</tr>
<tr>
<td>$T_{str}$</td>
<td>200 milliseconds.</td>
</tr>
<tr>
<td>$thld_{BW}$</td>
<td>1/8 spine layer network link capacity.</td>
</tr>
<tr>
<td>$\tau$</td>
<td>50 milliseconds.</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>0.5.</td>
</tr>
<tr>
<td>$T_{dre}$</td>
<td>25 milliseconds.</td>
</tr>
</tbody>
</table>

Experiments are conducted once on the virtualized DCN and once on the physical DCN.

The workload is generated by the identical procedure used in the simulation, but the total number of coflows is 100. We design a coflow traffic generator to generate realistic network traffic according to a coflow traffic probability distribution. The coflow generator follows the master/slave architecture. A centralized master (1) reads the coflow traffic probability distribution and the DCN topology, and (2) accordingly generates individual coflows/flows and notifies the slaves on source and destination hosts. A slave program sends or receives actual flows on each host or VM.

Other default framework related parameter configuration is presented in Table 4.4. We are not able to probe any network state with the time interval smaller than 1 millisecond because the timer implementation in Open vSwitch with DPDK only achieves milliseconds resolution. Each result is an average of 5 runs.
4.7.2 Improvement of Coflourish

We evaluate Coflourish’s improvement with two criteria, available bandwidth estimation accuracy and average coflow completion time.

First, we compare the available bandwidth estimation accuracies. We use regular traffic load and vary the weight of background traffic. The metric, Normalized Inaccuracy (lower is better), is already defined in Equation 4.5. The result on the virtualized DCN is shown in Figure 4.12. The Coflourish’s estimation is 2.8% (10% background) through 33.7% (90% background) more accurate than the Varys’. When the weight of background traffic is small, the interference from the background traffic to the Varys is small so that the Varys can estimate available bandwidth with slight error using its simple NIC-based estimation. Varys’ estimation is slightly worse than the estimation of Coflourish based on accurate available bandwidth feedback information. When the weight of background traffic is large, the interference from the background traffic to the Varys is severe so that the simple bandwidth estimation of Varys gives largely biased result. Varys’ estimation is much more erroneous than the estimation of Coflourish based on accurate available bandwidth feedback information.

The result on the physical DCN is similar to the result on the virtualized DCN. The Coflourish’s estimation is 3.3% (10% background) through 35.1% (90% background) more accurate than the Varys’.

Second, we compare the average CCT. We use regular traffic load and vary the weight of background traffic. The result is shown in Figure 4.13. The Coflourish shortens the average CCT by 1.4% (10% background) through 25.4% (90% background) compared to Varys’. As explained in the previous analysis, Varys esti-
FIGURE 4.12: The inaccuracy of Coflourish normalized to Varys’. Lower inaccuracy is better.

mates bandwidth with slight error when encountering small amount of background traffic, and Varys estimates bandwidth with huge error when encountering large amount of background traffic. Thus, Varys generates a comparable coflow schedule to Coflourish’s when background traffic is light, and Varys generates a much worse schedule than the Coflourish’s when background traffic is heavy.

The result on the physical DCN is similar to the result on the virtualized DCN. The Coflourish shortens the average CCT by 1.9% (10% background) through 26.8% (90% background) compared to Varys’.

4.7.3 Impact of Coflow Size

We generate 3 traffic loads with 25%, 50%, and 75% large coflows. The weight of background traffic varies from 10% through 90% with a 10% interval. The result is shown in Figure 4.14.
FIGURE 4.13: The average CCT of Coflourish normalized to Varys’. Lower CCT is better.

With less than 30% of background traffic in network, background traffic impact Varys’ bandwidth estimation little. Varys still generates comparable schedule to Coflourish.

With 30% through 80% of background traffic in network, background traffic heavily interferes the bandwidth estimation of Varys. Average CCT increases much. In contrast, Coflourish still generates good schedule with available bandwidth feedback information. The more the large background traffic, the larger the average CCTs differ between the Varys’ schedule and Coflourish’s schedule.

With more than 80% of background traffic in network, the background traffic still heavily impacts Varys bandwidth estimation, but the network tends to be saturated. The gap of average CCT resulted in by different quality of schedule closes. The average CCT also stops decreasing. However, the 3 curves do not converge as close as the simulation results. The reason is that the layer-by-layer congestion aggregation and report take physical time. Thus, there are propagation
FIGURE 4.14: The CCTs of Coflourish under various traffic loads normalized to Varys’. Lower CCT is better.

and processing delays between each switch sending congestion information and the coflow scheduler receiving the estimated available bandwidth. More instant variations are likely to happen in workload with more not large coflows so that the coflow scheduler cannot reschedule in time to utilize most available bandwidth.

The result on the physical DCN is similar to the result on the virtualized DCN with slight improvements because congestion aggregation at the VS layer is removed so that the VM layer more instantly reports.

4.7.4 Impact of Feedback Interval and Bandwidth Variation Threshold

We probe the impact of two critical parameters in Coflourish, the feedback interval ($T_{FB}$) and the bandwidth variation threshold ($thld_{BW}$).

We set the $T_{FB}$ to 100, 200, and 400ms. We vary the weight of background traffic. The result is shown in Figure 4.15. A smaller $T_{FB}$ results in more frequent update of available bandwidth information. The up-to-date available bandwidth information results in a schedule which is closer to the optimal. A larger $T_{FB}$ results
in the opposite. The curves of 100 and 200ms overlap because the aggregation and processing of the congestion and available bandwidth information dominate in the feedback interval at the scale as small as 100ms.

The result on the physical DCN is similar to the result on the virtualized DCN. We substitute $1/4$, $1/8$, and $1/16$ the spine layer network link capacity into $thld_{BW}$. We vary the weight of background traffic. The result is shown in Figure 4.16. A smaller $thld_{BW}$ results in more frequent rescheduling with the lastest available bandwidth. Scheduling with up-to-date available bandwidth results in a schedule which is closer to the optimal. A larger $thld_{BW}$ results in the opposite. The curves of $1/8$ and $1/16$ link capacity overlap because the re-scheduling too frequently takes place when the threshold is $1/16$ so that the sending hosts or VMs are consistently in the process of adjusting sending rate towards the value which scheduler optimally allocates. The aggregate result is almost identical to the result of $1/8$ link capacity.
FIGURE 4.16: The impact of available bandwidth variation thresholds on the CCT of Coflourish normalized to Varys’. Lower CCT is better.

The result on the physical DCN is similar to the result on the virtualized DCN.

4.8 Related Work

**Coflow Scheduling** Coflow scheduling frameworks Varys [5] is the most representative work. The Varys' scheduler runs the SEBF heuristic algorithm to shorten the average CCT. Its network model is a non-blocking switch connecting all host machines, and every machine is equipped with Varys’ daemon for coordination. Rapier [79] enhances Varys by taking the routing into consideration while scheduling coflows. Its network model is identical to Varys’. A framework which feeds routing and network utilization information to the scheduler is assumed to exist.

In the loosely-coupled type, Aalo [6] is the de-facto reference design. The Aalo’s scheduler runs a Discrete Coflow-aware Least-Attained Service algorithm to shorten the average CCT. It can be used to schedule a wider range of cluster application but has slightly worse performance compared to Varys. Its network model is identical to Varys’.
Coflourish removes the strong assumption and requirement for the underlying DCN. Coflourish-based framework can smoothly co-exist with background traffic from not coordinated applications in a cloud environment.

**Flow Scheduling** Flow scheduling frameworks targeting short average flow completion time (FCT) are similar to existing coflow scheduling frameworks. pFabric [7] prioritizes flows with size at switches. Small flows are transferred first. Baraat [51] gives higher priority to flows with cumulative small size. Baraat also injects task ID into network traffic. With the same priority, flows are serve in the First-in-First-Out order according to the value of the task ID.

Flow scheduling frameworks are not aware of the logical relationships of flows in the application level. Blindly shortening FCT may result in a sub-optimal application completion time.

**Load Balancing** The load balancing frameworks evenly distribute traffic among available network links. CONGA [1] maintains the leaf-to-leaf switch congestion information in the spine-leaf fat-tree DCN and place flows on the least congested links.

Load balancing frameworks consider flow as the unit for balancing. Thus, it has similar problem as the flow scheduling frameworks. However, CONGA develops the DRE for estimating the congestion at each port. We adapt DRE to the switch-assisted bandwidth estimation function of Coflourish.

4.9 Conclusion

Existing coflow scheduling frameworks suffer from performance degradation in a shared network environment such as the cloud because of their unawareness of the background traffic. We propose Coflourish, the first coflow scheduling frame-
work which exploits congestion feedback assistances from SDN-enabled switches in the networks for available bandwidth estimation, to alleviate the problem resulted in by insufficient network information. Simulation results reveal that our framework is 78.7% more accurate in terms of bandwidth estimation and 75.5% better in terms of average CCT than the existing framework. Experiment results on software switch illustrate that our framework is 33.7% more accurate in terms of bandwidth estimation and 26.8% better in terms of average CCT than the existing framework. There is consistently an apparent gap between the simulation results and the experiment results due to the imperfect timing and synchronization implementations. By refining the software implementations or using specialized hardware switches, there is likely room for improvements. We also discover the potential for Coflourish to integrate with congestion-aware load balancing frameworks in the underlying DCN.
CHAPTER 5
CONCLUSION AND FUTURE WORK

In this study, first, we survey publications and documents of data center network, software-defined networking, and application-aware network scheduling/routing to construct an overview of the research trends in application-awareness in software-defined data center networks. We recognize the demands from upper-level applications and deficiencies of the existing software-defined data center network control frameworks. We set our goal to concentrate on designing application-aware SDD-CNCF for shortening average application completion time.

Second, we focus on improving application-aware SDDCNCF using OpenFlow. Its application-awareness is at the transfer application task level. Evaluations reveal that our SDDCNCF successfully reduces average transfer application completion time. A campus IT cyberinfrastructure, BIC-LSU, is built on our SDDCNCF for big data migration and analysis. BIC-LSU consists of an application-aware Clos OpenFlow network, high-performance cache storage servers, customized high-performance transfer applications, a light-weight control framework to manipulate existing big data storage systems and job scheduling systems, and a comprehensive social networking-enabled web portal. BIC-LSU achieves 40Gb/s disk-to-disk big data transmission, maintains short average transmission task completion time, enables the convergence of control on commonly deployed storage and job scheduling systems, and enhances easiness of big data analysis with a universal user-friendly interface. BIC-LSU software requires minimum dependencies and has high extensibility. Other research institutes can easily customize and deploy BIC-LSU as an augmented service on their existing IT infrastructures.
Third, on the basis of the success of BIC-LSU, we design a more sophisticated coflow level application-aware SDDCNCF using OpenFlow, MinCOF, for big data storage area network. MinCOF incorporates all critical features of coflow scheduling/routing frameworks, per-flow rate-limiting, coflow routing, coflow scheduling. MinCOF is further optimized for Spine-leaf DCN topology. Experiment results illustrate that MinCOF achieves shorter average coflow completion time than existing frameworks.

Last but not least, we recognize the limitation of OpenFlow SDN resulted in by the communication overhead between the switch and remote controller and design an SDN switch-assisted coflow level application-aware SDDCNCF, Coflourish. The switch is designed to coordinate and calculate available bandwidth in DCN for more precise coflow schedules. We implement the switch on Open vSwitch with fast packet processing library, DPDK. Both simulations and experiments show that Coflourish outperforms the existing coflow scheduling. However, the experiment results reveal a noticeable degradation from the simulation results. The timing and synchronization issues in practice greatly confine the effectiveness of Coflourish. One potential future improvement might be tightly integrating Coflourish into OpenSwitch switch OS on physical switches. The specialized switch hardware which is optimized for massive packet processing is likely to alleviate the overhead which our pure software implementation suffers from. Another potential future improvement might be implementing all software components in compiled programming languages to shorten the processing delay in each software component.
References


[28] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild,” in *Proceedings of the 10th ACM SIGCOMM Conference*


Vita

Chui-Hui Chiu is from Taipei City, Republic of China (Taiwan). He received a Bachelor of Science in Computer Science and Education at Taipei Municipal University of Education (presently University of Taipei) in 2006. After finishing his mandatory military service, he joined Department of Computer Science at Louisiana State University and Agricultural and Mechanical College as a Master’s student in 2009. In 2011, he received a Master of Science in System Science and continued to pursue a doctoral degree. During his doctoral study, he concentrated on improving the performance of the data center network using the cutting-edge technologies of the software-defined networking and application-aware network. In addition to the academic research works, he developed the prototype of the production BIC-LSU service, constructed the 144-node Big Data Park research computer cluster, established the CRON-DC data center network research testbed, administrated the GENI Rack of LSU, and managed the CRON high-speed network testbed. He is currently a candidate for the degree of Doctor of Philosophy in Computer Science. He plans to graduate in December 2017.