Managing data traffic in both intra- and inter-datacenter networks

Hu, Zhiming

2016

Hu, Z. (2016). Managing data traffic in both intra- and inter- datacenter networks. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/68806>

<https://doi.org/10.32657/10356/68806>

Managing Data Traffic in both Intra- and Inter- Datacenter Networks

HU ZHIMING

School of Computer Science and Engineering

Nanyang Technological University

A thesis submitted to the Nanyang Technological University
in partial fulfilment of the requirement for the degree of
Doctor of Philosophy (Ph.D)

2016

To my family, for their unconditional love and endless support.

Acknowledgement

First I would like to thank my supervisor, Prof. Jun Luo, for his great support and guidance over the years. I have learned not only about solving problems, but also about finding the right research problem.

I would like to express my special thanks to my co-authors, Prof. Baochun Li, Prof. Kai Han, Prof. Yonggang Wen, Prof. Yan Qiao, and Dr. Liu Xiang, for their insightful discussions and invaluable suggestions.

My deepest gratitude goes to my parents, my parents-in-law, my wife, my son and my brothers, for their unconditional love and support. I owe to them every step of my progress in life.

I am greatly indebted to my friends, for their friendships and encouragement. I would always appreciate their lovely company during my Ph.D. study.

Abstract

To support large scale online services, governments and multinational companies such as Google and Microsoft have built a lot of datacenters across the world. As datacenter networks are critical on the performance of those services, both academic and industrial communities have started to explore how to better design and manage them. Among those proposals, most approaches are designed for intra-datacenter networks to improve the performance of services running in a single datacenter, while another trend of research aims to enhance the performance of services on inter-datacenter networks that connect geo-distributed datacenters. In this thesis, we first propose an efficient network monitoring system for intra-datacenter networks, which can provide valuable information for applications like traffic engineering and anomaly detection inside the datacenter networks. We then take one step further to design a new task scheduling algorithm that improves the performance of big data processing jobs across geographically distributed datacenters on top of inter-datacenter networks.

In the first part of the thesis, we innovate in designing a new monitoring framework in intra-datacenter networks to get the traffic matrix, which serves as critical inputs for a variety of applications in datacenter networks. Our preliminary study shows that we cannot estimate the traffic matrix accurately through only *Simple Network Management Protocol* (SNMP) counters because the number of available measurements (the link counters) is much smaller than the number of variables (the end-to-end paths) in datacenter networks. Thus we creatively take advantage of the operational logs in datacenter networks to provide extra measurements for the traffic estimation problem. Namely, we utilize the resource provisioning information in public datacenter networks and service placement information in private datacenter networks respectively to improve the estimation accuracy. Moreover, we also make use of the lowly utilized links in

datacenter networks to obtain a more determined network tomography problem. The extensive results have strongly confirmed the promising performance of our approach.

In the second part of the thesis, we seek to improve the performance of geo-distributed big data processing, which has emerged as an important analytical tool for governments and multinational corporations, on top of inter-datacenter networks. The traditional wisdom calls for the collection of all the data across the world to a central datacenter location, to be processed using data-parallel applications. This is neither efficient nor practical as the volume of data grows exponentially. Rather than transferring data, we believe that computation tasks should be scheduled where the data is, while data should be processed with a minimum amount of transfers across datacenters. To this end, we first formulate our problem as an integer linear programming problem (ILP). We then transform it to a linear programming problem (LP) that can be efficiently solved using standard linear programming solvers in an online fashion. To demonstrate the practicality and efficiency of our approach, we also implement it based on Apache Spark, a modern framework popular for big data processing. Our experimental results have shown that we can reduce the job completion time by up to 25%, and the amount of traffic transferred among different datacenters by up to 75%.

Keywords

Datacenter networks, traffic matrix, cloud computing, big data processing, distributed computing.

Contents

Acknowledgement	iii
Abstract	iv
List of Figures	ix
List of Tables	xi
1 Introduction	1
1.1 Background	1
1.2 Scope of Research	3
1.3 Contributions	4
1.4 Thesis Organization	4
2 Literature Survey	6
2.1 Architectures for Datacenter Networks	6
2.2 Traffic Measurements in DCNs	8
2.3 Network Tomography	10
2.4 Inter-Datacenter Networks	11
2.4.1 Data Transfers over Inter-Datacenter Networks	11
2.4.2 Big Data Processing over Inter-Datacenter Networks	12
3 Traffic Matrix Estimation in both Public and Private Datacenter Networks	14
3.1 Introduction	15
3.2 Definitions and Problem Formulation	18
3.3 Overview	21

3.3.1	Traffic Characteristics of DCNs	22
3.3.2	ATME Architecture	23
3.4	Getting the Prior TM among ToRs	24
3.4.1	Computing the Prior TM among ToRs Using Resource Provision- ing Information in Public DCNs	25
3.4.2	Computing the Prior TM among ToRs Using Service Placement Information in Private DCNs	29
3.5	Link Utilization Aware Network Tomography	33
3.5.1	Eliminating Lowly Utilized Links and Computing Prior Vector	33
3.5.2	Combining Prior TM with Network Tomography constraints	35
3.5.3	The Algorithm Details	35
3.6	Evaluation	36
3.6.1	Experiment Settings	37
3.6.2	Testbed Evaluation of ATME-PB	38
3.6.3	Testbed Evaluation of ATME-PV	39
3.6.4	Simulation Evaluation of ATME-PB	41
3.6.5	Simulation Evaluations of ATME-PV	44
3.7	Summary	46
4	Scheduling Tasks for Big Data Processing Jobs Across Geo-Distributed Datacenters	48
4.1	Introduction	49
4.2	Flutter: Motivation and Problem Formulation	51
4.3	Network-aware Task Scheduling across Geo-Distributed Datacenters	54
4.3.1	Transform into a Nonlinear Programming Problem	54
4.3.2	Transform the Nonlinear Programming Problem into a LP	57
4.4	Design and Implementation	59
4.4.1	Obtaining Outputs of the Map Tasks	60
4.4.2	Task Scheduling with Flutter	61
4.5	Performance Evaluation	62
4.5.1	Experimental Setup	62
4.5.2	Experimental Results	63
4.6	Summary	68

5 Conclusion	70
5.1 Research Contributions	70
5.2 Future Directions	72
References	74

List of Figures

3.1	An example of conventional DCN architecture, suggested by Cisco [1]. .	18
3.2	The TM across ToR switches reported in [2].	21
3.3	Link utilizations of three DCNs, with “private” and “university” from [3] and “testbed” being our own DCN.	23
3.4	The ATME architecture.	24
3.5	Each color represent one user. Here there are totally three users. v3, v5, v7, v8 are not used by any user in this case.	27
3.6	The correlations between traffic and service in our datacenter.	30
3.7	Four different line styles represent four flows and three different colors represent three services.	33
3.8	After reducing the lowly utilized links in Figure 3.7	34
3.9	Hardware testbed with 10 racks and more than 300 servers.	38
3.10	The CDF of RE and RMSRE of ATME-PB and two baselines on testbed.	38
3.11	The CDF of RE and RMSRE of ATME-PV and two baselines on testbed.	39
3.12	The CDF of RE (a), the RMSRE (b), and the RMSE (c) of ATME-PB and two baselines for estimating TM under tree architecture.	40
3.13	The CDF of RE (a), the RMSRE (b), and the RMSE (c) of ATME-PB and two baselines for estimating TM under fat-tree architecture.	41
3.14	The CDF of RE (a), the RMSRE (b), and the RMSE (c) of ATME-PV and two baselines for estimating TM under tree architecture.	44
3.15	The CDF of RE (a), the RMSRE (b), and the RMSE (c) of ATME-PV and two baselines for estimating TM under fat-tree architecture.	45
4.1	Processing data locally by moving computation tasks: an illustrating example.	50

4.2	The design of <i>Flutter</i> in Spark.	59
4.3	The job computation times of the three workloads.	65
4.4	The completion times of stages in WordCount, PageRank and GraphX.	65
4.5	The amount of data transferred among different datacenters.	68
4.6	The computation times of <i>Flutter</i> 's linear program at different scales.	69

List of Tables

3.1	Commonly used notations	20
3.2	Correlation Coefficients of the Working Example	32
3.3	The Computing Time (seconds) of ATME-PB, Tomogravity and SRMF under Different Scales of DCNs (Fat-tree)	43
3.4	The Computing Time (seconds) of ATME-PV, Tomogravity and SRMF under Different Scales of DCNs (Tree)	46
4.1	Available bandwidths across geographically distributed datacenters. . .	52
4.2	Available bandwidths across geo-distributed datacenters (Mbps). . . .	64

Chapter 1

Introduction

Nowadays, datacenters have been widely deployed in universities and enterprises to support many kinds of applications ranging from web applications such as search engines, mail services, and news websites to computation and storage intensive applications such as scientific computation, data mining, and video streaming. *Datacenter networks* (DCNs), which connect not only servers and network devices in datacenters but also users and services, have a great impact on the performance of those services hosted in the datacenters. To this end, plenty of proposals from both academic and industrial communities are searching for ways to improve the performance of DCNs and applications on top of DCNs. Among those proposals, profiling the DCNs and optimizing the performance of applications on top of DCNs are two crucial issues to be solved and thus attract a lot of research attentions.

1.1 Background

Profiling the DCNs can provide the detailed traffic characteristics in the DCNs, which can help us understand the traffic characteristics in DCNs and thus has the potential to guide the designs of network architectures and applications in DCNs. For instance, Srikanth et al. [2] show that one *Top-of-Rack* (ToR) switch may only communicate with a few other ToRs instead of all the other ToRs, which further motivates the designs of wireless datacenter networks [4]. Moreover, if the link utilizations in the DCNs are low, it is possible to shut down part of the switch ports or even the whole switch based on

the observations to save energy [5]. Through these simple examples, we can see that network characteristics are very beneficial for network designs and operations in DCNs.

Network characteristics are normally described by *traffic matrix* (TM) with rows denoting traffic entries and columns representing different time slices. Most proposals adopt direct measurements to get the TM in DCNs. These proposals can be divided into server-based approaches and switch-based approaches. On one hand, the server-based approaches [6, 7] need to instrument all the servers/VMs first and then monitor the traffic flows. After that, the overall TM can be deduced from the data collected in all the servers/VMs. The drawbacks of these approaches are that instrumentations are needed for all the servers/VMs, especially when the hardware and software in servers/VMs are heterogeneous. In addition, these approaches would generate a huge amount of data in each server/VM [8], which incurs storage and computation overhead for storing and analyzing the data. On the other hand, the switch-based approaches (e.g., [9]) propose to instrument the switches, or directly use programmable switches such as OpenFlow-enabled [10] switches, to record the traffic flows. These approaches have similar limitations with the server-based approaches, as they both need instrumentations in the DCNs for the measurement tasks. The willingness of the owners for applying the instrumentations and upgrading datacenters would be another obstacle. Different from prior solutions, our measurement framework can estimate the TMs from the available SNMP link counts without the need of instrumenting datacenters, which makes it much more practical and easier to be adopted in practice.

Estimating the TMs can help improve the performance of applications inside the datacenters. However, the number of applications on top of several geo-distributed datacenters are growing rapidly and those applications have become an important part of applications in datacenters nowadays [11]. To reduce the service latency, some applications like Google search and Facebook websites are normally deployed in several datacenters located worldwide. Those applications would generate tremendous data ranging from user activities to application logs in those geo-distributed datacenters [12]. To analyze the geo-distributed data, traditional solutions call for collecting all the data to a centralized location before analysis [11, 12]. However, given the expensive and low bandwidths among different datacenters [13], traditional solutions are no more efficient nor practical. In this case, *geo-distributed big data*, which analyzes the geo-distributed

data with the least amount of transfers among different datacenters, appears to be a better solution.

Among the recent proposals for the geo-distributed data analysis, job completion time and the cost incurred during the process of the job are two main optimization goals. In [11], they propose to minimize the amount of data transfers among different datacenters through the optimizations of query execution and data replication. Another proposal [13] aims to reduce the amount of data transfers among different datacenters by solving the generalized min-k cut problem and is implemented in Spark [14]. These two proposals are both designed for reducing the data transfers (costs) for running the jobs. Qifan et al. propose to shorten the delay of geo-distributed data analysis jobs [12] by joint optimizing the data placements and reduce task placements, while having a few unrealistic assumptions such as the bottlenecks in the bandwidths among different datacenters exist in the sites.

1.2 Scope of Research

In this thesis, our first focus lies in estimating the TMs in both public and private DCNs as TMs are critical inputs for network designs and operations. Thus in the first part, we focus on intra DCNs. As the number of variables (end-to-end paths) are much more than the number of available measurements (link counts), directly estimating the TMs accurately relying only on the SNMP link counts are thus impractical. Therefore, we try to deduce more information from the operational logs in datacenters to provide more measurements for our estimation problem. In a nutshell, we attempt to estimate the TMs in both public and private DCNs given the easily available SNMP link counts and operational logs in DCNs.

Besides the workloads inside a single datacenter, analyzing the geo-distributed data has become another increasingly important sort of workloads. Therefore, our focus in the second part is inter datacenter networks. Giving the fact that traditional solutions that gather the geo-distributed data before analysis are no more efficient nor practical, we propose to design an efficient task scheduling algorithm for geo-distributed big data analysis framework to decrease both the job completion time and the amount of data transfers among datacenters. Through this way, we attempt to improve the overall performance of geo-distributed big data processing in the second part of the thesis.

1.3 Contributions

In this thesis, we are making the following contributions:

- We propose an **efficient traffic matrix estimation** framework for both public and private DCNs. This framework can efficiently estimate the TMs in intra DCNs with high precision through the SNMP link counts and the operational logs in DCNs.
 - We reveal two observations about the traffic characteristics in DCNs, which serve as part of the motivations for the designs of our framework.
 - We first estimate the prior TMs based on the SNMP link counts and operational logs in DCNs. We then obtain the final estimation by refining the prior TMs through the optimization methods.
- We propose a **task scheduling algorithm for geo-distributed big data processing**. This paradigm tackles the problem of reduce task scheduling when considering the exact sizes and locations of inputs for reduce tasks and network bandwidths of inter DCNs.
 - We focus on the geo-distributed big data processing, which poses new challenges to the big data processing framework as the bandwidths among datacenters are both diverse and low.
 - We formulate an integer linear programming problem (ILP) for the task scheduling problem. We then analyze the formulation and transform it to a linear programming problem (LP) that can be efficiently solved by standard LP solvers.
 - We implement our task scheduler in Apache Spark and evaluate it with representative applications on real datasets.

1.4 Thesis Organization

This thesis is organized as follows:

In Chapter 2, we survey related work of this thesis. We first review the literature concerning the designs of datacenter network architectures. Then we discuss some

measurement studies in DCNs and several traditional network tomography proposals in ISP networks. Finally we study a few representative work for data transfers and geo-distributed big data processing systems over inter-datacenter networks.

In Chapter 3, we present our first part of work on estimating the traffic matrix in both public and private DCNs. We innovating in utilizing the operational logs in public and private DCNs and proposing an efficient way to deduce the prior TMs first. We then feed the prior TM and SNMP link counts into an optimization framework and obtain the final estimation of TMs in intra DCNs.

In Chapter 4, we show our second part of work on task scheduling for geo-distributed big data processing systems. We first reveal that the bandwidths among datacenters are diverse and low, which means that we should carefully design the task placements to avoid network bottlenecks. We then formulate the problem while considering the network bandwidths and the characteristics of data analysis frameworks. We finally transform the problem to an efficient LP and implement it in Spark.

In Chapter 5, we conclude the work we have done. We summarize our research outputs with respect to efficient traffic matrix estimation in intra-DCNs and task scheduling algorithm in geo-distributed big data processing on top of inter-DCNs. We sum up the insights we learned from these results regarding these two schemes. We also point out some future research directions that could extend our current work.

Chapter 2

Literature Survey

In this chapter, we first review several prevalent datacenter network architectures, which have a great impact on the designs of systems and algorithms for *datacenter networks* (DCNs). We then survey the proposals related to the measurements and traffic characteristics in DCNs. After that, *network tomography*, a classic technique to obtain the traffic matrix through easily available link counters in the networks, is studied in detail, followed by the discussions of the applications on top of inter-datacenter networks.

2.1 Architectures for Datacenter Networks

The design of datacenter network architectures is a vital topic in the research areas of DCNs and is one of the key factors on the performance of DCNs. In this section, we briefly discuss some existing proposals of datacenter network architectures, which are popular in both academic and industrial communities. These designs for datacenter network architectures can be roughly divided into two categories: switch-centric architectures and server-centric architectures.

Switch-centric architectures. For switch-centric architectures, we mainly introduce tree [1], fat-tree [15] and VL2 [16]. To the best of our knowledge, the most widely used architecture for DCNs is a tree-based architecture recommended by Cisco [1], which has several advantages, for instance, simple and easy to deploy. It typically adopts two- or three-tier datacenter switching infrastructures. Three-tier DCNs are typically composed of core switches, aggregation switches, and *Top of Rack* (ToR)

switches, while two-tier datacenter architectures contain only core-switches and ToR switches. For its simplicity and ease of use, the tree-based architecture has been widely deployed in universities and small companies based on the statistics in [3].

However, conventional tree architecture also has many disadvantages, for example, bad scalability, static network assignments, and small server to server capacity, which in turn serve as the motivations for many novel datacenter network architectures such as fat-tree [15] and VL2 [16].

To solve the performance issues identified in the traditional architectures, fat-tree is one of the most popular solutions. Fat-tree [15] adopts a special instance of Clos topology [17] and has rich links between the core switches and aggregation switches compared to the conventional tree architecture. It is straightforward to find out that this topology is helpful for distributing the packets to multiple equal cost paths, thus greatly increases the server to server capacity. Moreover, it uses two level table for addressing and routing to make efficient routing.

Fat-tree [15] has better performance than the conventional tree architecture in terms of bisection bandwidth and scalability, but many problems are still unresolved such as the application isolation problem and the static resource assignments problem. Next, we are ready to have a look at VL2 [16], which aims to solve these problems.

Similar to the fat-tree [15] architecture, VL2 adopts another special instance of the Clos topology [17] in its architecture. But VL2 forms the Clos topology [17] among the core switches and the aggregation switches rather than among the aggregation switches and the ToR switches in the fat-tree topology [15]. VL2 leverages flat addressing to create the illusion that all servers are connected by a single non-inferencing Ethernet switch. As a result, the management platform can easily assign any server to any service on demand in a real-time fashion. When comes to how to benefit from the rich links among core switches and aggregation switches, VL2 uses *Valiant Load Balancing* (VLB) to cope with the volatility of workloads, traffic and failure patterns. In the traditional DCNs, ECMP [18] is commonly used to spread the traffic to multiple equal cost paths. While in VL2, a flow randomly selects an intermediate core switch to further deliver the packets of the flow. This load balance scheme is shown to be more effective in VL2 than ECMP [16].

Server-centric architectures. Different from switch-centric architectures, where switches are the main components for interconnecting and routing, servers are the main

components for interconnecting and routing in server-centric architectures. There are a few popular server-centric architectures such as DCell [19], BCube [20], FiConn [21], and MDCube [22]. Here we will introduce a few representative server-centric network architectures in detail.

DCell is a recursive server-centric datacenter architecture, which is designed for scalability, fault tolerant, and high network capacity. In the architecture, each server is connected with multiple level of DCells through multiple links. To provide scalability, it only uses the mini-switches instead of high-end switches, and the higher level of DCell can be constructed by the lower level of DCells [19]. The feature of fault tolerant is both for the rich connections in the architecture, but also for the distributed fault tolerant routing algorithm proposed for DCell. In addition, in the design of the architecture, it does not have the bottleneck links as in the traditional tree architecture, thus it has high network capacity compared with the traditional tree architectures.

Besides DCell, BCube [20] is another server-centric datacenter architecture that can also be constructed recursively. More specifically, $BCube_k(k \geq 1)$ can be constructed by n $BCube_{k-1}$ s and n^k n -port switches. Given the structure, in $BCube_k$, there are $k + 1$ parallel paths between any two servers and the length of the longest paths among all the server pairs is also $k + 1$ [20]. Thus it can accelerate the communication patterns like “one to many” and “one to all”, which are common communication patterns in data processing applications, and can also provide graceful performance degradation when failure happens because of its parallel paths.

Similar with DCell and BCube, FiConn [21] also aims at improving the datacenter networks by interconnecting servers, and it creatively utilizes the commonly unused backup network ports shipped with commodity servers. As the total number of ports in each server is two, the degree of server nodes is always two in this architecture. Thus it may result in loss of some flexibility compared with DCell, while for its special architecture, it has lower wiring costs and can balance the links in different levels through a distributed traffic-aware routing scheme.

2.2 Traffic Measurements in DCNs

As the architectures for datacenter networks are apparently different from other networks such as *Internet service provider* (ISP) networks, we can almost make sure that

the traffic in DCNs would show different characteristics. However, even though numerous studies have been conducted to improve the performance of DCNs [9, 15, 16, 20, 23, 24] and the awareness of traffic flow pattern is a critical input to all the above network designs or operations, little work has been devoted to the traffic measurements. Most proposals, when in need of *traffic matrix* (TMs), rely on either switch-based or server-based methods.

The switch-based methods (e.g., [9]) normally adopt programmable ToR switches (e.g., OpenFlow [10] switch) to record flow data, then utilize those flow data for higher layer applications or measurements [25, 26, 27]. However, these methods may not be feasible for three reasons. First, they incur high switch resource consumptions to maintain the flow entries. For example, if there are 30 servers per rack, the default lifetime of a flow entry is 60 seconds, and on average 20 flows are generated per host per second [28], then the ToR switch should be able to maintain $30 \times 60 \times 20 = 36,000$ entries, while the commodity switches with OpenFlow support such as HP ProCurve 5400zl can only support up to 1.7k OpenFlow entries per linecard [6]. Second, hundreds of controllers are needed to handle the huge number of flow setup requests. In the above example, the number of control packets can be as many as 20M per second. A NOX controller can only process 30,000 packets per second [28]; thus it needs about 667 controllers to handle the flow setups. Finally, not all the ToR switches are programmable in DCNs with legacy equipments, while the datacenter owners may not be willing to pay for upgrading the switches.

The server-based methods require to instrument all the servers to support flow data collection [6, 7]. In an operating datacenter, it is very difficult to instrument all the servers while supporting a lot of ongoing cloud services. Also, the heterogeneity of servers may also complicate the problem: dedicated software may need to be prepared for different servers and their OSs. Moreover, it does cost server resources to perform flow monitoring. Finally, similar to the switch-based approaches, the willingness of datacenter owners to upgrade all servers may yet be another obstacle.

Besides the above mentioned work, some other work [3, 8, 29] reveal some traffic characteristics in the operational datacenters. More specifically, in [8], it first shows how traffic is exchanged among servers and then presents some characteristics of flows in DCNs such as the lifetimes of flows and the inter-arrival times of flows. The paper also claims that network tomography cannot be adopted directly in datacenters through

the experiments, which serves as part of our motivations for the first part of the thesis. While in [3, 29], the focus is more about the detailed communication characteristics including the flow-level characteristics and packet-level characteristics. It also discusses the utilizations of links in datacenters according to the data from several operational datacenter networks.

2.3 Network Tomography

As discussed in the last section, direct measurements are normally expensive, now let us have a look at network tomography, a lightweight measurement method widely adopted in ISP networks. *Network tomography* [30, 31, 32, 33, 34, 35, 36, 37], which estimates the TMs from the ubiquitous link counters in the network, attracts a lot of attentions in the ISP networks.

In [31], it proposes a prior-based network tomography approach named Tomo-gravity. More specifically, it adopts the gravity model to get the prior TM and then formulate a least square problem to obtain the final estimation. The final estimation is the estimation that both meets the constraints of network tomography but also the closest estimation to the prior estimation. We also introduce a prior-based estimation framework for datacenter networks in the first part of the thesis.

Tomo-gravity [31] is a classic algorithm that utilizes the spatial characteristics in the networks. Here, spatial characteristics mean how one network device is related to other network devices. However, besides spatial characteristics, temporal characteristics are also common phenomenons in the networks. For instance, the network status from late night hours are similar from day to day, which is for the rest habits of network users. Thus it is possible to utilize the temporal characteristics to estimate or predict the traffic status in the coming hours/days. This is the reason for the work in [33] that utilizes the temporal characteristics in the networks, where Kalman Filter [38] is used for modelling the traffic in continuous time slices and predicting the traffic in the next time slice.

A method that combines both spatial and temporal characteristics is presented in [32, 39], which applies compressive sensing methods to combine the spatial and temporal characteristics of TMs and proposes a low rank optimization algorithm to obtain their final estimations. In the paper, they also present the ways to gather the

spatial and temporal characteristics in the network. This general method can be used for applications such as tomography, prediction and network anomaly detection.

There are also recent work on network tomography [40, 41, 42]. The first work [40] aims to efficiently estimate the additive link metrics such as packet delays given the available network monitors. While [41] is proposed to maximize the identification of the network tomography problem by deciding where to place the network monitors. [42] is also about monitor placement, while its goal is to locate the node failures efficiently instead of estimating the additive link metrics.

Thus on one hand, as we can see in the Section 2.2, currently adopted measurement methods in DCNs are expensive. On the other hand, network tomography is a mature practice in ISP networks for network measurements. Therefore, we may wonder that why not try *network tomography* to reduce the measurement overheads in DCNs. Having been widely investigated in ISP networks [31, 32, 33], it would be very convenient if we can adapt network tomographic methods in DCNs and apply those state-of-art algorithms. Unfortunately, due to the rich connections among the switches in DCNs, the number of end-to-end paths are far more than the number of links, which makes the network tomographic problem much more under-determined than the case in ISP networks. From both the results in [8] and our experimental results, we find out that we cannot directly apply those network tomographic methods in DCNs. We will illustrate how we conquer the challenges in detail in Chapter 3.

2.4 Inter-Datacenter Networks

Nowadays, many multinational companies have built a lot of datacenters across the world to serve their customers globally. In this case, besides the importance of intra-datacenter networks, the performance of inter-datacenter networks are becoming increasingly important. Therefore, in this section, we first review some work about managing the data transfers over inter-datacenter networks, followed by the discussions of big data processing applications over inter-datacenter networks.

2.4.1 Data Transfers over Inter-Datacenter Networks

NetStitcher [43] adopts a store-and-forward algorithm for bulk transfers across datacenters. While B4 [44] and SWAN [45] both employ traffic engineering mechanisms

to improve the utilizations of inter-datacenter networks. Also they both adopt *software defined networking* (SDN) for centralized network managements in the high level. While they have different focuses. In B4 [44], the main focus is to find the solutions to accommodate traditional routing protocols with OpenFlow-based switch control. While in SWAN [45], a scalable global allocation algorithm that maximizes the network utilization, a congestion-free rule update mechanism, and making the best use of limited forwarding table entries are the three main challenges tackled in the paper.

However, none of the above mentioned approaches consider the deadlines of transfers among datacenters, which are necessary for implementing the *service level agreements* (SLAs) for public cloud providers. Amoeba [46] makes efforts to achieve deadline guaranteed data transfers over inter-datacenter networks. It first proposes an adaptive spatial-temporal scheduling algorithm to decide whether to accept the coming request. If the request is admitted, it then applies a two step heuristic to reschedule the existing requests along with the newly arrival request. Finally, it adopts a bandwidth scheduling algorithm to maximize the utilizations of the networks.

2.4.2 Big Data Processing over Inter-Datacenter Networks

After reviewing the data transfers solutions over inter-datacenter networks, we now show a few most related work in geo-distributed big data processing, which can be roughly divided into two categories based on their objectives: reducing the amount of traffic transferred among different datacenters and shortening the whole job completion time. We also survey some other work related to scheduling in general distributed data processing systems.

Reducing the amount of traffic among different datacenters is proposed in [11, 13, 47]. In [11], they design an integer programming problem for optimizing the query execution plan and the data replication strategy to reduce the bandwidth costs. As they assume each datacenter has limitless storage, they aggressively cache the results of prior queries to reduce the data transfers of subsequent queries. In Pixida [13], they propose a new way to aggregate the tasks in the original DAG to make the original DAG simpler. Namely, they propose a new generalized min-k-cut algorithm to divide the simplified DAG into several parts for execution, and each part would be executed in one datacenter. However these solutions only address bandwidth cost without carefully considering the job completion time.

The most related recent work is Iridium [12] for low latency geo-distributed analysis, while we have some significant differences with it. First they assume that the network connecting the *sites* (datacenters) are congestion-free and the network bottlenecks only exist in the up/down links of VMs. This is not the case in our measurements. In our measurements, the in/out bandwidth of VMs are both 1Gbps in intra datacenters, while the bandwidth among VMs in different datacenters are only around 100Mbps. Therefore the network bottlenecks are more likely to exist in the network connecting the datacenters instead. Second, in their linear programming formulation for task scheduling, they assume reduce tasks are infinitesimally divisible and each reduce task would receive the same amount of intermediate results from the map tasks, which are not realistic assumptions as reduce tasks are not divisible with low overhead and the data skews are common in the data analysis frameworks [48]. While we use the exact amount of intermediate results that each reduce task would read from the outputs of map tasks. What is more, although they formulate the scheduling problem as a LP, in their implementation, they actually schedule the tasks by solving a mixed integer programming (MIP) problem as stated in their paper [12]. Besides Iridium, G-MR [49] is about executing a sequence of MapReduce [50] jobs on geo-distributed data sets with improved performance in terms of both job completion time and cost.

For scheduling in data processing systems, Yarn [51], Mesos [52], and Dynamic Hadoop Fair Scheduler (DHFS) [53] are resource provisioning systems designed for improving the cluster utilizations. Sparrow [54] is a decentralized scheduling system for Spark that can schedule a great number of jobs at the same time with small scheduling delays, and Hopper [55] is an unified speculation-aware scheduling framework for both centralized and decentralized schedulers. Quincy [56] is designed for scheduling tasks with both locality and fairness constraints. Moreover, there is plenty of work related to data locality such as [57, 58, 59, 60].

Chapter 3

Traffic Matrix Estimation in both Public and Private Datacenter Networks

Understanding the pattern of end-to-end traffic flows in *datacenter networks* (DCNs) is essential to many DCN designs and operations (e.g., traffic engineering and load balancing). However, little research work has been done to obtain traffic information efficiently and yet accurately. Researchers often assume the availability of traffic tracing tools (e.g., OpenFlow) when their proposals require traffic information as input, but these tools may have high monitoring overhead and consume significant switch resources even if they are available in a DCN (see Section 2.2). Although estimating the *traffic matrix* (TM) between origin-destination pairs using only basic switch SNMP counters is a mature practice in IP networks, traffic flows in DCNs show totally different characteristics, while the large number of redundant routes in a DCN further complicates the situation. To this end, we propose to utilize the resource provisioning information in public cloud datacenters and the service placement information in private datacenters for deducing the correlations among top-of-rack switches, and to leverage the uneven traffic distribution in DCNs for reducing the number of routes potentially used by a flow. These allow us to develop ATME (short for Accurate Traffic Matrix Estimation) as an efficient TM estimation scheme that achieves high accuracy for both public and private DCNs. We compare our two algorithms with two existing representative methods through both experiments and simulations; the results strongly

confirm the promising performance of our algorithms.

3.1 Introduction

As datacenters that house a huge number of inter-connected servers become increasingly central for commercial corporations, private enterprises and universities, both industrial and academic communities have started to explore how to better design and manage the *datacenter networks* (DCNs). The main topics under this theme include, among others, network architecture design [15, 16, 20], traffic engineering [9], scheduling in wireless DCNs [61, 62], capacity planning [24], and anomaly detection [23]. However, little is known so far about the characteristics of traffic flows within DCNs. For instance, how do traffic volumes exchanged between two servers or top-of-rack (ToR) switches vary with time? Which server communicates to other servers the most in a DCN? In fact, these real-time traffic characteristics, which are normally expressed in the form of *traffic matrix* (TM for short), serve as critical inputs to all the above DCN operations.

Existing proposals in need of detailed traffic flow information collect the flow traces by deploying additional modules on either switches [9] or servers [6] in small scale DCNs. However, both methods require substantial deployments and high administrative costs, and they are difficult to be implemented thanks to the heterogeneous nature of the hardware in DCNs [63]. More specifically, the switch-based approaches, on one hand, need all the ToRs to support flow tracing tools such as OpenFlow [10], and consume a substantial number of switch resources to maintain the flow entries.¹ On the other hand, the server-based approaches, which require instrumenting all the servers or VMs to support data collection, are not available in most datacenters [8] and are nearly impossible to be implemented peacefully and quickly while supporting a lot of cloud services in large scale DCNs.

It is natural then to ask whether we could borrow from *network tomography*, where several well-known techniques allow *traffic matrices* (TMs) of IP networks to be inferred from link level measurements (e.g., SNMP counters) [31, 32, 33]. As link level measurements are ubiquitously available in all DCN components, the overhead introduced by such an approach can be very light. Unfortunately, both experiments in medium scale

¹To the best of our knowledge, no existing switch with OpenFlow support is able to maintain so many entries in its flow table due to the huge number of flows generated per second in each rack.

DCNs [8] and our simulations (see Section 3.6) demonstrate that existing tomographic methods perform poorly in DCNs. This is attributed to the irregular behaviour of end-to-end flows in DCNs and the large quantity of redundant routes between each pair of servers or ToR switches.

There are actually two major barriers applying tomographic methods to DCNs. One is the sparsity of TM among ToR Pairs. This refers to the fact that one ToR switch may only exchange flows with a few other ToRs, as demonstrated in [2, 4, 8]. This fact substantially violates the underlying assumption of tomographic methods including, for example, the amount of traffic a node (origin) would send to another node (destination) is proportional to the traffic volume received by the destination [31]. The other barrier is the highly under-determined solution space. In other words, a huge number of flow solutions may potentially lead to the same SNMP byte counts. For a medium size DCN, the number of end-to-end routes is up to ten thousands [8] while the number of link constraints is only around hundreds.

As TMs are sparse in general, correctly identifying the zero entries in them may serve as crucial priors. In both public and private DCNs, if two VMs/servers are occupied by different users, which can be derived from *resource provisioning information*, we can be rather sure that these VMs/servers would not communicate with each other in most cases. Moreover, in private DCNs¹, we may further take advantage of having the *service placement information*. This allows us to deduce that two VMs/servers belonging to same user would probably not communicate with each other if they host different services, because different services in DCNs rarely exchange information [64].

In this chapter, we aim at conquering the aforementioned two barriers and making TM estimation feasible for DCNs, by utilizing the distinctive information or features inherent to these networks. First, we make use of the resource provisioning information in a public cloud and the service placement information in a private datacenter (both can be obtained from the controller node of DCNs) to derive the correlations among ToR switches. The communication patterns among ToR pairs inferred by such approaches are far more accurate than those assumed by conventional traffic models (e.g., the gravity traffic model [31]). Second, by analyzing the statistics of link counters, we find that the utilizations of both core links and aggregation links are extremely uneven. In

¹For private DCNs, the owner knows everything about what services are deployed and where the services are hosted in the datacenter.

other words, there are a considerable number of links undergoing very low utilization during a particular time interval. This observation allows us to eliminate the links whose utilization is under a certain (small) threshold and to substantially reduce the number of redundant routes. Combining the aforementioned two methods, we propose ATME (Accurate TM Estimation) as an efficient estimation scheme to accurately infer the traffic flows among ToR switch pairs without requiring any extra measurement tools. In summary, we make the following contributions in this chapter.

- We creatively use resource provisioning information in public datacenters for deriving the prior TM among ToRs. We group all the VMs into several clusters with respect to different users, resulting in the effect that communications only happen within the same cluster and the potential traffic patterns are epitomized among all VMs in turn.
- We pioneer in using the service placement information in private datacenters to deduce the correlations of ToR switch pairs, and we also propose a simple method to evaluate the correlation factor for each ToR pair. Our traffic model, assuming that ToR pairs with a high correlation factor may exchange higher traffic volumes, is far more accurate for DCNs than conventional models used for IP networks.
- We innovate in leveraging the uneven link utilization in DCNs to remove the potentially redundant routes. Essentially, we may consider the links with very low utilization as non-existent without affecting much the accuracy of the TM estimation, while they effectively lessens the redundant routes in DCNs, resulting in a more determined tomography problem. Moreover, we also demonstrate that changing a low-utilization threshold has an effect of trading estimation accuracy for its complexity.
- We propose ATME as an efficient scheme to infer the TM for DCN ToRs with high accuracy in both public and private DCNs. ATME first calculates a prior assignment of traffic volumes for each ToR pairs using aggregated traffic of VM pairs (in public DCNs) or the correlation factors (in private DCNs). Then it removes lowly utilized links and thus operates only on a sub-graph of the DCN topology. It finally adapts a quadratic programming to determine the TM under

Figure 3.1: An example of conventional DCN architecture, suggested by Cisco [1].

the constraints of the tomography model, the enhanced prior assignments, and the reduced DCN topology.

- We validate ATME with both experiments on a relatively small scale datacenter and extensive large scale simulations in *ns-3*. All the results strongly demonstrate that our new method outperforms two representative traffic estimation methods on both accuracy and running speed.

The rest of the chapter is organized as follows. We present system model and formally describe our problem in Section 3.2. In Section 3.3, we reveal some traffic characteristics in DCNs and propose the architecture of our system design motivated by those traffic characteristics. After that, we present the way we compute the prior TM among ToRs and the link utilization aware network tomography in Section 3.4 and Section 3.5, respectively. We evaluate ATME using both real testbed and different scales of simulations in Section 3.6, before concluding this chapter in Section 3.7.

3.2 Definitions and Problem Formulation

We consider a typical DCN as shown in Figure 3.1. It consists of n ToR switches, aggregation switches, and core switches connecting to the Internet. Note that our method is not confined to this commonly used DCN topology; it accommodates other more advanced topologies also, e.g., VL2 [16], fat-tree [15], as will be shown in our simulations.

We let $x'_{i \rightarrow j}$ denote the estimated volume of traffic sent from the i -th ToR to the j -th ToR and $x'_{i \leftrightarrow j}$ denote the estimated volume of traffic exchanged between the two

switches. Given the volatility of DCN traffic, we further introduce $x'_{i \rightarrow j}(t)$ and $x'_{i \leftrightarrow j}(t)$ to represent values of these two variables at discrete time t , where $t \in [1, \Gamma]$.¹ Note that although these variables would form the TM for conventional IP networks, we actually need more detailed information of the DCN traffic pattern: the routing path(s) taken by each traffic flow. Therefore, we split $x'_{i \leftrightarrow j}(t)$ on all possible routes between the i -th and j -th ToRs. Let $\mathbf{x}(t) = [x_1(t), x_2(t), \dots, x_p(t)]$ represents the volumes of traffic on all possible routes among ToR Pairs, where p is the total number of the routes. Consequently, the *traffic matrix* $X = [\mathbf{x}(1), \mathbf{x}(2), \dots, \mathbf{x}(\Gamma)]$, where Γ is the total number of time periods, is the one we need to estimate.² Our commonly used notions are listed in Table 3.1, where we drop time indices for brevity.

The observations that we utilize to make the estimation are the *SNMP counters* on each port of the switches. Basically, we poll the SNMP MIBs for bytes-in and bytes-out of each port every 5 minutes. The SNMP data obtained from a port can be interpreted as the load of the link with that port as one end; it equals to the total volume of the flows that traverse the corresponding link. In particular, we denote ToR_i^{in} and ToR_i^{out} the total “in” and “out” bytes at the i -th ToR. We represent links in the network as $\mathbf{l} = \{l_1, l_2, \dots, l_m\}$, where m is the number of links in the network. Let $\mathbf{b} = \{b_1, b_2, \dots, b_m\}$ denote the bandwidth of the links, and $\mathbf{y}(t) = \{y_1(t), y_2(t), \dots, y_m(t)\}$ denote the traffic loads of the links at discrete time t , and $Y = [\mathbf{y}(1), \mathbf{y}(2), \dots, \mathbf{y}(\Gamma)]$ becomes the *load matrix*.³

Based on the network tomography, the correlation between traffic assignment $\mathbf{x}(t)$ and link load assignment $\mathbf{y}(t)$ can be formulated as

$$\mathbf{y}(t) = A\mathbf{x}(t) \quad t = 1, \dots, \Gamma, \quad (3.1)$$

where A denotes the routing matrix, with rows corresponding to links and columns indicating routes among ToR switches. $a_{k\ell} = 1$ if the ℓ -th route traverses the k -th link; $a_{k\ell} = 0$ otherwise. In this chapter, we aim to efficiently estimate the TM X using the load matrix Y derived from the easy-collected SNMP data.

¹Involving time as another dimension of the TM was proposed earlier in [32, 33].

²Here we only estimate the TMs among ToR switches. The problem of estimating the TMs among servers is much more under-determined and thus is left for future work.

³We only consider intra-DCN traffic in this chapter. However, our methods can easily take care of DCN-Internet traffic by considering the Internet as a “special rack”.

Notation	Description
n	The number of ToR switches in the DCN
m	The number of links in the DCN
p	The number of routes in the DCN
r	The number of services running in the DCN
Γ	The number of time periods
A	Routing matrix
$\mathbf{l}$	$\mathbf{l} = [l_i]_{i=1, \dots, m}$, where l_i is the i -th link
$\mathbf{b}$	$\mathbf{b} = [b_i]_{i=1, \dots, m}$, where b_i is the bandwidth of l_i
$\mathbf{y}$	$\mathbf{y} = [y_i]_{i=1, \dots, m}$, where y_i is the load of l_i
λ_i	The number of servers belonging to the i -th rack
$x'_{i \rightarrow j}$	The estimated volume of traffic send from the i -th ToR to the j -th ToR
$x'_{i \leftrightarrow j}$	The estimated volume of traffic exchanged between the i -th and j -th ToRs
$\mathbf{x}$	$\mathbf{x} = [x_i]_{i=1, \dots, p}$, where x_i is the traffic on the r -th routing path
$\bar{x}_i$	The prior estimation of the traffic on the i -th routing path.
ToR_i^{in}	The total “in” bytes of the i -th ToR during a certain interval
ToR_i^{out}	The total “out” bytes of the i -th ToR during a certain interval
S	$S = [s_{ij}]_{i=1, \dots, r; j=1, \dots, n}$, where s_{ij} is the number of servers under the j -th ToR that run the i -th service
$corr_{ij}$	The correlation coefficient between the i -th and j -th ToR.
θ	The threshold of link utilization
$\mathcal{T}$	The set of tuples for (userId, serverId, rackId)
$\mathcal{T}_u$	The set of VMs owned by the u -th user
$\mathcal{T}^i$	The set of VMs in i -th rack.
v_i^{in}	The total “in” bytes of i -th VM during a certain interval.
v_i^{out}	The total “out” bytes of i -th VM during a certain interval.
e_{ab}	The volume of traffic from a -th VM to b -th VM.
$\mathcal{U}$	The set of all users.
q	The total number of VMs in the datacenter.

Table 3.1: Commonly used notations

Figure 3.2: The TM across ToR switches reported in [2].

Although Eqn. (3.1) is a typical system of linear equations, it is impractical to solve it directly. On one hand, the traffic pattern in DCNs is practically sparse and skewed [2]. As shown in Figure 3.2, the sparse and skew nature of TM in DCNs can be immediately seen from the figure: only a few ToRs are hot and most of their traffic goes to a few other ToRs. On the other hand, as the number of unknown variables is much more than the number of observations in Eqn. (3.1), the problem is highly under-determined. For example in Figure 3.1, the network consists of 8 ToR switches, 4 aggregation switches and 2 core switches. The number of possible routes in the architecture is more than 100, while the number of link load observations is only 24. Even worse, the difference between these two numbers grows exponentially with the number of switches (i.e., the DCN scale). Consequently, directly applying tomographic methods to solve Eqn. (3.1) would not work, and we need to derive a new method to handle TM estimation in DCNs.

3.3 Overview

As directly applying network tomography to DCNs is infeasible for several challenges, we first reveal some observations about the traffic characteristics in DCNs. Then we

present the system architecture of ATME that applies these observations to conquer the challenges.

3.3.1 Traffic Characteristics of DCNs

As mentioned earlier, several proposals including [2, 4, 8] have indicated that the TM among ToRs is very sparse. More specifically, for each ToR in a DCN, it only exchanges data flows with a few other ToRs rather than most of them. Figure 3.2, adopted from [2], plots the traffic normalized volumes among ToR switches in a DCN with 75 ToRs. In Figure 3.2, we can see that each ToR is exchanging major flows with no more than 10 out of 74 other ToRs; the remaining ToR pairs share either very minor flows or nothing. Therefore our first observation is the following:

Observation 1: TMs among ToRs are very sparse, so prior TMs among ToRs should also be sparse with similar sparse patterns to gain enough accuracy for the final estimation.

Although we may infer the skewness in the TM in some way (more details can be found in the following sections), the existence of multiple routes between every ToR pair still persists. Interestingly, literature does suggest that some of these routing paths can be removed to simplify the DCN topology by making use of link statistics. According to Benson *et al.* [3], the link utilizations in DCNs are rather low in general. They collect the link counts from 10 DCNs ranging from private DCNs, university DCNs to Cloud DCNs and reveal that about 60% of aggregation links and more than 40% of core links have low utilizations (e.g. in the level of 0.01%). To give more concrete examples, we retrieve the data sets publicized along with [3], as well as the statistics obtained from our DCN, then we draw the CDF of core/aggregation link utilizations in three DCNs for one representative interval selected from several hundred 5-minute intervals in Figure 3.3. As shown in the figure, more than 30% of the core links in a private DCN, 60% of core links in an university DCN and more than 45% of aggregation links in our testbed DCN have the utilizations less than 0.01%.

Due to the low utilization of certain links, eliminating them will not affect much the estimation accuracy but will greatly reduce the number of possible routes between two racks. For instance, in an conventional DCN shown in Figure 3.1, eliminating a core link will reduce 12.5% of the routes between any two ToRs, while cutting an aggregation

Figure 3.3: Link utilizations of three DCNs, with “private” and “university” from [3] and “testbed” being our own DCN.

link halves the outgoing paths from the ToR below it. Therefore, we may significantly reduce the number of potential routes between any two ToRs by eliminating the lowly utilized links. Though this comes at a cost of slightly losing actual flow counts, the overall estimation accuracy or the running speed should be improved, thanks to the elimination of the ambiguity in the actual routing path taken by the major flows. Another of our observations is:

Observation 2: Eliminating the lowly utilized links can greatly mitigate the under-determinism of our tomography problems in DCNs; it thus has the potential to increase the overall accuracy and the speed of the TM estimation.

3.3.2 ATME Architecture

Based on these two observations, we design ATME as a novel prior-based TM estimation method for DCNs. In a nutshell, we periodically compute the prior TM among different ToRs and eliminate lowly utilized links. This allows us to perform network tomography under a more accurate prior TM and a more determined system (with fewer routes).

To the best of our knowledge, ATME is the first practical framework for accurate TM estimation in both public and private DCNs.

Figure 3.4: The ATME architecture.

As shown in Figure 3.4, our framework ATME contains two algorithms in total: ATME-PB for public DCNs and ATME-PV for private DCNs. Both of them take two main steps to estimate the TM for DCN ToRs. They have different ways to compute the prior TM among ToRs, while share the same link utilization aware tomography process as the second step. More specifically, first of all, ATME calculates the prior TM among different ToRs based on SNMP link counts and some other operational information such as resource provisioning information in a public DCN or the service placement information in a private DCN motivated by **Observation 1**. We elaborate the first step in Section 3.4. Second, it eliminates the lowly utilized links to reduce redundant routes and narrows the searching space of potential TMs suggested by the load vector $\mathbf{y}$ according to **Observation 2**. After that, it takes the prior TM among ToRs and network tomography constraints as input and solve the optimization problem to estimate the TM. We discuss the second step later in Section 3.5.

3.4 Getting the Prior TM among ToRs

An accurate prior TM is a good beginning for our prior-based network tomography algorithm. In this section, we introduce two light-weighted methods to get the prior TM $\mathbf{x}'$ with the help of operational information in DCNs. More specifically, as only

resource provisioning information is available in public DCNs, we use them to deduce the relationship between communication pairs. Since service placement information provides more information than resource provisioning information in private DCNs, we adopt service placement information instead to enhance the estimation accuracy of $\mathbf{x}'$ in private DCNs.

3.4.1 Computing the Prior TM among ToRs Using Resource Provisioning Information in Public DCNs

In a public cloud datacenter, we can only know which part of VMs is occupied by whom, but we have no idea about how users will use their VMs for privacy issues. However we can still use the resource provisioning information, which specify the mappings between VMs and users, to infer the sparse prior TM among ToRs for the following reasons. In a multi-tenant datacenter or IaaS platform, the hardware resources are provisioned to different users, with users accessing only their own VMs. Thus the VMs belonging to one user may only communicate with each other and would not communicate with VMs occupied by other users. The volume of traffic between two ToRs can be computed by the volume of traffic among VMs (occupied by same uses) in these two racks. Therefore, the problem of computing the prior TM among ToRs can be converted to computing the volume of traffic among VMs belonging to the same user.

To better illustrate the algorithm details, here are some notations that will be used in the following sections. After analyzing the resource provisioning information, we can get a tuple set $\mathcal{T}$, with each tuple containing the `userId`, `vmId` and `rackId`, respectively. For instance, for a tuple $(i, j, k) \in \mathcal{T}$, it means that the i -th user is using the j -th VM located at the k -th rack. Here one VM can only be located in one rack at a certain moment. For simplicity, $\mathcal{T}_u$ denotes the set of VMs owned by the u -th user. All the VMs in the i -th rack is stored in $\mathcal{T}^i$. We also use $\mathcal{U}$ to denote the set of all the users in the public DCNs. Because the computation process also takes the VMs into account, we also need the total in/out bytes of every VM during a certain interval, which can be easily collected through the hypervisor (Domain 0) of VMs. We use v_i^{in} and v_i^{out} to denote in/out bytes of the i -th VM.

3.4.1.1 Building Blocks of ATME-PB

Deriving VM Locations After analyzing the resource provisioning information, we can easily know the number of VMs and the locations of VMs owned by each user. Here for the location, we are only concerned with the index of the rack that one VM belongs to. For instance, if $user_1$ has two VMs (vm_1 (rack₁), vm_3 (rack₂)) and $user_2$ has one VM (vm_2 (rack₁)) allocated in a datacenter, we should get the following tuples after deriving the VM locations: ($user_1$, vm_1 , rack₁), ($user_2$, vm_2 , rack₁) and ($user_1$, vm_3 , rack₂). In this example, $\mathcal{T}_1$ is (vm_1 (rack₁), vm_3 (rack₂)), which denotes the set of VMs owned by $user_1$, and $\mathcal{T}^1$ consists of (vm_1 (rack₁), vm_2 (rack₁)), which specifies the set of VMs located at rack₁.

Computing the TM among VMs in each cluster There are roughly two steps in computing the TM among VMs. The first step is to group the VMs in $\mathcal{T}$ by user and to get $\mathcal{T}_u$ for all the users. Then in the second step, we need to compute the TM among VMs belonging to each user, given the total volume of traffic sent and received by each VM recorded by SNMP link counts during each interval. As we assume each VM will only communicate with other VMs that belong to the same user, a wise choice may be the gravity model [30], which is well suited to all-to-all traffic pattern. Therefore the volume of traffic from the a -th VM to the b -th VM e_{ab} can be computed by the gravity model as follows:

$$e_{ab} = v_a^{out} \frac{v_b^{in}}{\sum_{k \in \mathcal{T}_u} v_k^{in}}. \quad (3.2)$$

We conduct the same process for each group of VMs grouped by user and obtain the TM among VMs.

Computing Rack to Rack Prior After getting the TM among VMs for each user, we then compute the rack to rack prior TM based on the locations of VMs. As we have computed the volumes of traffic among VMs and we also know the racks where VMs are, we can just sum up those volumes of traffic among VMs in different racks to get the estimated prior TM among ToRs. For example, if vm_1 and vm_2 belong to rack₁ and rack₂ respectively, then the volume of traffic from rack₁ to rack₂ will add the volume of traffic from vm_1 to vm_2 .

3.4.1.2 The Algorithm Details

We present the details of computing resource provisioning enhanced prior TM among ToRs with $\mathcal{U}$ and in/out bytes of each VM as the input in **Algorithm 1**, where q is the total number of VMs in the DCN. It returns the prior traffic vector among ToRs $\mathbf{x}'$. More specifically, in line 1, we get $\mathcal{T}$ from resource provisioning information as additional information. From line 2 to line 6, we compute the prior volume of traffic among different VMs belonging to the same user. For each user $u \in \mathcal{U}$, the volume of traffic from the a -th VM to the b -th VM is calculated by Eqn. (3.2), according to the gravity traffic model. We then present our new ways to compute the prior volume of traffic between the i -th rack and the j -th rack in lines 9–11. Here, line 9 calculates the volume of traffic from the i -th ToR to the j -th ToR $x'_{i \rightarrow j}$ by summing up the volumes of traffic from a -th VM to b -th VM e_{ab} that originating at the i -th ToR and ending at the j -th ToR. Line 10 calculates $x'_{j \rightarrow i}$ in the similar way. $x'_{i \leftrightarrow j}$ in line 11 denotes the total volumes across the i -th ToR and the j -th ToR that equals to the summation of $x'_{i \rightarrow j}$ and $x'_{j \rightarrow i}$. As the algorithm runs for every time instance t , we drop the time indices. The complexity of the algorithm is $\mathcal{O}(\max\{|\mathcal{U}|\mathcal{T}_u^2, n^2\})$.

3.4.1.3 A Working Example

Figure 3.5: Each color represent one user. Here there are totally three users. v3, v5, v7, v8 are not used by any user in this case.

Here we give an example about how to estimate the TM among ToRs. As shown in Figure 3.5, there are three users in total. The VMs owned by those users are listed

Algorithm 1: Compute Resource Provisioning Enhanced Prior TM among ToRs

Input: $\mathcal{U}, \{v_a^{out} | a = 1, \dots, q\}, \{v_b^{in} | b = 1, \dots, q\}$
Output: $\mathbf{x}'$

- 1 Get $\mathcal{T}$ by analyzing the resource provisioning information.
- 2 **forall** $u \in \mathcal{U}$ **do**
- 3 **forall** $a, b \in \mathcal{T}_u$ **do**
- 4 $e_{ab} = v_a^{out} * \frac{v_b^{in}}{\sum_{c \in \mathcal{T}_u} v_c^{in}}$
- 5 **end**
- 6 **end**
- 7 **for** $i = 1$ **to** n **do**
- 8 **for** $j = i + 1$ **to** n **do**
- 9 $x'_{i \rightarrow j} \leftarrow \sum_{a \in \mathcal{T}^i} \sum_{b \in \mathcal{T}^j} e_{ab}$
- 10 $x'_{j \rightarrow i} \leftarrow \sum_{a \in \mathcal{T}^j} \sum_{b \in \mathcal{T}^i} e_{ab}$
- 11 $x'_{i \leftrightarrow j} \leftarrow x'_{i \rightarrow j} + x'_{j \rightarrow i}$
- 12 **end**
- 13 **end**
- 14 **return** $\mathbf{x}'$

below:

- user₁: vm₁(rack₁), vm₉(rack₅), vm_{11,12}(rack₆),
- user₂: vm₄(rack₂), vm₆(rack₃), vm_{13,14}(rack₇),
- user₃: vm₂(rack₁), vm₁₀(rack₅), vm_{15,16}(rack₈).

Those information can be gathered in the process of resource provisioning for the cloud users. Here for simplicity, the volume of traffic that each vm sends out and receives is 1000 MB and 100 MB for user1 and user3 in a certain interval, respectively. Then if we want to know the volume of traffic from ToR₁ to ToR₅, we should know the volume of traffic from v1 to v9 and the volume of traffic from v2 to v10, respectively. The volume of traffic from v1 to v9 is computed by the gravity model among v1, v9, v11 and v12. Therefore $e_{1,9} = 1000 * \frac{1000}{1000+1000+1000+1000} = 250$ MB. We can also get $e_{2,10} = 100 * \frac{100}{100+100+100+100} = 25$ MB. Thus based on our algorithm, the estimated prior volume of traffic from ToR₁ to ToR₅ is 275 MB. Similarly, we can also compute the prior volume of traffic from ToR₅ to ToR₁.

3.4.2 Computing the Prior TM among ToRs Using Service Placement Information in Private DCNs

In ATME-PB, we assume that only VMs/servers belonging to the same user may exchange information. However, it may not be the case if a user deploys different and unrelated services on two VMs/servers. As we can also take advantage of service placement information in private DCNs, it is natural for us to utilize the service placement information to derive more fine-grained relationship among communication pairs in private DCNs.

As stated in **Observation 1**, the TM among ToRs in DCNs is very sparse. According to the literature, as well as our experience with our own datacenter, the sparse nature of TM in DCNs may originate from the correlation between traffic and service. In other words, racks running the same services have higher chances to exchange traffic flows, and the volume of the flows may be inferred by the number of instances of the shared services. Bodík *et al.* [64] has analyzed a medium scale DCN and claimed that only 2% of distinct service pairs communicates with each other. Moreover, several proposals such as [65, 66] allocate almost all virtual machines of the same service under one aggregation switch to prevent traffic from going through oversubscribed network elements. Consequently, as each service may only be allocated to a few racks and the racks hosting the same services have a higher chance to communicate with each other, it naturally leads to sparse TMs among DCN ToRs. To better illustrate this phenomenon in our DCN, we show the placement of services in 5 racks using the percentage of servers occupied by individual services in each rack in Figure 3.6(a), and we depict the traffic volumes exchanged among these 5 racks in Figure 3.6(b). Clearly, the racks that host more common services tend to exchange greater volume of traffic (e.g., for racks 3 and 5, more than 50% of the traffic flows are generated by the “Hadoop” service), whereas those do not share any common services rarely communicate (e.g., racks 1 and 3). Therefore, we propose to compute the prior TM among ToRs by service placement information in private DCNs.

In ATME-PV, we use service placement information recorded by controllers of a private datacenter as the extra information. Suppose there are r services running in a DCN, we can then get the *service placement matrix* $S = [s_{ij}]_{i=1,\dots,r;j=1,\dots,n}$ with rows corresponding to services and columns representing the ToR switches. In particular,

$s_{ij} = k$ means that there are k servers under the j -th ToR running the i -th service in the DCN. We also denote λ_j the number of servers belonging to the j -th rack.

(a) Percentages of servers per service in our DCN. Only services in 5 racks are shown.

(b) The traffic volume from one rack (row) to another (column) with the service placements in (a).

Figure 3.6: The correlations between traffic and service in our datacenter.

3.4.2.1 Building Blocks of ATME-PV

The first step stems from **Observation 1**: we design a novel way to evaluate the correlation coefficient between two ToRs, leveraging on the easily obtained service placement information. We use $corr_{ij}$ to quantify the correlation between the i -th and the j -th ToRs, and we calculate it as follows:

$$corr_{ij} = \sum_{k=1}^r [(s_{ki} \times s_{kj}) / (\lambda_i \times \lambda_j)] \quad i, j = 1, \dots, n, \quad (3.3)$$

where the concerning quantities are derived from the service placement information.

In the second step, we derive a new way to compute the prior TM among ToRs based on the correlation coefficient among ToRs and the total in/out bytes of the ToRs during a certain interval. More specifically, we first compute $x_{i \leftrightarrow j}$ as the volume of traffic between ToR_i and ToR_j by the following procedure based on the correlation coefficients.

$$\begin{aligned} x'_{i \rightarrow j} &= ToR_i^{out} \times \frac{corr_{ij}}{\sum_{k=1}^n corr_{ik}} & i, j = 1, \dots, n, \\ x'_{i \leftrightarrow j} &= x'_{i \rightarrow j} + x'_{j \rightarrow i} & i, j = 1, \dots, n. \end{aligned}$$

Due to symmetry, $x_{i \rightarrow j}$ can also be computed through ToR_j^{in} in similar ways.

As our TM estimation takes the time dimension into account (to cope with the volatile DCN traffics), one may wonder whether the correlation coefficient $[corr_{ij}]$ has to be computed for each discrete time t . In fact, as it often takes a substantial amount of time for servers to accommodate new services, the service placements will not change frequently [64]. Therefore, once $[corr_{ij}]$ is computed, they can be used for a certain period of time. Recomputing these coefficients are needed only when a new service is deployed or an existing service quits. Even under those circumstances, we only need to re-compute the coefficients among the ToRs that are affected by service changes.

3.4.2.2 The Algorithm Details

We show the pseudocode of calculating correlation enhanced prior TM in **Algorithm 2**. This algorithm takes service placement matrix S and the ToR SNMP counts as the main inputs, and it also returns the prior traffic vector among ToRs $\mathbf{x}'$. After computing the correlation coefficients in line 1, we compute the volume of traffic exchanged between

the i -th and j -th ToR using ToR_i^{out} , ToR_j^{out} and the computed correlation coefficients in lines 4–6. The complexity of the algorithm is $\mathcal{O}(n^2)$, where n is the number of racks in the datacenter. As n is generally small, the computation times are acceptable as we will see in the evaluations.

Algorithm 2: Compute Correlation Enhanced Prior TM among ToRs

Input: $S, \{ToR_i^{out} | i = 1, \dots, n\}$

Output: $\mathbf{x}'$

```

1  $[corr_{ij}] \leftarrow \text{Correlation}(S)$ 
2 for  $i = 1$  to  $n$  do
3   for  $j = i + 1$  to  $n$  do
4      $x'_{i \rightarrow j} \leftarrow ToR_i^{out} * corr_{ij} / (\sum_{1 \leq k \leq n} corr_{ik})$ 
5      $x'_{j \rightarrow i} \leftarrow ToR_j^{out} * corr_{ij} / (\sum_{1 \leq k \leq n} corr_{kj})$ 
6      $x'_{i \leftrightarrow j} \leftarrow x'_{i \rightarrow j} + x'_{j \rightarrow i}$ 
7   end
8 end
9 return  $\mathbf{x}'$ 

```

3.4.2.3 A Working Example

Figure 3.7 presents an example to illustrate how ATME-PV works. The three colors represent three services deployed in the datacenter as follows:

- service₁: server₂(rack₁), server₁₂(rack₆),
- service₂: server₄(rack₂), server₆(rack₃),
server_{13,14}(rack₇),
- service₃: server₈(rack₄), server₁₀(rack₅).

The correlation coefficients among the ToR pairs are shown in Table 3.2. More

ToR Pairs	1:2-5	1:6	1:7,8	2:3	2:4-6	2:7	2:8	3:7	4:5
Corr. Coef.	0	0.25	0	0.25	0	0.5	0	0.5	0.25

Table 3.2: Correlation Coefficients of the Working Example

specifically, ToR₂ is related to ToR₃ and ToR₇ by a coefficient of 0.25 and 0.5, respectively. So if ToR₂ totally sends out 10000 bytes during the 5 minutes interval,

the traffic sent to ToR₃ and ToR₇ should be $10000 * 0.25 / (0.25 + 0.5) = 3334$ and $10000 * 0.5 / (0.25 + 0.5) = 6667$, respectively. Similarly, we can compute the traffic volume that ToR₇ sends to ToR₂. Then we add the traffic of two directions together to get the traffic volumes between ToR₂ and ToR₇. A similar situation applies to ToR₂ and ToR₃. The estimated prior TM is then fed to the final estimation, as discussed later in Section 3.5.

Figure 3.7: Four different line styles represent four flows and three different colors represent three services.

3.5 Link Utilization Aware Network Tomography

In this section, we first propose to eliminate the links with low utilizations to turn the network tomography problem in DCNs into a more determined one. We then compute the prior volumes of traffic on the routes in DCNs and feed it to the network tomography constrained optimization problem.

3.5.1 Eliminating Lowly Utilized Links and Computing Prior Vector

This step is motivated by **Observation 2**, which states that there are plenty of lowly utilized links in DCNs. As we all know, there are many redundant routes between any two ToR switches in DCNs. Thus in the perspective of network tomography, the number of available measurements (link counts) is much smaller than the number of variables (routes). To this end, we eliminate the lowly utilized links to turn the original network tomography problem into a more determined one. More specifically, we collect the

SNMP link counts and compute the link utilization for each link. If the link utilization of a link is below a certain threshold θ , we consider the flow volumes of the routes that pass the link as zero, which effectively removes this link from the DCN topology. Note that we conduct the above process independently in each interval, thus whether the set of lowly links is changed in different intervals would not affect our solution. As a result, the number of variables in the equation system Eqn. (3.1) can be substantially reduced, resulting in a more determined tomography problem. On one hand, this threshold sets non-zero link counts to zero, possibly resulting in estimation errors. On the other hand, it removes redundant routes and mitigates the under-determinism of the tomography problem, potentially improving the estimation accuracy or running speed of algorithms. In our experiments, we shall try different values of the threshold to see the trade-off between these two sides.

Figure 3.8 is the result of reducing lowly utilized links through thresholding, hence we can estimate the traffic volumes on the remaining routes from one ToR to another. In order to compute the prior vector $\bar{\mathbf{x}}$ (we omit time slice t , so the TM at time slot t is a vector), we estimate the traffic volumes on each route by dividing the total number of bytes between two ToRs, which are also stored in $\mathbf{x}'$ and can be computed by **Algorithm 1** or **Algorithm 2**, equally on every path connecting them. The reason for this equal share is the widely used ECMP [18] in DCNs; it by default selects routing paths between two switches with equal probability on each. The computed prior vector $\bar{\mathbf{x}}$ will give us a good start in solving a quadratic programming problem to determine the final estimation.

Figure 3.8: After reducing the lowly utilized links in Figure 3.7

3.5.2 Combining Prior TM with Network Tomography constraints

Here we provide more details on the computation involved in getting the final estimation, which is also a **QuadProgram**. Basically, we want to obtain $\mathbf{x}$ that is as close as possible to the prior $\bar{\mathbf{x}}$ but also satisfies the tomographic constraints. This problem can be formulated as follows:

$$\text{Minimize} \quad \|\mathbf{x} - \bar{\mathbf{x}}\| + \|A\mathbf{x} - \mathbf{y}\| \quad (3.4)$$

where $\|\mathbf{x} - \bar{\mathbf{x}}\|$ is the distance between the final solution and the prior, $\|A\mathbf{x} - \mathbf{y}\|$ is the deviation from the tomographic constraints, and $\|\cdot\|$ is L_2 -norm of a vector.

To tackle this problem, we first compute the deviation of prior values $\tilde{\mathbf{y}} = \mathbf{y} - A\bar{\mathbf{x}}$, then we solve the following constrained least square problem in Eqn.(3.5) to obtain the $\tilde{\mathbf{x}}$ as the adjustments to $\bar{\mathbf{x}}$ for offsetting the deviation $\tilde{\mathbf{y}}$.

$$\begin{aligned} \text{Minimize} \quad & \|A\tilde{\mathbf{x}} - \tilde{\mathbf{y}}\| \\ \text{s.t.} \quad & \beta\tilde{\mathbf{x}} \geq -\bar{\mathbf{x}} \end{aligned} \quad (3.5)$$

We use a tunable parameter β , $0 \leq \beta \leq 1$ to make the tradeoff between the similarity to the prior solution and the precise fit to the link loads. The constraint is meant to guarantee a non-negative final estimation $\hat{\mathbf{x}}$. Finally, $\hat{\mathbf{x}}$ is obtained by making a tradeoff between the prior and the tomographic constraint as $\hat{\mathbf{x}} = \bar{\mathbf{x}} + \beta\tilde{\mathbf{x}}$. According to our experience, we take $\beta = 0.8$ to give a slightly more bias towards the prior.

3.5.3 The Algorithm Details

We summarize the link utilization aware network tomography in **Algorithm 3**. It takes routing matrix A , the vector of link capacities $\mathbf{b}$, link counts vector $\mathbf{y}$, threshold of link utilization θ and the prior TM among ToRs $\mathbf{x}'$ as the main inputs. Its output is the vector of final estimations of the traffic volume on each path among ToRs $\hat{\mathbf{x}}$. In particular, we first check each of the links to see whether their utilizations are below θ (lines 2). If so, we remove the paths which contain such links from the path set $\mathcal{P}_{ij}$ (includes all paths between the i -th ToR and the j -th ToR), and adjust the matrix A , vector $\mathbf{x}$ and $\mathbf{y}$ by removing the corresponding rows and components (line 5). Here, the utilization of link k is computed by y_k/b_k , where y_k is the load on link k , and b_k is the link's bandwidth. Then for each of the ToR pairs (i, j) , and the loads on the

Algorithm 3: Link Utilization-aware Network Tomography

Input: $A, \mathbf{b}, \mathbf{y}, \theta, \mathbf{x}'$
Output: $\hat{\mathbf{x}}$

```

1 for  $k = 1$  to  $m$  do
2   if  $y_k/b_k \leq \theta$  then
3     forall  $r \in \mathcal{P}_{ij}$  do
4       if  $r$  contains  $l_k$  then
5          $\mathcal{P}_{ij} \leftarrow \mathcal{P}_{ij} - \{r\}$ ; Adjust  $A, \mathbf{x}$  and  $\mathbf{y}$ 
6       end
7     end
8   end
9 end
10 for  $i = 1$  to  $n$  do
11   for  $j = i + 1$  to  $n$  do
12     forall  $r \in \mathcal{P}_{ij}$  do  $\bar{x}_r \leftarrow x'_{i \leftrightarrow j} / |\mathcal{P}_{ij}|$ ;
13   end
14 end
15  $\hat{\mathbf{x}} \leftarrow \text{QuadProgram}(A, \bar{\mathbf{x}}, \mathbf{y})$ 
16 return  $\hat{\mathbf{x}}$ 

```

remaining paths in $\mathcal{P}_{ij}$ are calculated by averaging the total traffic across the two ToRs $x'_{i \leftrightarrow j}$ (line 12). Finally, the algorithm applies a quadratic programming to refine $\bar{\mathbf{x}}$ to obtain $\mathbf{x}$ subject to the constraints posed by $\mathbf{y}$ and A (line 15).

Obviously, The dominant running time of the algorithm is spent on $\text{QuadProgram}(A, \bar{\mathbf{x}}, \mathbf{y})$, whose main component Eqn. (3.5) is equivalent to a *non-negative least squares* (NNLS) problem. The complexity of solving this NNLS is $\mathcal{O}(m^2 + p^2)$, but can be reduced to $\mathcal{O}(p \log m)$ though parallel computing in a multi-core system [67].

3.6 Evaluation

In this section, we evaluate ATME-PB and ATME-PV with both hardware testbed and extensive simulations.

3.6.1 Experiment Settings

We implement ATME-PB and ATME-PV together with two representative TM inference algorithms:

- Tomogravity [31] is known as a classical TM estimation algorithm that performs well in IP networks. In contrast to ATME, it assumes traffic flows in the networks follow the gravity traffic model, and traffic exchanged by two ends is proportional to the total traffic on the two ends.
- Sparsity Regularized Matrix Factorization (SRMF for short) [32] is a state-of-art traffic estimation algorithm. It leverages the spatio-temporal structure of traffic flows, and utilizes the compressive sensing method to infer TM by rank minimization.

These algorithms serve as benchmarks to evaluate ATME-PB and ATME-PV under different network settings.

We quantify the performance of the three algorithms using four metrics: *Relative Error* (RE), *Root Mean Squared Error* (RMSE), *Root Mean Squared Relative Error* (RMSRE) and the computing time. RE is defined for individual elements as:

$$\text{RE}_i = |x_i - \hat{x}_i|/x_i, \quad (3.6)$$

where x_i denotes the true TM element and $\hat{x}_i$ is the corresponding estimated value. RMSE and RMSRE are metrics to evaluate the overall estimation errors:

$$\text{RMSE} = \sqrt{\frac{1}{n_x} \sum_{i=1}^{n_x} (x_i - \hat{x}_i)^2}, \quad (3.7)$$

$$\text{RMSRE}(\tau) = \sqrt{\frac{1}{n_\tau} \sum_{i=1, x_i > \tau}^{n_x} \left(\frac{x_i - \hat{x}_i}{x_i} \right)^2}. \quad (3.8)$$

Similar to [31], we use τ to pick up the relative large traffic flows since larger flows are more important for engineering DCNs. n_x is the number of elements in the ground truth $\mathbf{x}$ and n_τ is the number of elements $x_i > \tau$.

3.6.2 Testbed Evaluation of ATME-PB

3.6.2.1 Testbed Setup

We use a testbed with 10 switches and about 300 servers as shown in Figure 3.9 for our experiments, and the architecture for this testbed is a conventional tree similar to the one in Figure 3.1. The testbed hosts a variety of services and part of which has been shown in Figure 3.6(a). We gather the resource provisioning information and SNMP link counts for all switches. We also record the flows exchanged among servers by using Linux `iptables` in each server (not a scalable approach) to form the ground truth. The data are all collected every 5 minutes. The capacities of links are all 1Gbps.

Figure 3.9: Hardware testbed with 10 racks and more than 300 servers.

Figure 3.10: The CDF of RE and RMSRE of ATME-PB and two baselines on testbed.

Figure 3.11: The CDF of RE and RMSRE of ATME-PV and two baselines on testbed.

3.6.2.2 Testbed Results

Figure 3.10(a) depicts the relative errors of the three algorithms. As we can see in this figure, our algorithm can accurately infer about 80% of TM elements, while the two other competitive algorithms can only infer less than 60% of them. We can also clearly see that about 99% of percent of inference results of our algorithm has the relative error less than 0.5. An intuitive explanation for this is that our algorithm can clearly separate the traffic into many groups by user in the multi-tenant cloud datacenter. Consequently, it is closer to the real traffic patterns and is more suitable for the assumptions of gravity model after clustering. Therefore, our algorithm can get a more accurate prior TM and final estimated TM than the state-of-art algorithms.

We then present the RMSRE of the algorithms in Figure 3.10(b). Clearly we can see that our algorithm has the lowest RMSRE as the flow size increases. When the flow size is less than 4000Mbit (500MBytes), the RMSRE is stable with the flow size, and it starts to decrease after the flow size is greater than 500MBytes, which demonstrates that our algorithm performs even better when handling elephant flows in the network.

3.6.3 Testbed Evaluation of ATME-PV

3.6.3.1 Testbed Setup

We use the same testbed as stated in Section 3.6.2, and we also use the Linux `iptables` in each server to collect the real TM as the ground truth. Besides all the SNMP link counts in the servers and switches, we also gather the service placement information in the controller nodes of the datacenter. All the data are collected every 5 minutes.

3.6.3.2 Testbed Results

Figure 3.11(a) plots the CDF of REs of the three algorithms. Clearly, ATME-PV performs significantly better than the other two: it can accurately estimate the volumes of more than 78% of traffic flows. As the TM of our DCN may not be of low rank, SRMF performs similarly to tomogravity.

We then study these algorithms with respect to the RMSREs in Figure 3.11(b). It is natural to see that the RMSREs of all three algorithms are non-increasing with τ , because estimation algorithms are all subject to noise for the light traffic flows, but they normally performs better for heavy traffic flows. However, ATME-PV still achieves the lowest RMSRE for all values of τ among the three. As our experiments with real DCN traffic are confined by the scale of our testbed, we conduct extensive simulations with larger DCNs in *ns-3*.

Figure 3.12: The CDF of RE (a), the RMSRE (b), and the RMSE (c) of ATME-PB and two baselines for estimating TM under tree architecture.

Figure 3.13: The CDF of RE (a), the RMSRE (b), and the RMSE (c) of ATME-PB and two baselines for estimating TM under fat-tree architecture.

3.6.4 Simulation Evaluation of ATME-PB

3.6.4.1 Simulation Setup

We adopt both the conventional datacenter architecture [1] and fat-tree architecture [15] in our simulations. For the conventional tree, there are 32 ToR switches, 16 aggregation switches, and 3 core switches; for fat-tree, we use $k = 8$ fat-tree with the same number of ToR switches as the conventional tree, but with 32 aggregation switches, 16 core switches. The link capacities are all set to be 1Gbps. We could not conduct simulations on BCube [20] because it does not arrange servers into racks. It would be an interesting problem to study how to extend our proposal for estimating the TM for servers in BCube.

We take the simulated datacenter as a multi-tenant environments, so there are many users in the datacenter and all the users are sending or receiving traffic in their own

VM/servers independently. In our simulations, we record the resource provisioning information, which are used to enhance the network tomography results.

We install both on-off and bulk-send applications in *ns-3*. The packet size is set to 1400 bytes (varying the packet size has little effect on the performance of our scheme in our experiments), and the flow sizes are randomly generated but still follows the characteristics of real DCNs [3, 8, 23]. For instance, 10% of the flows contributes to about 90% of the total traffic in a DCN [9, 16]. We use TCP flows in our simulations [68], and apply the widely used ECMP [18] as the routing protocol.

We record the total number of bytes and packets that enter and leave every port of each switch in the network every 5 minutes. We also record the total bytes and packets of flows on each route in the corresponding time periods as the ground truth. For every setting we run simulations for 10 times.

To evaluate the computing time, we measure the time period starting from when we input the topologies and link counts to the algorithm until the time when all TM elements are returned. All three algorithms are implemented by Matlab (R2012b) on 6-core Intel Xeon CPU @3.20GHz, with 16GB of memory and the Windows 7 64-bit OS.

3.6.4.2 Simulation Results

We set θ to be 0.001. In Figure 3.12(a), we plot the CDF of relative errors of the three algorithms under conventional tree architecture. Our algorithm has the lowest relative errors when compared with the other two state-of-art algorithms. More specifically, about 80% of the relative errors are less than 0.5. While for the other two algorithms, about 80% of the relative errors is bigger than 0.5. We draw RMSREs of the three algorithms under different threshold of flow size in Figure 3.12(b). In this figure, all the three algorithms show declining trends with the increasing size of flows. However, our algorithm still performs the best among the three algorithms. The reason for these two figures is that no matter how the traffic changes in datacenter, our algorithm can accurately identify the communication groups by the easily collected resource provisioning information. When tomography fails to get a good prior TM, a bad final estimation would be obtained. For SRMF, it may get the TMs, which are much more sparse than the ground truth due to the rank minimization approach. We also present how the RMSEs change with the threshold θ of link utilization in Figure 3.12(c). As we can see

that, the curve is stable when θ is smaller than 0.10 and becomes fluctuant afterwards. As removing the lowly utilized links can decrease the running time of the algorithm, it is a good trade off between accuracy and running speed if we set the θ properly (less than 0.10 in this case).

We also set θ to be 0.001 in the fat-tree case. We draw the CDF of relative errors of the three algorithms under fat-tree architecture in Figure 3.13(a). Here our algorithm still has the best performance among the three algorithms. About 90% of the relative errors are smaller than 0.5. The corresponding percentage for the other two algorithms is about 40%. In Figure 3.13(b), we can see that the RMSRE of our algorithm decreases from 0.4 and approximates 0 with the increase of the size of flows. Finally, we also depict how RMSE changes with θ in Figure 3.13(c). In this figure, the RMSE is stable when θ is lower than 0.1 and increases slowly with θ after that, which also demonstrates that removing some lowly utilized links will not decrease the accuracy of our algorithm. While we will see that it can decrease the running time instead if we set θ properly, as shown in Table 3.3.

Switches	Links	Routes	Computing Time			
			ATME-PB		Tomo-gravity	SRMF
			$\theta = 0$	$\theta = 0.1$		
80	256	7360	4.90	3.60	4.28	251.12
125	500	28625	48.08	40.10	45.32	-

Table 3.3: The Computing Time (seconds) of ATME-PB, Tomogravity and SRMF under Different Scales of DCNs (Fat-tree)

Table 3.3 lists the computing time of the three algorithms under fat-tree architecture. Obviously, ATME-PB also performs faster than both tomogravity and SRMF with proper threshold settings. SRMF often cannot deliver a result for several hours when the topology is big. If we slightly increase θ , we may further reduce the computing time, as shown in Table 3.3. In other words, our proposal, ATME-PB, can run even faster without sacrificing accuracy by setting the threshold θ properly as we can see in the table and Figure 3.13(c).

Figure 3.14: The CDF of RE (a), the RMSRE (b), and the RMSE (c) of ATME-PV and two baselines for estimating TM under tree architecture.

3.6.5 Simulation Evaluations of ATME-PV

3.6.5.1 Simulation Setup

The simulation setup is almost the same with the setup in Section 3.6.4: we simulate datacenters with conventional tree and fat-tree architecture by *ns-3*. The differences are that we randomly deploy services in the DCN and record the service placement information.

3.6.5.2 Simulation Results

Figure 3.14(a) compares the CDF of REs of the three algorithms under conventional tree architecture and we set $\theta = 0.001$. We can clearly see that ATME-PV has much smaller relative errors. The advantage of ATME-PV over the other two algorithms stems from the fact that ATME-PV can clearly find out the ToR pairs that do not

Figure 3.15: The CDF of RE (a), the RMSRE (b), and the RMSE (c) of ATME-PV and two baselines for estimating TM under fat-tree architecture.

communicate with each other. Tomogravity has the worst performance because it gives each ToR pair a communication traffic whenever one of them has “out” traffic and the other has “in” traffic, thus introducing non-existing positive TM entries. SRMF obtains the TM by rank minimization, so it performs better than tomogravity when the traffic in DCNs does lead to low ranked TM. The worse performance of SRMF (compared with ATME-PV) may be due to its over-fitting of the sparsity in eigenvalues, according to [8].

We then study the RMSREs of the three algorithms under different τ in Figure 3.14(b). Again, ATME-PV exhibits the lowest RMSRE and a (expectable) reducing trend with the increase of τ , while the other two remain almost constant with τ . In Figure 3.14(c), we then study how the RMSE changes with the threshold θ of link utilizations. As we can see in this figure, when we gradually increase the threshold, RMSE does slightly decrease until the sweet point $\theta = 0.12$. While the improvement

on accuracy may be minor, the computing time can be substantially reduced as we will show later.

Figure 3.15 evaluates the same metrics as Figure 3.14 but under fat-tree architecture, which has even more redundant routes. We set $\theta = 0.001$. Since TM in fat-tree DCNs is far more sparse, the errors are evaluated only against the non-zero elements in TM. In general, ATME-PV retains its superiority over others in both RE and RMSRE. The effect of θ becomes more interesting in Figure 3.15(c) (compared with Figure 3.14(c)); it clearly shows a “valley” in the curve and a sweet point around $\theta = 0.03$. This is indeed the trade-off effect of θ mentioned in Section 3.5.1: it trades the estimation accuracy of light flows for that of heavy flows. More respectively, on one hand, eliminating lowly utilized links would incur errors for the flows that pass through those links, which affects light flows. On the other hand, it would make the network tomography problem to be more determined, which has the potential to improve the overall accuracy of estimations for the heavy flows.

Switches	Links	Routes	Computing Time			
			ATME-PV		Tomogravity	SRMF
			$\theta = 0.001$	$\theta = 0.01$		
51	112	5472	0.54	0.51	2.54	1168.22
102	320	46272	8.12	7.81	73.59	-

Table 3.4: The Computing Time (seconds) of ATME-PV, Tomogravity and SRMF under Different Scales of DCNs (Tree)

Table 3.4 lists the computing time of the three algorithms under conventional tree architecture. Obviously, ATME-PV performs much faster than both tomogravity and SRMF. While both ATME-PV and tomogravity have their computing time grow quadratically with the scale of the DCNs, SRMF often cannot deliver a result within a reasonable time scale. In fact, if we slightly increase θ , we may further reduce the computing time, as shown in Table 3.4. In summary, our algorithm has both a higher accuracy and faster running speed compared to the two state-of-art algorithms.

3.7 Summary

To meet the increasing demands for detailed traffic characteristics in DCNs, we make the first step towards estimating the TM among ToRs in both public and private DCNs,

relying only on the easily accessible SNMP counters and the datacenter operational information. We pioneer in applying tomographic methods to DCNs by overcoming the barriers of solving the ill-posed linear system in DCNs for TM estimation. We first obtain two major observations on the rich statistics of traffic data in DCNs. The first observation reveals that the TMs among ToRs of DCNs are extremely sparse. The other observation demonstrates that eliminating part of lowly utilized links can potentially increase both overall accuracy and the efficiency of TM estimation. Based on these two observations, we develop a new TM estimation framework ATME, which is applicable to most prevailing DCN architectures without any additional infrastructure supports. We validate ATME with both hardware testbed and simulations, and the results show that ATME outperforms the other two well-known TM estimation methods on both accuracy and efficiency. Particularly, ATME can accurately estimate more than 80% traffic flows in most cases with far less computing time.

Chapter 4

Scheduling Tasks for Big Data Processing Jobs Across Geo-Distributed Datacenters

Typically called big data processing, processing large volumes of data from geographically distributed regions with machine learning algorithms has emerged as an important analytical tool for governments and multinational corporations. The traditional wisdom calls for the collection of all the data across the world to a central datacenter location, to be processed using data-parallel applications. This is neither efficient nor practical as the volume of data grows exponentially. Rather than transferring data, we believe that computation tasks should be scheduled where the data is, while data should be processed with a minimum amount of transfers across datacenters. In this chapter, we design and implement *Flutter*, a new task scheduling algorithm that improves the completion times of big data processing jobs across geographically distributed datacenters. To cater to the specific characteristics of data-parallel applications, we first formulate our problem as a lexicographical min-max integer linear programming (ILP) problem, and then transform it into a nonlinear program with a separable convex objective function and a totally unimodular constraint matrix, which can be further transformed into a linear programming (LP) problem and thus can be solved using a standard linear programming solver efficiently in an online fashion. Our implementation of *Flutter* is based on Apache Spark, a modern framework popular for big data processing. Our experimental results have shown that we can reduce the job completion time by up to

25%, and the amount of traffic transferred among different datacenters by up to 75%.

4.1 Introduction

It has now become commonly accepted that the volume of data — from end users, sensors, and algorithms alike — have been growing exponentially, and mostly stored in geographically distributed datacenters around the world. *Big data processing* refers to applications that apply machine learning algorithms to process such large volumes of data, typically supported by modern data-parallel frameworks such as Spark. Needless to say, big data processing has become routine in governments and multinational corporations, especially those in the business of social media and Internet advertising.

To process large volumes of data that are geographically distributed, we will traditionally need to transfer all the data to be processed to a single datacenter, so that they can be processed in a centralized fashion. However, at times, such traditional wisdom may not be practically feasible. First, it may not be practical to move user data across country boundaries, due to legal reasons or privacy concerns [11]. Second, the cost, in terms of both bandwidth and time, to move large volumes of data across geo-distributed datacenters may become prohibitive as the volume of data grows exponentially.

It has been pointed out that [11, 12, 13], rather than transferring data across datacenters, it may be a better design to move computation tasks to where the data is, so that data can be processed locally within the same datacenter. Of course, the intermediate results after such processing may still need to be transferred across datacenters, but they are typically much smaller in size, significantly reducing the cost of data transfers. An example showing the benefits of processing big data over geo-distributed datacenters is shown in Figure 4.1. The fundamental objective, in general, is to minimize the job completion times in big data processing applications, by placing the tasks at their respective best possible datacenters. Yet, previous works (*e.g.*, [12]) were designed with assumptions that were often unrealistic — such as bottlenecks do not occur on inter-datacenter links.

Intuitively, it may be a step towards the right direction to design an offline optimal task scheduling algorithm, so that the job completion times are globally minimized. However, such offline optimization inevitably relies upon *a priori* knowledge of task execution times and transfer times of intermediate results, neither of which is readily

Figure 4.1: Processing data locally by moving computation tasks: an illustrating example.

available without complex prediction algorithms. Even if such knowledge were available, a big data processing job in Spark may involve a directed acyclic graph (DAG) with hundreds of tasks; and optimal solutions for scheduling such a DAG is NP-Complete in general [69].

In this chapter, we design and implement *Flutter*, a new system to schedule tasks across datacenters over the wide area. Our primary focus when designing *Flutter* is on practicality and real-world implementation, rather than on the optimality of our results. To be practical, *Flutter* is first and foremost designed as an *online* scheduling algorithm, making adjustments on-the-fly based on the current job progress. *Flutter* is also designed to be *stage-aware*: it minimizes the completion time of each stage in a job, which corresponds to the slowest of the completion times of the constituent tasks in the stage.

Practicality also implies that our algorithm in *Flutter* would need to be efficient at runtime. Our problem of stage-aware online scheduling can be formulated as a lexicographical min-max integer linear programming (ILP) problem. A highlight of this chapter is that, after transforming the problem into a nonlinear program, we show that it has a separable convex objective function and a totally unimodular constraint matrix, which can then be solved using a standard linear programming solver efficiently, and in an online fashion.

To demonstrate that it is amenable to practical implementations, we have imple-

mented *Flutter* based on Apache Spark, a modern framework popular for big data processing. Our experimental results on a production wide-area network with geo-distributed servers have shown that we can reduce the job completion time by up to 25%, and the amount of traffic transferred among different datacenters by up to 75%.

4.2 Flutter: Motivation and Problem Formulation

To motivate our work, we begin with a real-world experiment, with *Virtual Machines* (VMs) initiated and distributed in four representative regions in Amazon EC2: EU (Frankfurt), US East (N. Virginia), US West (Oregon), and Asia Pacific (Singapore). All the VM instances we used are `m3.xlarge`, with 4 cores and 15 GB of main memory each. To illustrate the actual available capacities on inter-datacenter links, we have measured the bandwidth available across datacenters using the `iperf` utility, and our results are shown in Table 4.1.

From this table, we can make two observations with convincing evidence. On one hand, when VMs in the same datacenter communicate with each other across the intra-datacenter network, the available bandwidth is consistently high, at around 1 Gbps. This is sufficient for typical Spark-based data-parallel applications [70]. On the other hand, bandwidth across datacenters is an order of magnitude lower, and varies significantly for different inter-datacenter links. For example, the link with the highest bandwidth is 175 Mbps, while the lowest is only 49 Mbps.

Our observations have clearly implied that transfer times of intermediate results across datacenters can easily become the bottleneck when it comes to job completion times, when we run the same data-parallel application across different datacenters. Scheduling tasks carefully to the best possible datacenters is, therefore, important to utilize available inter-datacenter bandwidth better; and more so when the inter-datacenter bandwidth is lower and more divergent. *Flutter* is first and foremost designed to be *network-aware*, in that tasks can be scheduled across geo-distributed datacenters with the awareness of available inter-datacenter bandwidth.

To formulate the problem that we wish to solve with the design of *Flutter*, we revisit the current task scheduling disciplines in existing data-parallel frameworks that support big data processing, taking Spark [14] as an example. In Spark, a job can be represented by a Directed Acyclic Graph (DAG) $G = (\mathcal{V}, \mathcal{E})$. Each node $v \in \mathcal{V}$

	EU	US-East	US-West	Singapore
EU	946Mbps	136Mbps	76.3Mbps	49.3Mbps
US-East	-	1.01Gbps	175Mbps	52.6Mbps
US-West	-	-	945Mbps	76.9Mbps
Singapore	-	-	-	945Mbps

Table 4.1: Available bandwidths across geographically distributed datacenters.

represents a task; each directed edge $e \in \mathcal{E}$ indicates a precedence constraint, and the length of e represents the transfer time of intermediate results from the source node to the destination node of e .

Scheduling all the tasks in the DAG to a number of worker nodes — while minimizing the completion time of the job — is known as a NP-Complete problem in general [69], and is neither efficient nor practical. Rather than scheduling all the tasks together, Spark schedules ready tasks stage by stage in an online fashion. As it is a much more practical way of designing a task scheduler, *Flutter* follows suit and only schedules the tasks within the same stage to geo-distributed datacenters, rather than considering all the ready tasks in the DAG. Here we denote the set of tasks in a stage by $\mathcal{N} = \{1 \dots n\}$, and the set of datacenters by $\mathcal{D} = \{1 \dots d\}$.

There is, however, one more complication when tasks within the same stage are to be scheduled. The complication comes from the fact that the completion time of a stage in data-parallel jobs is determined by the completion time of the *slowest* task in that stage. Without awareness of the stage that a task belongs to, it may be scheduled to a datacenter with a much longer transfer time to receive all the intermediate results needed (due to capacity limitations on inter-datacenter links), slowing down not only the stage it belongs to, but the entire job as well.

More formally, *Flutter* should be designed to solve a *network-aware* and *stage-aware* online task scheduling problem, formulated as a lexicographical min-max integer linear

programming (ILP) problem as follows:

$$\text{lexmin}_X \quad \max_{i,j} (x_{ij} \cdot (c_{ij} + e_{ij})) \quad (4.1)$$

$$\text{s.t.} \quad \sum_{j=1}^d x_{ij} = 1, \quad \forall i \in \mathcal{N} \quad (4.2)$$

$$\sum_{i=1}^n x_{ij} \leq f_j, \quad \forall j \in \mathcal{D} \quad (4.3)$$

$$c_{ij} = \max_{k \in s_i} (m_{d_k j} / b_{d_k j}), \quad \forall i \in \mathcal{N}, \forall j \in \mathcal{D} \quad (4.4)$$

$$x_{ij} \in \{0, 1\}, \quad \forall i \in \mathcal{N}, \forall j \in \mathcal{D} \quad (4.5)$$

In our objective function (4.1), $x_{ij} = 1$ indicates the assignment of the i -th task to j -th datacenter; otherwise $x_{ij} = 0$. c_{ij} is the transfer time to receive all the intermediate results, computed in Eq. (4.4). e_{ij} denotes the execution time of the i -th task in the j -th datacenter. Our objective is to minimize the maximum task completion time within a stage, including both the network transfer time and the task execution time.

To achieve this objective, there are four constraints that we will need to satisfy. The first constraint in Eq. (4.2) implies that each task should be scheduled to only one datacenter. The second constraint, Eq. (4.3), implies that the number of tasks assigned to the j -th datacenter should not exceed the maximum number of tasks f_j that can be scheduled on the existing VMs on that datacenter. Though it is indeed conceivable to launch new VMs on-demand, it takes a few minutes in reality to initiate and launch a new VM, making it far from practical. The total number of tasks that can be scheduled depends on the number of VMs that have already been initiated, which is limited due to budgetary constraints.

The third constraint, Eq. (4.4), is to compute the transfer time of the i -th task on j -th datacenter, where s_i and d_k represent the number of inputs for the i -th task and the index of the datacenter that has the k -th input, respectively. For example, let $m_{d_k j}$ denote the amount of bytes that need to be transferred from the d_k -th datacenter to the j -th datacenter if the i -th task is scheduled to the j -th datacenter. If $d_k = j$, then $m_{d_k j} = 0$. We let b_{uv} to denote the bandwidth between the u -th datacenter and the v -th datacenter, and assume that the network bandwidth $B_{d \times d} = \{b_{uv} \mid u, v = 1 \dots d\}$ across all the datacenters can be measured, and is stable over a few minutes. We can then compute the maximum transfer times for each possible way of scheduling the i -th task. The last constraint indicates that x_{ij} is a binary variable.

4.3 Network-aware Task Scheduling across Geo-Distributed Datacenters

Given the formal problem formulation of our network-aware task scheduling across geo-distributed datacenters, we now study how we solve the proposed ILP problem efficiently, which is the key for the practicality of *Flutter* in the real data processing systems. In this section, we first propose to transform the lexicographical min-max integer problem in our original formulation into a special class of nonlinear programming problem. We then further transform this special class of nonlinear programming problem into a *linear programming* (LP) problem that can be solved efficiently with standard linear programming solvers.

4.3.1 Transform into a Nonlinear Programming Problem

The special class of nonlinear programs that can be transformed into a LP have two characteristics [71], a separable convex objective function and a totally unimodular constraint matrix. We will show how we transform our original formulation to meet these two conditions.

4.3.1.1 Separable Convex Objective Function

If a function can be represented as a summation of multiple convex functions with a single variable, then it is separable convex. To make this transformation, we first define the lexicographical order. Let $\mathbf{p}$ and $\mathbf{q}$ represent two integer vectors of length k . We define $\vec{\mathbf{p}}$ and $\vec{\mathbf{q}}$ as the sorted $\mathbf{p}$ and $\mathbf{q}$ with non-increasing order, respectively. If $\mathbf{p}$ is lexicographically less than $\mathbf{q}$, represented by $\mathbf{p} \prec \mathbf{q}$, it means that the first non-zero item of $\vec{\mathbf{p}} - \vec{\mathbf{q}}$ is negative. Then if $\mathbf{p}$ is lexicographically no greater than $\mathbf{q}$, denoted as $\mathbf{p} \preceq \mathbf{q}$, it is equivalent to $\mathbf{p} \prec \mathbf{q}$ or $\vec{\mathbf{p}} = \vec{\mathbf{q}}$.

Our objective function is to find a vector that is lexicographically minimal over all the feasible vectors with its components rearranged in a non-increasing order. In our problem, if $\mathbf{p}$ is lexicographically no greater than $\mathbf{q}$, then vector $\mathbf{p}$ is a better solution for our lexicographical min-max problem. However, directly finding the lexicographical minimal vector is not a easy task, we find out that we can use a summation of exponents to preserve the lexicographical order among vectors. Consider the convex function

$g : \mathbb{Z}^k \rightarrow R$ that has the form of

$$g(\boldsymbol{\lambda}) = \sum_{i=1}^k k^{\lambda_i},$$

where $\boldsymbol{\lambda} = \{\lambda_i \mid i = 1 \dots k\}$ is an integer vector with length k . We prove that we can preserve the lexicographical order of vectors through $g : \mathbb{Z}^k \rightarrow R$ by the following lemma¹.

Lemma 1. For $\mathbf{p}, \mathbf{q} \in \mathbb{Z}^k$, $\mathbf{p} \preceq \mathbf{q} \iff g(\mathbf{p}) \leq g(\mathbf{q})$.

Proof. We first prove that $\mathbf{p} \prec \mathbf{q} \implies g(\mathbf{p}) < g(\mathbf{q})$. We assume that the index of the first positive element of $\vec{\mathbf{q}} - \vec{\mathbf{p}}$ is r . As both vectors only have integral elements, $\vec{q}_r > \vec{p}_r$ implies $\vec{q}_r \geq \vec{p}_r + 1$. Then we have:

$$g(\mathbf{q}) - g(\mathbf{p}) = g(\vec{\mathbf{q}}) - g(\vec{\mathbf{p}}) \tag{4.6}$$

$$= \sum_{i=1}^k k^{\vec{q}_i} - \sum_{i=1}^k k^{\vec{p}_i} \tag{4.7}$$

$$= \sum_{i=r}^k k^{\vec{q}_i} - \sum_{i=r}^k k^{\vec{p}_i} \tag{4.8}$$

$$> \sum_{i=r}^k k^{\vec{q}_i} - k \times k^{\vec{p}_r} \tag{4.9}$$

$$= (k^{\vec{q}_r} - k^{\vec{p}_r+1}) + \sum_{i=r+1}^k k^{\vec{q}_i} \tag{4.10}$$

$$> 0 \tag{4.11}$$

Hence the first part is proved.

We then show $g(\mathbf{p}) < g(\mathbf{q}) \implies \mathbf{p} \prec \mathbf{q}$ and we assume r is the index of first

¹Since scaling the coefficients of x_{ij} would not change the optimal solution, we can always make the coefficients to be integers.

non-zero element in $\vec{q} - \vec{p}$, then $\vec{p}_i = \vec{q}_i$ for all $i < r$.

$$g(\mathbf{q}) - g(\mathbf{p}) = g(\vec{q}) - g(\vec{p}) \quad (4.12)$$

$$= \sum_{i=1}^k k^{\vec{q}_i} - \sum_{i=1}^k k^{\vec{p}_i} \quad (4.13)$$

$$< \sum_{i=1}^{r-1} k^{\vec{q}_i} + (k+1-r) \times k^{\vec{q}_r} \quad (4.14)$$

$$- \sum_{i=1}^{r-1} k^{\vec{p}_i} - k^{\vec{p}_r} \quad (4.15)$$

$$= (k+1-r) \times k^{\vec{q}_r} - k^{\vec{p}_r} \quad (4.16)$$

Therefore if $g(\mathbf{q}) - g(\mathbf{p}) > 0$, then we have $(k+1-r) \times k^{\vec{q}_r} - k^{\vec{p}_r} > 0$. For $r = 1$, it implies $\vec{q}_r + 1 > \vec{p}_r$. If $\vec{q}_r < \vec{p}_r$, the previous inequation would not hold. $\vec{q}_r$ also does not equal $\vec{p}_r$ as r is the index of the first non-zero item in $\vec{q} - \vec{p}$. We then have $\vec{q}_r > \vec{p}_r$. For $r > 1$, $(k+1-r) \times k^{\vec{q}_r} - k^{\vec{p}_r} > 0$ implies $\log_k(k+1-r) + \vec{q}_r > \vec{p}_r$. Because $r > 1$, $\log_k(k+1-r)$ is less than 1 and $\vec{q}_r \neq \vec{p}_r$ because r is the index of first non-zero item in $\vec{q} - \vec{p}$. Thus we can also have $\vec{q}_r > \vec{p}_r$ when $r > 1$. In sum, $\vec{q}_r > \vec{p}_r$ for all $r \geq 1$. As a result, it can be concluded that $\mathbf{p} \prec \mathbf{q}$.

Regarding the equations, if $\vec{p} = \vec{q}$, it is straightforward to see that $g(\mathbf{q}) = g(\mathbf{p})$. Now if $g(\mathbf{q}) = g(\mathbf{p})$, let us prove whether we have $\vec{p} = \vec{q}$. Without loss of generality, we can assume that $\mathbf{p} \prec \mathbf{q}$ when $g(\mathbf{q}) = g(\mathbf{p})$. While if $\mathbf{p} \prec \mathbf{q}$, then we have $g(\mathbf{p}) < g(\mathbf{q})$ based on previous proofs, which contradicts to the assumption. Thus if $g(\mathbf{q}) = g(\mathbf{p})$, we also have $\vec{p} = \vec{q}$. $\square$

Let $\mathbf{h}(X)$ denote the vector in the objective function of our problem in Eq. (4.1). Then our problem can be denoted by $\text{lexmin}_X (\max \mathbf{h}(X))$. Based on **Lemma 1**, the objective function of our problem can be further replaced by $\min g(\mathbf{h}(X))$, which is

$$\min \sum_i^n \sum_j^d k^{x_{ij} \cdot (c_{ij} + e_{ij})}, \quad (4.17)$$

where k equals nd , which is the length of vectors in the solution space of the problem in our formulation.

We can clearly see that each term of summation in Eq. (4.17) is an exponential function, which is convex. Therefore this new objective function consists of a separable convex objective function. Now let us see whether the coefficients in the constraints of our formulation form a totally unimodular matrix.

4.3.1.2 Totally Unimodular Constraint Matrix

A totally unimodular matrix is an important concept as it can quickly determine whether a LP is integral, which means that the LP would only have integral optimum if it has any. For instance, if a problem has the form of $\{\min cx \mid AX \leq \mathbf{b}, x \geq 0\}$, where A is a totally unimodular matrix and $\mathbf{b}$ is an integral vector, then the optimal solutions for this problem must be integral. The reason is that in this case, the feasible region $\{x \mid AX \leq \mathbf{b}, x \geq 0\}$ is an integral polyhedron, which has only integral extreme points. Hence if we can prove that the coefficients in the constraints of our formulation form a totally unimodular matrix, then our problem would only have integral solutions. We prove that the constraint matrix in our problem formulation form a totally unimodular matrix by the following lemma.

Lemma 2. *The coefficients of the constraints (4.2) and (4.3) form a totally unimodular matrix.*

Proof. A totally unimodular matrix is a $m \times r$ matrix $A = \{a_{ij} \mid i = 1 \dots m, j = 1 \dots r\}$ that meets the following two conditions. First, all of its elements must be selected from $\{-1, 0, 1\}$. It is straightforward to see that all the elements in the coefficients of our constraints are 0 or 1, so it meets the first condition. The second condition is that for any subset of rows $J \in \{1 \dots m\}$, it can be separated into two sets J_1, J_2 such that $\|\sum_{i \in J_1} a_{ij} - \sum_{i \in J_2} a_{ij}\| \leq 1$. In our formulation, we can take the variable $X = \{x_{ij} \mid i = 1 \dots n, j = 1 \dots d\}$ as a $nd \times 1$ vector, then we can write down the constraint matrix in (4.2) and (4.3), respectively. We can then find out that for these two matrices, the sum over all the rows in each matrix both equal a $1 \times nd$ vector whose entries are all equal to 1. For any subset J of the matrix formed by the coefficients in constraint (4.2) and (4.3), we can always assign the rows related to (4.2) to J_1 , and the rows related to (4.3) to J_2 . In this case, as both $\sum_{i \in J_1} a_{ij}$ and $\sum_{i \in J_2} a_{ij}$ are smaller than a $1 \times nd$ vector with nd 1s, we will always have $\|\sum_{i \in J_1} a_{ij} - \sum_{i \in J_2} a_{ij}\| \leq 1$. Then this lemma got proven. $\square$

4.3.2 Transform the Nonlinear Programming Problem into a LP

We have transformed our integer programming problem into a nonlinear programming with a separable convex function. We have also shown that the coefficients in the constraints of our formulation form a totally unimodular matrix. Now we can further transform the nonlinear programming problem into a LP based on the method proposed

in [71]. In this transformation, the optimal solutions would not change. The key transformation is named λ -representation as listed below.

$$f(x) = \sum_{h \in \mathcal{P}} f(h) \lambda_h \quad (4.18)$$

$$\sum_{h \in \mathcal{P}} h \lambda_h = x \quad (4.19)$$

$$\sum_{h \in \mathcal{P}} \lambda_h = 1 \quad (4.20)$$

$$\forall \lambda_h \in R^+, \forall h \in \mathcal{P} \quad (4.21)$$

where $\mathcal{P}$ is the set that consists of all the possible values of x . Therefore in our case, $\mathcal{P} = \{0, 1\}$. As we can see that, it introduces $|\mathcal{P}|$ extra variables λ_h in the transformation and makes the original function to be a new function over λ_h and x . As indicated in the formulation, λ_h could be any positive real numbers and x equals the weighted combination of λ_h . By applying λ -representation to (17), we can easily get the new form of our problem, which is a LP as listed below:

$$\min_{X, \lambda} \quad \sum_{i=1}^n \sum_{j=1}^d \left(\sum_{h \in \mathcal{P}} k^{(c_{ij} + e_{ij}) \cdot h} \lambda_{ij}^h \right) \quad (4.22)$$

$$\text{s.t.} \quad \sum_{h \in \mathcal{P}} h \lambda_{ij}^h = x_{ij}, \quad \forall i \in \mathcal{N}, \quad \forall j \in \mathcal{D} \quad (4.23)$$

$$\sum_{h \in \mathcal{P}} \lambda_{ij}^h = 1, \quad \forall i \in \mathcal{N}, \quad \forall j \in \mathcal{D} \quad (4.24)$$

$$\lambda_{ij}^h \in R^+, \quad \forall i \in \mathcal{N}, \quad \forall j \in \mathcal{D}, \quad \forall h \in \mathcal{P} \quad (4.25)$$

$$(4.2), (4.3), (4.4), (4.5). \quad (4.26)$$

As $\mathcal{P} = \{0, 1\}$, we can further expand and simplify the above formulation to get our final formulation as follows:

$$\min_{\lambda} \quad \sum_{i=1}^n \sum_{j=1}^d ((k^{(c_{ij} + e_{ij})} - 1) \cdot \lambda_{ij}^1) \quad (4.27)$$

$$\text{s.t.} \quad \lambda_{ij}^1 = x_{ij}, \quad \forall i \in \mathcal{N}, \quad \forall j \in \mathcal{D} \quad (4.28)$$

$$(4.2), (4.3), (4.4), (4.5), (4.25). \quad (4.29)$$

Figure 4.2: The design of *Flutter* in Spark.

We can clearly see that it is a LP with only nd variables, where n is the number of tasks and d is the number of datacenters. As it is a LP, it can be efficiently solved by standard linear programming solvers like Breeze [72] in Scala [73], and because the coefficients in the constraints form a totally unimodular matrix, its optimal solutions for X are integral and exactly the same as the solutions of the original ILP problem.

4.4 Design and Implementation

After we discussed how our task scheduling problem can be solved efficiently, we are now ready to see how we implement it in Spark, a modern framework popular for big data processing.

Spark is a fast and general distributed data analysis framework. Different from disk-based Hadoop [74], Spark would cache a part of the intermediate results in memory, thus it would greatly speed up iterative jobs as it can directly obtain the outputs of the previous stage from the memory instead of the disk. Now as Spark becomes more and more mature, several projects designed for different applications are built upon Spark such as MLlib, Spark Streaming and Spark SQL. All these projects rely on the core module of Spark, which contains several fundamental functionalities of Spark including *Resilient Distributed Datasets* (RDD) and scheduling.

To incorporate our scheduling algorithm in Spark, we override the scheduling modules to implement our algorithm. From the top of the view, after a job is launched in

Spark, the job would be transformed into a DAG of tasks and handled by the DAG scheduler. Then, the DAG scheduler would first check whether the parent stages of the final stage are complete. If they are, the final stage is directly submitted to the task scheduler. If not, the parent stages of the final stage are submitted recursively until the DAG scheduler finds a ready stage.

The detailed architecture of our implementation can be seen in Figure 4.2. As we can observe from the figure, after the DAG scheduler finds a ready stage, it would create a new **TaskSet** for that ready stage. Here if the **TaskSet** is a set of reduce tasks, we would first get the output information of the map tasks from the **MapOutputTracker**, and then save it to this **TaskSet**. Then this **TaskSet** would be submitted to the task scheduler and added to a list of pending **TaskSets**. When the **TaskSets** are waiting for resources, the **SchedulerBackend**, which is also the cluster manager, would offer some free resources in the cluster. After receiving the resources, *Flutter* would pick a **TaskSet** in the queue, and determine which task should be assigned to which executor. It also needs to interact with **TaskSetManager** to obtain the description of the tasks, and later return these task descriptions to the **SchedulerBackend** for launching the tasks. During the entire process, getting the outputs of the map tasks and the scheduling process are the two key steps; in what follows, we will present more details about these two steps.

4.4.1 Obtaining Outputs of the Map Tasks

Flutter needs to compute the transfer time to obtain all the intermediate results for each reduce task if it is scheduled to one datacenter. Therefore, obtaining the information about the outputs of map tasks including both the locations and the sizes is a key step towards our goal. Here we will first introduce how we obtain the information about the map outputs.

A **MapOutputTracker** is designed in the driver of Spark to let reduce tasks know where to fetch the outputs of the map tasks. It works as follows. Each time when a map task finishes, it would register the sizes and the locations of its outputs to the **MapOutputTracker** in the driver. Then if the reduce tasks want to know the locations of the map outputs, it will send messages to the **MapOutputTracker** directly to obtain the information.

In our case, we can obtain the output information of map tasks in the DAG scheduler through the `MapOutputTracker`, as the map tasks have already registered its output information to the `MapOutputTracker`. We can directly save the output information of map tasks to the `TaskSet` of reduce tasks before submitting the `TaskSet` to the task scheduler. Therefore the `TaskSet` would carry the output information of the map tasks and be submitted to the task scheduler for task scheduling.

4.4.2 Task Scheduling with Flutter

The task scheduler serves as a “bridge” that connects tasks and resources (executors in Spark). On one hand, it will keep receiving `TaskSets` from the DAG scheduler. On the other hand, it would be notified if there are newly available resources by the `SchedulerBackend`. For instance, each time when a new executor joins the cluster or an executor has finished one task, it would offer its resources along with its hardware specifications to the task scheduler. Usually, multiple offers from several executors would reach the task scheduler at the same time. After receiving these resource offers, the task scheduler then starts to use its scheduling algorithm to pick up the right pending tasks that are most suited to the offered resources.

In our task scheduling algorithm, after we receive the resource offers, we first pick a `TaskSet` in the sorted list of `TaskSets` and check whether it has shuffle dependency. In other words, we want to check whether tasks in this `TaskSet` are reduce tasks. If they are, we need to do two things. The first is to get the output information of the map tasks and calculate the transfer times for each possible scheduling decision. We do not consider the execution time of the tasks in the implementation because the execution time of the tasks in a stage are almost uniform. The second is to figure out the amount of available resources on each datacenter through received resource offers. After these two steps, we feed these information to our linear programming solver, and the solver would return an index of the most suitable datacenter for each reduce task. Finally, we randomly choose a host that has enough resource for the task on that datacenter and return the task description to `SchedulerBackend` for launching the task. If the `TaskSet` does not have shuffle dependency, the default delay scheduling [57] would be adopted. Thus each time, when there are new resource offers, and the pending `TaskSet` is a set of reduce tasks, *Flutter* would be invoked. Otherwise, the default scheduling strategy is used.

4.5 Performance Evaluation

In this section, we will present our experimental setup in geo-distributed datacenters and detailed experiment results on real-world workloads.

4.5.1 Experimental Setup

We first describe the testbed we used in our experiments, and then briefly introduce the applications, baselines and metrics used throughout the evaluations.

Testbed: Our experiments are conducted on 6 datacenters with a total of 25 instances, among which two datacenters are in Toronto. The other datacenters are located at various academic institutions: Victoria, Carleton, Calgary and York. All the instances used in the experiments are m.large, which has 4 cores and 8 GB of main memory. The bandwidth capacities among VMs in these regions are measured by `iperf` and are shown in Table 4.2. The datacenters in Ontario are inter-connected through dedicated 1GE link. Hence we can see in the table that the bandwidth capacities between the datacenters in Toronto, Carleton and York are relatively high, while they are still lower than the bandwidth capacities within the same datacenter.

The distributed file system used in our geo-distributed cluster is the Hadoop Distributed File System (HDFS) [74]. We use one instance as the master node for both HDFS and Spark. All the other nodes are served as `datanodes` and worker nodes. The block size in HDFS is 128MB, and the number of replications is 3. Our method does not need to explicitly manipulate the data placements, so we just upload our data through the master node of HDFS, and the data is then distributed to different datacenters based on the default fault tolerance strategies, which is similar with the practice in [13].

Applications: We deploy three applications on Spark. They are WordCount, PageRank [75] and GraphX [76].

- **WordCount:** WordCount calculates the frequency of every single word appearing in a single or batch of files. It would first calculate the frequency of words in each partition, and then aggregate the results in the previous step to get the final result. We choose WordCount because it is a fundamental application in distributed data processing and it can be used to process the real-world data traces such as Wikipedia dump.

- **PageRank:** It computes the weights for websites based on the number and quality of links that point to the websites. This method relies on the assumption that a website is important if many other important website is linking to it. It is a typical data processing application with multiple iterations. In our case, we use it for calculating both the ranks for the websites and the impact of users in real social networks.
- **GraphX:** GraphX is a module built upon Spark for parallel graph processing. We run the application *LiveJournalPageRank* as the representative application of GraphX. Even though the application is also named “PageRank,” the computation module is completely different on GraphX. We choose it because we also wish to evaluate *Flutter* on systems built upon Spark.

Inputs: For WordCount, we use 10GB of Wikipedia dump as the input. For PageRank, we use an unstructured graph with 875713 nodes and 5105039 edges released by Google [77] and a directed graph with 1632803 nodes and 30622564 edges from Pokec online social network [78]. For GraphX, we adopt a directed graph in *LiveJournal* online social network with 4847571 nodes and 68993773 edges [77], where *LiveJournal* is a free online community.

Baseline: We compare our task scheduler with delay scheduling [57], which is the default task scheduler in Spark.

Metrics: The first two metrics used are job completion times and stage completion times of the three application. As the bandwidths among different datacenters are expensive in terms of cost, so we also take the amount of traffic transferred among different datacenters as another metric. Moreover, we also report the running times of solving the LP in different scales to show the scalability of our approach.

4.5.2 Experimental Results

In our experiments, we wish to answer the following questions. (1) What are the benefits of *Flutter* in terms of job completion times, stage completion times, as well as the volume of data transferred among different datacenters? (2) Is *Flutter* scalable in terms of the times to compute the scheduling results, especially for short-running tasks?

	Tor-1	Tor-2	Victoria	Carleton	Calgary	York
Tor-1	1000	931	376	822	99.5	677
Tor-2	-	1000	389	935	97.1	672
Victoria	-	-	1000	381	82.5	408
Carleton	-	-	-	1000	93.7	628
Calgary	-	-	-	-	1000	95.6
York	-	-	-	-	-	1000

Note: “Tor” is short for Toronto. Tor-1 and Tor-2 are two datacenters located at Toronto.

Table 4.2: Available bandwidths across geo-distributed datacenters (Mbps).

4.5.2.1 Job Completion Times

We plot the job completion times of the three applications in Figure 4.3. As we can see that, completion times of all three applications with *Flutter* have been reduced. More specifically, *Flutter* reduced the job completion time of WordCount and PageRank by 22.1% and 25%, respectively. The completion time of GraphX is also reduced by more than 20 seconds. There are primarily two reasons for the improvements. The first is that *Flutter* can adaptively schedule the reduce tasks to a datacenter that would cost the least amount of transfer times to get all the intermediate results, thus it can start the tasks as soon as possible. The second is that *Flutter* would schedule the tasks in the stage as a whole, thus it can significantly mitigate the stragglers — the slow-running tasks in that stage — and further improve the overall performance.

It seems that the improvements in terms of job completion times on GraphX are small, which may be because that it only spends a small portion of the job completion time for shuffle reads, as the total size of shuffle reads are relatively lower than other applications, which limits the room for improvements. Even though the job completion time is not reduced significantly for GraphX applications, we will show that *Flutter* would significantly reduce the amount of traffic transferred across different datacenters for GraphX applications.

Figure 4.3: The job computation times of the three workloads.

(a) WordCount

(b) PageRank

(c) GraphX

Figure 4.4: The completion times of stages in WordCount, PageRank and GraphX.

4.5.2.2 Stage Completion Times

As *Flutter* schedules the tasks stage by stage, we also plot the completion times of stages in these applications in Figure 4.4, we can thus have a closer view of the scheduling performance of both our approach and the default scheduler in Spark, by checking the performance gap stage by stage and find out how the overall improvements of job completion times are achieved. We will explain the performance of the three applications one by one.

For WordCount, we repartition the input datasets as the input size is large. Therefore it has three stages as shown in Figure 4.4(a). In the first stage, as it is not a stage with shuffle dependency, we use the default scheduler in Spark. Thus the performance achieved is almost the same. The second stage is a stage with shuffle dependency. We can see that the stage completion time of this stage for the two schedulers are almost the same, which is because the default scheduler also schedules the tasks in the same datacenters as ours while not necessarily in the same executors. In the last stage, our approach takes only 163 seconds, while the default scheduler in Spark takes 295 seconds, which is almost twice as long. The performance improvements are due to both network-awareness and stage-awareness, as *Flutter* schedules the tasks in that stage as a whole, and take the transfer times into consideration at the same time. It can effectively reduce the number of straggler tasks and the transfer times to get all the inputs.

We draw the stage completion times of PageRank in Figure 4.4(b). As we can see in this figure, it has 13 stages in total, including two *distinct stages*, 10 *reduceByKey stages* and one *collect stage* to collect the final results. We have 10 *reduceByKey stage* because the number of iterations is 10. Except the first *distinct stage*, all the other stages are shuffle dependent. So we adopt *Flutter* instead of delay scheduling for task scheduling in those stages. As we can see that in stage 2, 3 and 13, we have far shorter stage completion times compared with the default scheduler. Especially in the last stage, *Flutter* takes only 1 second to finish that stage, while the default scheduler takes 11 seconds.

Figure 4.4(c) depicts the completion times of reduce stages in GraphX. As the total number of stages is more than 300, we only draw the stages named “reduce stage” in that job. Because the stage completion times of these two schedulers are similar, we

only draw the stage completion time of *Flutter* to illustrate the performance of GraphX. First we can see that the first reduce stage took about 28 seconds, while the following reduce stages completed quickly, which takes only 0.4 seconds. This may be for the reason that GraphX is designed for reducing the data movements and duplications, thus the stages can complete very quickly.

4.5.2.3 Data Volume Transferred across Datacenters

After we see the improvements of job completion times, we are now ready to evaluate the performance of *Flutter* in terms of the amount of data transferred across geo-distributed datacenters in Figure 4.5. In WordCount, the amount of data transferred across different datacenters with the default scheduler is around three times to the one of *Flutter*. The amount of data across datacenters when running GraphX is four times to our approach. In the case of PageRank, we also achieved lower volumes of data transfers than the default scheduler.

Even though reducing the amount of data transferred across different datacenters is not the main goal of our optimization, we find out that it is in line with the goal of reducing the job completion time for data processing applications on distributed datacenters. This is because the bandwidth capacities across VMs in the same datacenter are higher than those on inter-datacenter links, so when *Flutter* tries to place the tasks to reduce the transfer times to get all the inputs, it may prefer to put the tasks in the datacenter that has most of the input data. Thus, it is able to reduce the volume of data transfers across different datacenters by a substantial margin.

4.5.2.4 Scalability

Practicality is one of the main objectives when designing *Flutter*, which means that *Flutter* needs to be efficient at runtime. Therefore, we record the time it takes to solve the LP when we run Spark applications. The results have been shown in Figure 4.6. In the figure, the number of variables vary from 6 to 120 and the computation times are averaged over multiple runs. We can see that the linear program is rather efficient: it takes less than 0.1 second to return a results for 60 variables. Moreover, the computation time is less than 1 second for 120 variables, which is also acceptable because the transfer times could be tens of seconds across distributed datacenters. *Flutter* is scalable for two reasons: (1) it is formulated as an efficient LP; and (2) the number of

Figure 4.5: The amount of data transferred among different datacenters.

variables in our problem is small because the number of datacenters and reduce tasks are both small in practice.

4.6 Summary

In this chapter, we focus on how tasks may be scheduled closer to the data across geo-distributed datacenters. We first find out that the network could be a bottleneck for geo-distributed big data processing, by measuring available bandwidth across Amazon EC2 datacenters. Our problem is then formulated as an integer linear programming problem, considering both the network and the computational resource constraints. To achieve both optimal results and high efficiency of the scheduling process, we are able to transform the integer linear programming problem into a linear programming problem, with exactly the same optimal solutions.

Based on these theoretical insights, we have designed and implemented *Flutter*, a new framework for scheduling tasks across geo-distributed datacenters. With real-world performance evaluation using an inter-datacenter network testbed, we have shown convincing evidence that *Flutter* is not only able to shorten the job completion times, but also to reduce the amount of traffic that needs to be transferred across different datacenters. As part of our future work, we will investigate how data placement,

Figure 4.6: The computation times of *Flutter*'s linear program at different scales.

replication strategies, and task scheduling can be jointly optimized for even better performance in the context of wide-area big data processing.

Chapter 5

Conclusion

As more and more applications are hosted in one or several datacenters, profiling the datacenter networks and improving the performances of applications on top of those networks have become two crucial issues for the performance of the applications. To address these challenges, in this thesis, we first propose an efficient framework to estimate the traffic matrix in intra-datacenter networks (Chapter 3). We then design a lightweight task scheduler to speed up the data analysis jobs in inter-datacenter networks (Chapter 4). We summarize our contributions in Section 5.1 and some future directions in Section 5.2.

5.1 Research Contributions

In Chapter 3, we have shown two observations about the traffic characteristics in datacenter networks and proposed a two-step method to get the traffic matrix estimation results. In the first step, we estimate the prior traffic matrix among ToRs based on the first observation. The first observation is that the TMs among ToRs are sparse, thus the prior TMs should also be sparse to gain enough accuracy. According to this observation, we have derived two methods to get the prior TM in both public datacenter networks and private datacenter networks. In public datacenter networks, we use the resource provisioning information to infer the communication pairs among VMs and ToRs. While in private datacenter networks, we have more detailed information about the usage of the hardware resources. More specifically, in private datacenters, we know not only who is using the VMs but also what services are deployed in those VMs. In

this case, we adopt service placement information to improve the estimation accuracy of prior TMs in private datacenter networks as different services rarely communicate with each other [64].

In the second step, we have successfully narrowed the gap between the number of unknown variables and the number of available measurements motivated by the second observation. In the second observation, we have found out that the utilizations of most links in datacenter networks are very low (in the scale of 0.01 percent). Thus we propose to “eliminate” those lowly utilized links to reduce the difference between the number of variables and the number of measurements. The reasons for this step are as follows. First, those lowly utilized links only carry limited information compared with other links. Second, the number of unknown variables would be reduced a lot with the decrease of each lowly utilized link. Overall, we can greatly mitigate the severe under-determined problem and make it a more determined one.

In Chapter 4, we have formulated our problem as an integer linear programming problem (ILP) and further transformed it to a linear programming problem (LP). The objective function of our problem is to minimize the longest task completion time in each stage, which is also to minimize the stage completion time. Regarding the constraints, we have both considered the bandwidths among different datacenters, which affect the time to get the intermediate data, and resource constraints in each datacenter. The original formulation of the problem is an integer linear programming problem (ILP), which cannot be efficiently solved for an online task scheduler. However, we have fortunately found out that this ILP can be transformed to linear programming problem if it meets two conditions, which are separable convex objective function and totally unimodular constraint matrix. We have proved that the original problem can meet those two conditions after transformation and thus can be solved efficiently in an online fashion.

Besides theoretical analysis and transformation of the formulation, we have also implemented our task scheduler in a popular data analysis framework Spark and evaluated its performance over the default task scheduler. In the implementation part, we first obtained the sizes of intermediate data among different stages and then integrated our formulation in the task scheduler to compute the scheduling results. In the experiments, we used real datasets of social networks and Wikipedia to validate the performance of our task scheduler over several popular benchmark applications.

With those results, we have shown that our task scheduler can decrease the job completion time, stage completion time, and the amount of data transferred among different datacenters effectively by a substantial margin.

In sum, in this thesis, we have contributed in proposing

- a framework estimating the traffic matrix in both public and private datacenter networks
 - two observations about the traffic characteristics in DCNs are revealed and they serve as our motivations for the measurement framework.
 - two specific methods that utilize the operational logs in datacenter networks are proposed separately for public datacenter networks and private datacenter networks.
- a task scheduler for geo-distributed big data analysis
 - a batch of measurements for bandwidths among different datacenters in inter-datacenter networks.
 - a carefully designed ILP formulation and an optimization method that transforms the ILP to a LP.
 - an implementation of our task scheduler on Spark, a framework popular for data analysis.

5.2 Future Directions

In Chapter 3, we assume that communications could only happen among the servers/VMs running the same services or the servers/VMs belonging to the same user. Some special cases that violate these assumptions actually exist. In our future work, such special cases that fail to follow the two assumptions will be considered. We could figure out the correlations among different services and the correlations among the VMs belonging to different users using learning methods. Besides, we are also interested in combining network tomography with direct measurements such as adopting software defined network (SDN) to derive a hybrid network monitoring scheme. The initial results have been reported in [79].

In Chapter 4, we only consider utilizing task scheduler to improve the performance of data analysis jobs and there is a lot we can do in this scheme. First, some other issues such as data placements and data replications also have great impacts on the performance of the data analysis jobs. For example, in some cases, the size of data after processing is bigger than the size of input data [12]. Then it would be better to move the data first before processing to reduce the total amount of data transfers. To this end, we may consider finding a way to jointly optimize task scheduling, data placements, and data replication strategies to improve the overall performance of geo-distributed data processing jobs. Second, we should also take the bandwidth costs into consideration for the task scheduling problem as the bandwidth costs are diverse and high among different datacenters. Thus it is possible to propose cost-constrained solutions for task/job scheduling, which is closer to the real practice especially when using the public clouds for data analysis. We can also take the bandwidth cost as the sole objective to be optimized.

References

- [1] CISCO DATA CENTER INFRASTRUCTURE. **2.5 Design Guide**. <http://goo.gl/kBpzgh>. Accessed: 2016-04-19. ix, 6, 18, 41
- [2] KANDULA SRIKANTH, PADHYE JITENDRA, AND BAHL PARAMVIR. **Flyways To De-Congest Data Center Networks**. In *Proc. of ACM HotNets*, 2009. ix, 1, 16, 21, 22
- [3] THEOPHILUS BENSON, ADITYA AKELLA, AND DAVID A MALTZ. **Network Traffic Characteristics of Data Centers in the Wild**. In *Proc. of ACM IMC*, pages 267–280, 2010. ix, 7, 9, 10, 22, 23, 42
- [4] DANIEL HALPERIN, SRIKANTH KANDULA, JITENDRA PADHYE, PARAMVIR BAHL, AND DAVID WETHERALL. **Augmenting Data Center Networks with Multi-Gigabit Wireless Links**. In *Proc. of ACM SIGCOMM*, pages 38–49, 2011. 1, 16, 22
- [5] XIAODONG WANG, YANJUN YAO, XIAORUI WANG, KEFA LU, AND QING CAO. **Carp0: Correlation-aware Power Optimization in Data Center Networks**. In *Proc. of IEEE INFOCOM*, pages 1125–1133, 2012. 2
- [6] A. R. CURTIS, WONHO KIM, AND P. YALAGANDULA. **Mahout: Low-overhead Datacenter Traffic Management Using End-host-based Elephant Detection**. In *Proc. of IEEE INFOCOM*, pages 1629–1637, 2011. 2, 9, 15
- [7] THEOPHILUS BENSON, ASHOK ANAND, ADITYA AKELLA, AND MING ZHANG. **MicroTE: Fine Grained Traffic Engineering for Data Centers**. In *Proc. of ACM CoNEXT*, pages 8:1–8:12, 2011. 2, 9

-
- [8] SRIKANTH KANDULA, SUDIPTA SENGUPTA, ALBERT GREENBERG, PARVEEN PATEL, AND RONNIE CHAIKEN. **The Nature of Data Center Traffic: Measurements & Analysis**. In *Proc. of ACM IMC*, pages 202–208, 2009. 2, 9, 11, 15, 16, 22, 42, 45
- [9] MOHAMMAD AL-FARES, SIVASANKAR RADHAKRISHNAN, BARATH RAGHAVAN, NELSON HUANG, AND AMIN VAHDAT. **Hedera: Dynamic Flow Scheduling for Data Center Networks**. In *Proc. of USENIX NSDI*, 2010. 2, 9, 15, 42
- [10] NICK MCKEOWN, TOM ANDERSON, HARI BALAKRISHNAN, GURU PARULKAR, LARRY PETERSON, JENNIFER REXFORD, SCOTT SHENKER, AND JONATHAN TURNER. **OpenFlow: Enabling Innovation in Campus Networks**. *ACM SIGCOMM CCR*, **38**(2):69–74, 2008. 2, 9, 15
- [11] ASHISH VULIMIRI, CARLO CURINO, B GODFREY, J PADHYE, AND G VARGHESE. **Global Analytics in the Face of Bandwidth and Regulatory Constraints**. In *Proc. of USENIX NSDI*, 2015. 2, 3, 12, 49
- [12] QIFAN PU, GANESH ANANTHANARAYANAN, PETER BODIK, SRIKANTH KANDULA, ADITYA AKELLA, VICTOR BAHL, AND ION STOICA. **Low Latency Geodistributed Data Analytics**. In *Proc. of ACM SIGCOMM*, 2015. 2, 3, 13, 49, 73
- [13] KONSTANTINOS KLOUDAS, MARGARIDA MAMEDE, NUNO PREGUIÇA, AND RODRIGO RODRIGUES. **Technical Report: Pixida: Optimizing Data Parallel Jobs in Bandwidth-Skewed Environments**. Technical report, 2015. 2, 3, 12, 49, 62
- [14] MATEI ZAHARIA, MOSHARAF CHOWDHURY, TATHAGATA DAS, ANKUR DAVE, JUSTIN MA, MURPHY MCCAULEY, MICHAEL J FRANKLIN, SCOTT SHENKER, AND ION STOICA. **Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing**. In *Proc. of USENIX NSDI*, 2012. 3, 51
- [15] MOHAMMAD AL-FARES, ALEXANDER LOUKISSAS, AND AMIN VAHDAT. **A Scalable, Commodity Data Center Network Architecture**. In *Proc. of ACM SIGCOMM*, pages 63–74, 2008. 6, 7, 9, 15, 18, 41

-
- [16] ALBERT GREENBERG, JAMES R HAMILTON, NAVENDU JAIN, SRIKANTH KANDULA, CHANGHOON KIM, PARANTAP LAHIRI, DAVID A MALTZ, PARVEEN PATEL, AND SUDIPTA SENGUPTA. **VL2: A Scalable and Flexible Data Center Network**. In *Proc. of ACM SIGCOMM*, pages 51–62, 2009. 6, 7, 9, 15, 18, 42
- [17] CHARLES CLOS. **A Study of Non-blocking Switching Networks**. *Bell System Technical Journal*, **32**(2):406–424, 1953. 7
- [18] C. HOPPS. **Analysis of an Equal-Cost Multi-Path Algorithm**, 2000. 7, 34, 42
- [19] CHUANXIONG GUO, HAITAO WU, KUN TAN, LEI SHI, YONGGUANG ZHANG, AND SONGWU LU. **Dcell: a Scalable and Fault-tolerant Network Structure for Data Centers**. In *Proc. of ACM SIGCOMM*, pages 75–86, 2008. 8
- [20] CHUANXIONG GUO, GUOHAN LU, DAN LI, HAITAO WU, XUAN ZHANG, YUNFENG SHI, CHEN TIAN, YONGGUANG ZHANG, AND SONGWU LU. **BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers**. In *Proc. of ACM SIGCOMM*, pages 63–74, 2009. 8, 9, 15, 41
- [21] DAN LI, CHUANXIONG GUO, HAITAO WU, KUN TAN, YONGGUANG ZHANG, AND SONGWU LU. **FiConn: Using Backup Port for Server Interconnection in Data Centers**. In *Proc. of IEEE INFOCOM*, pages 2276–2285, 2009. 8
- [22] HAITAO WU, GUOHAN LU, DAN LI, CHUANXIONG GUO, AND YONGGUANG ZHANG. **MDCube: a High Performance Network Structure for Modular Data Center Interconnection**. In *Proc. of ACM CoNext*, pages 25–36, 2009. 8
- [23] PHILLIPA GILL, NAVENDU JAIN, AND NACHIAPPAN NAGAPPAN. **Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications**. In *Proc. of ACM SIGCOMM*, pages 350–361, 2011. 9, 15, 42
- [24] JOE WENJIE JIANG, TIAN LAN, SANGTAE HA, MINGHUA CHEN, AND MUNG CHIANG. **Joint VM Placement and Routing for Data Center Traffic Engineering**. In *Proc. of IEEE INFOCOM*, pages 2876–2880, 2012. 9, 15

-
- [25] MINLAN YU, LAVANYA JOSE, AND RUI MIAO. **Software Defined Traffic Measurement with OpenSketch**. In *Proc. of USENIX NSDI*, pages 29–42, 2013. 9
- [26] N.L.M. VAN ADRICHEM, C. DOERR, AND F.A. KUIPERS. **OpenNetMon: Network Monitoring in OpenFlow Software-Defined Networks**. In *Proc. of IEEE/IFIP NOMS*, pages 1–8, 2014. 9
- [27] MEHDI MALBOUBI, LIYUAN WANG, CHEN-NEE CHUAH, AND PUNEET SHARMA. **Intelligent SDN based Traffic (de)Aggregation and Measurement Paradigm (iSTAMP)**. In *Proc. of IEEE INFOCOM*, pages 934–942, 2014. 9
- [28] ARSALAN TAVAKOLI, MARTIN CASADO, TEEMU KOPONEN, AND SCOTT SHENKER. **Applying NOX to the Datacenter**. In *Proc. of HotNets*, 2009. 9
- [29] THEOPHILUS BENSON, ASHOK ANAND, ADITYA AKELLA, AND MING ZHANG. **Understanding Data Center Traffic Characteristics**. *ACM SIGCOMM CCR*, 40(1):92–99, 2010. 9, 10
- [30] JACEK P KOWALSKI AND BOB WARFIELD. **Modelling Traffic Demand between Nodes in a Telecommunications Network**. In *Proc. of ATNAC*. Cite-seer, 1995. 10, 26
- [31] YIN ZHANG, MATTHEW ROUGHAN, NICK DUFFIELD, AND ALBERT GREENBERG. **Fast Accurate Computation of Large-scale IP Traffic Matrices from Link Loads**. In *Proc. of ACM SIGMETRICS*, pages 206–217, 2003. 10, 11, 15, 16, 37
- [32] YIN ZHANG, MATTHEW ROUGHAN, WALTER WILLINGER, AND LILI QIU. **Spatio-temporal Compressive Sensing and Internet Traffic Matrices**. In *Proc. of ACM SIGCOMM*, pages 267–278, 2009. 10, 11, 15, 19, 37
- [33] AUGUSTIN SOULE, ANUKOOL LAKHINA, NINA TAFT, KONSTANTINA PAPA-
GIANNAKI, KAVE SALAMATIAN, ANTONIO NUCCI, MARK CROVELLA, AND
CHRISTOPHE DIOT. **Traffic Matrices: Balancing Measurements, Inference and Modeling**. In *Proc. of ACM SIGMETRICS*, pages 362–373, 2005. 10, 11, 15, 19

-
- [34] MATTHEW ROUGHAN, ALBERT GREENBERG, CHARLES KALMANEK, MICHAEL RUMSEWICZ, JENNIFER YATES, AND YIN ZHANG. **Experience in measuring backbone traffic variability: Models, metrics, measurements and meaning.** In *Proc. of ACM IMW*, pages 91–92. ACM, 2002. 10
- [35] ANDERS GUNNAR, MIKAEL JOHANSSON, AND THOMAS TELKAMP. **Traffic Matrix Estimation on A Large IP Backbone: A Comparison on Real Data.** In *Proc. of ACM IMC*, pages 149–160, 2004. 10
- [36] PENG QIN, BIN DAI, BENXIONG HUANG, GUAN XU, AND KUI WU. **A Survey on Network Tomography with Network Coding.** *IEEE Communications Surveys & Tutorials*, **16**(4):1981–1995, 2014. 10
- [37] ZHIMING HU, YAN QIAO, JUN LUO, PENG SUN, AND YONGGANG WEN. **CRE-ATE: CoRrelation Enhanced trAffic maTriX Estimation in Data Center Networks.** In *Proc. of IFIP Networking*, pages 1–9, 2014. 10
- [38] ANDREW C HARVEY. *Forecasting, Structural Time Series Models and the Kalman Filter.* Cambridge university press, 1990. 10
- [39] MATTHEW ROUGHAN, YIN ZHANG, WALTER WILLINGER, AND LILI QIU. **Spatio-temporal Compressive Sensing and Internet Traffic Matrices (extended version).** *IEEE/ACM Transactions on Networking*, **20**(3):662–676, 2012. 10
- [40] LIANG MA, TING HE, KIN K LEUNG, DON TOWSLEY, AND ANANTHRAM SWAMI. **Efficient Identification of Additive Link Metrics via Network Tomography.** In *Proc. of IEEE ICDCS*, pages 581–590, 2013. 11
- [41] LIANG MA, TING HE, KIN K LEUNG, ANANTHRAM SWAMI, AND DON TOWSLEY. **Monitor Placement for Maximal Identifiability in Network Tomography.** In *Proc. of IEEE INFOCOM*, pages 1447–1455, 2014. 11
- [42] LIANG MA, TING HE, ANANTHRAM SWAMI, DON TOWSLEY, AND KIN K LEUNG. **On Optimal Monitor Placement for Localizing Node Failures via Network Tomography.** *Performance Evaluation*, **91**:16–37, 2015. 11

-
- [43] NIKOLAOS LAOUTARIS, MICHAEL SIRIVIANOS, XIAOYUAN YANG, AND PABLO RODRIGUEZ. **Inter-datacenter Bulk Transfers with Netstitcher**. In *Proc. of ACM SIGCOMM*, 2011. 11
- [44] SUSHANT JAIN, ALOK KUMAR, SUBHASREE MANDAL, JOON ONG, LEON POUTIEVSKI, ARJUN SINGH, SUBBAIAH VENKATA, JIM WANDERER, JUNLAN ZHOU, MIN ZHU, ET AL. **B4: Experience with a Globally-deployed Software Defined WAN**. In *Proc. of ACM SIGCOMM*, pages 3–14, 2013. 11, 12
- [45] CHI-YAO HONG, SRIKANTH KANDULA, RATUL MAHAJAN, MING ZHANG, VIJAY GILL, MOHAN NANDURI, AND ROGER WATTENHOFFER. **Achieving High Utilization with Software-driven WAN**. In *Proc. of ACM SIGCOMM*, pages 15–26, 2013. 11, 12
- [46] HONG ZHANG, KAI CHEN, WEI BAI, DONGSU HAN, CHEN TIAN, HAO WANG, HAIBING GUAN, AND MING ZHANG. **Guaranteeing Deadlines for Inter-datacenter Transfers**. In *Proc. of ACM Eurosys*, page 20, 2015. 12
- [47] ASHISH VULIMIRI, CARLO CURINO, BRIGHTEN GODFREY, KONSTANTINOS KARANASOS, AND GEORGE VARGHESE. **WANalytics: Analytics for a Geodistributed Data-intensive World**. In *Proc. of CIDR*, 2015. 12
- [48] YONGCHUL KWON, MAGDALENA BALAZINSKA, BILL HOWE, AND JEROME ROLIA. **Skewtune: mitigating skew in mapreduce applications**. In *Proc. of ACM SIGMOD*, pages 25–36, 2012. 13
- [49] CHAMIKARA JAYALATH, JOSE STEPHEN, AND PATRICK EUGSTER. **From the Cloud to the Atmosphere: Running MapReduce Across Data Centers**. *IEEE Transactions on Computers*, **63**(1):74–87, 2014. 13
- [50] JEFFREY DEAN AND SANJAY GHEMAWAT. **MapReduce: Simplified Data Processing on Large Clusters**. *Communications of the ACM*, **51**(1):107–113, 2008. 13
- [51] VINOD KUMAR VAVILAPALLI, ARUN C MURTHY, CHRIS DOUGLAS, SHARAD AGARWAL, MAHADEV KONAR, ROBERT EVANS, THOMAS GRAVES, JASON LOWE, HITESH SHAH, SIDDHARTH SETH, ET AL. **Apache Hadoop Yarn: Yet Another Resource Negotiator**. In *Proc. of ACM SoCC*, 2013. 13

-
- [52] BENJAMIN HINDMAN, ANDY KONWINSKI, MATEI ZAHARIA, ALI GHODSI, ANTHONY D JOSEPH, RANDY H KATZ, SCOTT SHENKER, AND ION STOICA. **Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.** In *Proc. of USENIX NSDI*, 2011. 13
- [53] SHANJIANG TANG, BU-SUNG LEE, AND BINGSHENG HE. **Dynamic Slot Allocation Technique for MapReduce Clusters.** In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–8, 2013. 13
- [54] KAY OUSTERHOUT, PATRICK WENDELL, MATEI ZAHARIA, AND ION STOICA. **Sparrow: Distributed, Low Latency Scheduling.** In *Proc. of ACM SOSP*, pages 69–84, 2013. 13
- [55] XIAOQI REN, GANESH ANANTHANARAYANAN, ADAM WIERMAN, AND MINLAN YU. **Hopper: Decentralized Speculation-aware Cluster Scheduling at Scale.** In *Proc. of ACM SIGCOMM*, 2015. 13
- [56] MICHAEL ISARD, VIJAYAN PRABHAKARAN, JON CURREY, UDI WIEDER, KUNAL TALWAR, AND ANDREW GOLDBERG. **Quincy: Fair Scheduling for Distributed Computing Clusters.** In *Proc. of the ACM SIGOPS*, pages 261–276, 2009. 13
- [57] MATEI ZAHARIA, DHRUBA BORTHAKUR, JOYDEEP SEN SARMA, KHALED ELMELEEGY, SCOTT SHENKER, AND ION STOICA. **Delay Scheduling: a Simple Technique for Achieving Locality and Fairness in Cluster Scheduling.** In *Proc. of ACM Eurosys*, pages 265–278, 2010. 13, 61, 63
- [58] XIAOHONG ZHANG, ZHIYONG ZHONG, SHENGZHONG FENG, BIBO TU, AND JIANPING FAN. **Improving Data Locality of MapReduce by Scheduling in Homogeneous Computing Environments.** In *IEEE 9th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 120–126, 2011. 13
- [59] MOHAMMAD HAMMOUD AND MAJD F SAKR. **Locality-aware Reduce Task Scheduling for MapReduce.** In *IEEE Conference on Cloud Computing Technology and Science (CloudCom)*, pages 570–576, 2011. 13

-
- [60] JIONG XIE, SHU YIN, XIAOJUN RUAN, ZHIYANG DING, YUN TIAN, JAMES MAJORS, ADAM MANZANARES, AND XIAO QIN. **Improving Mapreduce Performance through Data Placement in Heterogeneous Hadoop Clusters.** In *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1–9, 2010. 13
- [61] YONG CUI, HONGYI WANG, XIUZHEN CHENG, DAN LI, AND ANTTI YLA-JAASKI. **Dynamic Scheduling for Wireless Data Center Networks.** *IEEE Transactions on Parallel and Distributed Systems*, **24**(12):2365–2374, 2013. 15
- [62] KAI HAN, ZHIMING HU, JUN LUO, AND LIU XIANG. **RUSH: RoUting and Scheduling for Hybrid Data Center Networks.** In *Proc. of IEEE INFOCOM*, 2015. 15
- [63] CHARLES REISS, ALEXEY TUMANOV, GREGORY R. GANGER, RANDY H. KATZ, AND MICHAEL A. KOZUCH. **Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis.** In *Proc. of ACM SoCC*, pages 7:1–7:13, 2012. 15
- [64] PETER BODÍK, ISHAI MENACHE, MOSHARAF CHOWDHURY, PRADEEPKUMAR MANI, DAVID A MALTZ, AND ION STOICA. **Surviving Failures in Bandwidth-Constrained Datacenters.** In *Proc. of ACM SIGCOMM*, pages 431–442, 2012. 16, 29, 31, 71
- [65] HITESH BALLANI, PAOLO COSTA, THOMAS KARAGIANNIS, AND ANTONY IT ROWSTRON. **Towards Predictable Datacenter Networks.** In *Proc. of ACM SIGCOMM*, pages 242–253, 2011. 29
- [66] CHUANXIONG GUO, GUOHAN LU, HELEN J WANG, SHUANG YANG, CHAO KONG, PENG SUN, WENFEI WU, AND YONGGUANG ZHANG. **Secondnet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees.** In *Proc. of ACM Co-NEXT*, pages 15:1–15:12. ACM, 2010. 29
- [67] Y. LUO AND RAMANI DURAI SWAMI. **Efficient Paraller Non-Negative Least Square on Multi-core Architectures.** *SIAM Journal on Scientific Computing*, **33**(5):2848–2863, 2011. 36

-
- [68] MOHAMMAD ALIZADEH, ALBERT GREENBERG, DAVID A MALTZ, JITENDRA PADHYE, PARVEEN PATEL, BALAJI PRABHAKAR, SUDIPTA SENGUPTA, AND MURARI SRIDHARAN. **Data Center Tcp (DCTCP)**. In *Proc. of ACM SIGCOMM*, pages 63–74, 2010. 42
- [69] YU-KWONG KWOK AND ISHFAQ AHMAD. **Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors**. *ACM Computing Surveys*, **31**(4):406–471, 1999. 50, 52
- [70] KAY OUSTERHOUT, RYAN RASTI, SYLVIA RATNASAMY, SCOTT SHENKER, BYUNG-GON CHUN, AND VMWARE ICSI. **Making Sense of Performance in Data Analytics Frameworks**. In *Proc. of USENIX NSDI*, pages 293–307, 2015. 51
- [71] RR MEYER. **A Class of Nonlinear Integer Programs Solvable by a Single Linear Program**. *SIAM Journal on Control and Optimization*, **15**(6):935–946, 1977. 54, 58
- [72] **Breeze: a numerical processing library for Scala**. <https://github.com/scalanlp/breeze>. Accessed: 2016-04-19. 59
- [73] **Scala**. <http://www.scala-lang.org/>. Accessed: 2016-04-19. 59
- [74] **Hadoop**. <https://hadoop.apache.org/>. Accessed: 2016-04-19. 59, 62
- [75] LAWRENCE PAGE, SERGEY BRIN, RAJEEV MOTWANI, AND TERRY WINOGRAD. **The PageRank Citation Ranking: Bringing Order to the Web**. 1999. 62
- [76] JOSEPH E GONZALEZ, REYNOLD S XIN, ANKUR DAVE, DANIEL CRANKSHAW, MICHAEL J FRANKLIN, AND ION STOICA. **Graphx: Graph Processing in a Distributed Dataflow Framework**. In *Proc. of USENIX OSDI*, pages 599–613, 2014. 62
- [77] JURE LESKOVEC, KEVIN J LANG, ANIRBAN DASGUPTA, AND MICHAEL W MAHONEY. **Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters**. *Internet Mathematics*, **6**(1):29–123, 2009. 63

- [78] JURE LESKOVEC AND ANDREJ KREVL. **SNAP Datasets: Stanford Large Network Dataset Collection**. <http://snap.stanford.edu/data>. Accessed: 2016-04-19. 63
- [79] ZHIMING HU AND JUN LUO. **Cracking the Network Monitoring in DCNs with SDN**. In *Proc. of IEEE INFOCOM*, 2015. 72

Publications

JOURNAL ARTICLES

1. **Z. Hu**, Y. Qiao, and J. Luo, “ATME: Accurate Traffic Matrix Estimation in both Public and Private DCNs”. *IEEE Transactions on Cloud Computing*. To appear.
2. **Z. Hu**, Y. Qiao, and J. Luo, “Coarse-Grained Traffic Matrix Estimation for Data Center Networks”. *Elsevier Computer Communications* 56 (2015), pp. 25-34.
3. **Z. Hu** and J. Luo, “Software Defined Network and Polarization Effect Enhanced Network Monitoring in DCNs”. Under submission.
4. **Z. Hu**, B. Li, and J. Luo, “Scheduling Reduce Tasks Across Geo-Distributed Datacenters”. Under submission.

CONFERENCE PAPERS

5. **Z. Hu**, B. Li, and J. Luo, “Flutter: Scheduling Tasks Closer to Data Across Geo-Distributed Datacenters”. In the *Proc. of IEEE INFOCOM*, 2016.
6. **Z. Hu**, and J. Luo, “Cracking Network Monitoring in DCNs with SDN”. In the *Proc. of IEEE INFOCOM*, 2015.
7. K. Han, **Z. Hu**, J. Luo, and L. Xiang, “RUSH: RoUting and Scheduling for Hybrid Data Center Networks”. In the *Proc. of IEEE INFOCOM*, 2015.
8. **Z. Hu**, Y. Qiao, J. Luo, P. Sun, and Y. Wen, “CREATE: CoRrelation Enhanced trAffic maTrix Estimation in Data Center Networks”. In the *Proc. of IFIP Networking*, 2014.

9. Y. Qiao, **Z. Hu**, and J. Luo, “Efficient Traffic Matrix Estimation for Data Center Networks”. In the *Proc. of IFIP Networking*, 2013.