Congestion-Minimizing Network Update in Data Centers

Jiaqi Zheng, *Student Member, IEEE,* Hong Xu*, *Member, IEEE,* Guihai Chen, *Member, IEEE,* Haipeng Dai, *Member, IEEE,* and Jie Wu, *Fellow, IEEE,*

Abstract—The SDN control plane needs to frequently update the data plane as the network conditions change. Since each switch updates its flow table independently and asynchronously, the transition of data plane state—if done directly from the initial to the final stage—may result in serious flash congestion. Prior work strives to find a congestion-free update plan with multiple stages, each with the property that there will be no congestion independent of the update order. Yet congestion-free update may prevent the network from being fully utilized. It also requires solving a series of LP which is time-consuming. In this paper, we propose congestion-minimizing update and focus on two general problems: The first is to find routing at each intermediate stage that minimizes transient congestion for a given number of intermediate stages. The second is to find the minimum number of intermediate stages and an update plan for a given maximum level of transient congestion. We formulate them as two optimization programs and prove their hardness. We propose a set of algorithms to find the update plan in a scalable manner. Extensive experiments with Mininet show that our solution reduces update time by 50% and saves control overhead by 38% compared to prior work.

Index Terms—SDN, data center networks, network update, transient congestion.

1 Introduction

In software defined networking (SDN), a logically centralized controller has a global view of the network state, and is responsible for delivering the control decisions to the data plane. The controller enforces policies by installing, modifying, or deleting forwarding rules in switch flow tables through southbound APIs such as Openflow [37]. SDN presents tremendous advantages for data center networks. Google [21] and Microsoft [19] for example rely on SDN to interconnect their data centers and achieve higher network utilization, lower delay, and less packet drops. The link utilization can approach 100% as reported from Google [21], while traditional networks have only 30% to 40% average utilization. Applications with deadlines [24] also benefit from SDN and their performance can be improved significantly.

Despite the centralization of control plane, the data plane remains a distributed system. When network conditions change due to routing policy reconfiguration, switch upgrade, device failures, or traffic variations, the controller needs to update the data plane by modifying the flow tables. This process is not *atomic* [46]: each switch is updated independently and asynchronously. Thus network update may result in serious congestion during the transient period, even though the initial and final configurations are uncongested. For example, congestion may happen when new flows—those that are supposed to be carried by a switch after the update—arrive before old ones that need to be migrated have left. Updates in both intra-datacenter and inter-datacenter networks, if not carefully planned, may disrupt many applications [19], [30].

Previous work on network update, especially SWAN [19] and zUpdate [30], proposes to find a *congestion-free* update plan to solve this problem. The update plan consists of discrete stages, each of which involves changing flow tables on a set of switches, with the property that there will be no congestion independent of the update order or timing. This approach suffers from several drawbacks, however.

TABLE 1
Running time of LP for congestion-free update

	1K	2K	3K	4K	5K
DCN	0.73 min	1.40 min	2.10 min	2.96 min	4.12 min
WAN	0.60 min	1.01 min	1.57 min	2.43 min	3.12 min

For DCN, the topology is an 8-pod fat-tree, with 16 long-lived flows in the background. For WAN the topology is Microsoft's production network topology from [19]. The initial and final routing is generated randomly. We use the algorithms in [30] and [19] to find a congestion-free update plan for DCN and WAN, respectively, using LINGO as the solver, with different number of flows in the network.

First, to guarantee that a congestion-free update plan always exists, a portion (10%–50% [19]) of the network capacity has to be left vacant before update. This leads to reduced utilization of the expensive network infrastructure, especially the wide-area links. Second, calculating a congestion-free update plan requires solving a series of LPs,

*Hong Xu is the corresponding author.
The work was supported in part by the Hong Kong RGC ECS-21201714, GRF-11202315, CRF-C7036-15G, China 973 projects(2014CB340303), China NSF grants(61472252,61133006,61321491,61502229), NSF grants CNS 1449860, CNS 1461932, CNS 1460971, CNS 1439672, CNS 1301774, ECCS 1231461.
Jiaqi Zheng is with State Key Laboratory for Novel Software Technology, Nanjing University, China, and Department of Computer Science, City University of Hong Kong, Hong Kong. (e-mail: jiaqi369@gmail.com)
Hong Xu is with Department of Computer Science, City University of Hong Kong, Hong Kong. (e-mail: henry.xu@cityu.edu.hk)
Guihai Chen and Haipeng Dai are with State Key Laboratory for Novel Software Technology, Nanjing University, China. (email: gchen@nju.edu.cn, haipengdai@nju.edu.cn)
Jie Wu is with Department of Computer and Information Sciences, Temple University, USA. (e-mail: jiewu@temple.edu)
Part of the work was presented in IEEE ICNP 2015.

which is too slow for production scale networks. Table 1 shows the running time of the algorithms in [19], [30] for data center networks (DCN) and inter-data center wide area networks (WAN) which connect geo-distributed data centers of the same operator. Note that a production DCN have much more than 5K flows [25]. However not all of them need to be explicitly managed by the controller; only elephant flows may be subject to explicit control of SDN and require controller intervention. In Table 1, all flows in DCN refer to elephant flows. When the number of flows is larger than 2K, the running time in both scenarios is well beyond one minute. Thus congestion-free update is infeasible for operators like Google [21] and Microsoft [19] who perform centralized traffic engineering (TE) every five minutes to improve link utilization. In addition, slower update speed limits the controller's ability to react to failures and degrades application performance.

Instead of congestion-free network update [22], in this paper we propose *congestion-minimizing* network update, i.e., the update procedure admits a small extent of transient congestion. We argue that since switches are deeply buffered [18], [52], and many low-priority data transfers tolerate packet loss especially in inter-data center WAN [24], it is worthwhile to explore solutions with a small extent of transient congestion. Our problems widen the scope and allow the operator to navigate a broader design space, where one may trade off update speed, represented by the number of intermediate stages, for performance represented by the maximum level of transient congestion during the update. In previous work, the number of intermediate stages is unknown until a update plan is found and cannot be adjusted [19], [30].

We make three novel contributions in this paper. First, we propose a general optimization framework for two problems of finding congestion-minimizing network update plans: the minimum congestion update problem (MCUP) and bounded congestion update problem (BCUP) both in DCN and WAN. The MCUP optimization aims to determine routing for all ρ intermediate stages, where ρ is given, such that the transient congestion (i.e. maximum link utilization during the transition) is minimized. The BCUP optimization aims to find the minimum number of intermediate stages and corresponding routing for each stage, given the maximum transient congestion threshold σ. For both problems, we take into account single-path routing and multipath routing as constraints to meet the different application requirements, since applications such as video do not work well when their flows are split.

Our second contribution is a set of efficient algorithms to solve MCUP and BCUP. We prove that MCUP and BCUP are NP-hard, and thus focus on designing approximation algorithms and heuristics. For MCUP, we first propose a randomized rounding algorithm and prove that it yields a $\mathcal{O}(\log k)$ upper bound of link congestion, where k is the number of switches. This algorithm only requires solving one LP and has faster run time than prior work. We further propose a greedy improvement algorithm which improves the rounding result by greedily rerouting each flow in each stage. The greedy algorithm has the same approximation ratio $\mathcal{O}(\log k)$ as the rounding algorithm in general topologies, and an approximation ratio of 4 for fat-tree in particular. Based on these two algorithms, we finally design an iterative insertion algorithm to solve BCUP.

Our third contribution is a comprehensive performance evaluation of our algorithms in large-scale DCN and WAN topologies. Simulation results show that our algorithms can reduce average link congestion by 29.1% and 32.3%, respectively, in both scenarios, and save 38% control overhead compared to prior work. We also develop a prototype of our algorithm on Mininet using the Floodlight controller. Experimental results show that our solution is 50% faster than prior work.

The remainder of the paper is organized as follows. We summarize related work in §2. We give formal definitions of both MCUP and BCUP, and analyze their hardness in §3. In §4, we present a rounding algorithm to solve MCUP and prove its congestion upper bound. Based on the solution of the rounding algorithm, in §5 we propose the greedy improvement algorithm. Based on the solution of MCUP, we propose a binary insertion algorithm to solve BCUP in §6. Experimental evaluation and implementation are presented in §7 and §8, and finally §9 concludes the paper.

2 Related Work

We review prior art on network update in both traditional networks and SDN.

In traditional networks with distributed routing protocols, most work focuses on avoiding transient misbehavior during network update. For example, consensus routing [23] considers the problem of eliminating transient state inconsistency. Francois et al. [17] and Kushman et al. [27] present a series of solutions aimed at avoiding traffic disruption during BGP update. Vanbever et al. [47] focus on lossless migration and modification when moving from one routing protocol to another. Raza et al. [45] propose graceful network state migration, which strives to minimize the overall performance disruption by reassigning and maintaining link weights in the network. Noyes et al. [42] propose a tool for automatically synthesizing network updates and avoiding errors caused by manual configurations, such as forwarding loops and access control violations.

Recent work starts to study network updates in SDN. Reitblatt et al. [46] introduce two novel concepts during network updates: per-flow consistency and per-packet consistency, and propose a two-phase commit protocol to preserve consistency when transitioning between two different routing configurations. FLIP [48] combines the advantage of both two-phase commit and order-based rules replacement, which significantly reduce the memory overhead during network update as well as preserve routing policies. FOUM [20] proposes a flow-ordered update mechanism that can guarantee per-packet consistency in adversarial settings. Ludwig et al. [32] aim to minimize the number of sequential controller interactions when directly transitioning from the initial to the final stage. They prove that it is NP-hard to find such an update sequence that avoids all the forwarding loops. They thus introduce relaxed loop-freedom, which is not harmful and can be solved in polynomial time. Saeed et al. [8] show that finding the maximum update set in each round is NP-hard for both strong and relaxed loop-freedom. Another work from Ludwig et al. is Waypoint [31],

[33], which considers network update for middleboxes [43]. However, this transitioning procedure does not consider transient congestion. McGeer [36] uses SDN controller as a cache to forward packets during update procedure. This method may bring the congestion on the communication channel between controller and switches, when the number of flows is large. ICU [26] incrementally updates forwarding rules in the switch, which takes limited flow table space into consideration. Jedidiah et al. [35] develop an automatic synthesizing update program that is guaranteed to preserve specified properties.

For data centers, SWAN [19] and zUpdate [30] try to find congestion-free update plans in WAN and DCN, respectively. SWAN shows that if each link has certain slack capacity, there always exists a congestion-free update sequence. This condition is too strong to always hold in practice. Brandt et al. [11] propose that a congestion-free update sequence still exists even if some links are full. They analyze the cases of splittable and unsplittable flows, and propose a polynomial time algorithm to find a congestion-free update sequence for splittable flows. Dionysus [22] employs dependency graphs to find a fast congestion-free update plan according to different runtime conditions of switches. Cupid [49] divides the global update dependencies among switches into local restrictions to avoid high overhead when generating a update plan. Mizrahi et al. [38], [39], [40] propose time synchronization protocols between controller and data plane, which use accurate timing to trigger network updates and reduce congestion. CCG [51] studies how to safely perform customizable consistency polices in order to minimize transition delay. CUP [34] calculates network update plans based on a user requirements model. Brandt et al. [10] study network update for anycast network flows, i.e., the flow can be routed to any node in the destination set during the update.

3 An Optimization Framework

We introduce our optimization framework for MCUP and BCUP in this section.

3.1 A Motivating Example

In a software defined data center network, whenever the topology or traffic matrix changes, the controller needs to recalculate routing in order to optimize performance. Consider the example in Fig. 1, where there are six switches $R_1, \ldots, R_6$, and the link capacity is 1 unit. F_A and F_B are two flows from R_1 to R_5, whose demands are 0.2 unit and 1 unit, respectively. F_C is a flow from R_3 to R_5, whose demand is 0.8 unit. The initial routing is illustrated in Fig. 1(a). At this point, suppose a new flow appears from R_3 to R_4 with a demand of 1 unit. The controller then wants to change routing to Fig. 1(b). Due to asynchronous update, the three flows may be routed temporarily as in Fig. 1(c) during the transition. In this case congestion occurs at the link from R_2 to R_5, which is overloaded with twice its capacity, and results in severe packet drops.

Introducing intermediate stages can reduce transient congestion [19], [30]. For example, zUpdate [30] takes advantage of vacant link capacity to find a congestion-free update plan. It first moves 80% of F_B onto path $\langle R_1, R_6, R_5 \rangle$ and keeps the remaining 20%. Then F_A is moved from $\langle R_1, R_2, R_5 \rangle$ to $\langle R_1, R_6, R_5 \rangle$. Finally the remaining 20% of F_B and all F_C is moved to their final paths. The process has four intermediate stages and involves solving four LPs, one for each stage. Yet a congestion-free update plan may not always exist especially when the number of flows is large. Thus one has to set aside some capacity on each link to guarantee its existence, as proved in SWAN [19], and resource utilization is reduced. In this case, F_B with 1 unit demand cannot be completely satisfied through path $\langle R_1, R_2, R_5 \rangle$. It must be split onto different paths. Flow splitting may not be feasible for certain applications that are sensitive to TCP packet reordering, such as video applications.

TABLE 2
Congestion during different transition plan

	Transition Plan	Congestion
1	Fig. 1(a) → Fig. 1(b)	1.0
2	Fig. 1(a) → Fig. 1(g) → Fig. 1(h) → Fig. 1(b)	0
3	Fig. 1(a) → Fig. 1(d) → Fig. 1(b)	0.2

The first line represents the transition plan that updates from initial stage to final stage directly, without intermediate stages involvement. The second line represents congestion-free update plan. The third line represents minimizing transient congestion update plan with one intermediate stage.

In contrast, we intend to find an update plan that admits a small extent of transient congestion. Assume F_A, F_B and F_C are three unsplittable flows. They should be routed only through a single path during update. If we consider the update plan where routing is first changed from Fig. 1(a) to Fig. 1(d), and then to Fig. 1(b), transient congestion can be reduced significantly. Specifically, transitioning from Fig. 1(a) to Fig. 1(d) only causes the link capacity to exceed by 0.2, with two possible transient states shown in Fig. 1(e) and Fig. 1(f), and the transition from Fig. 1(d) to Fig. 1(b) is congestion-free. So the overall transient congestion is 0.2. This update plan may be acceptable in practice because switches have buffers to accommodate traffic bursts, and data center transports such as DCTCP can detect congestion early on with ECN to adjust sending rate in a fine granularity [7]. Further many applications, such as data processing frameworks, are elastic to bandwidth and can tolerate temporary rate reduction [29].

Moreover, the problem of finding an update plan that minimizes transient congestion is more general than finding a congestion-free plan. We expose the tradeoff between update speed and congestion, reducing the number of LPs that need to be solved and allowing the operator to speed up the update process. In the motivating example, there is only one intermediate stage which requires solving just one LP. This is more important in large-scale networks which makes the LP computationally expensive to solve.

3.2 Network Model and Problem Definitions

Before presenting the problem definitions, we first discuss our network model. A network is a directed graph $G = (V, E)$, where V is the set of switches and E the set of links with capacities C_e for each link $e \in E$. We

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2016.2631519, IEEE Transactions on Services Computing

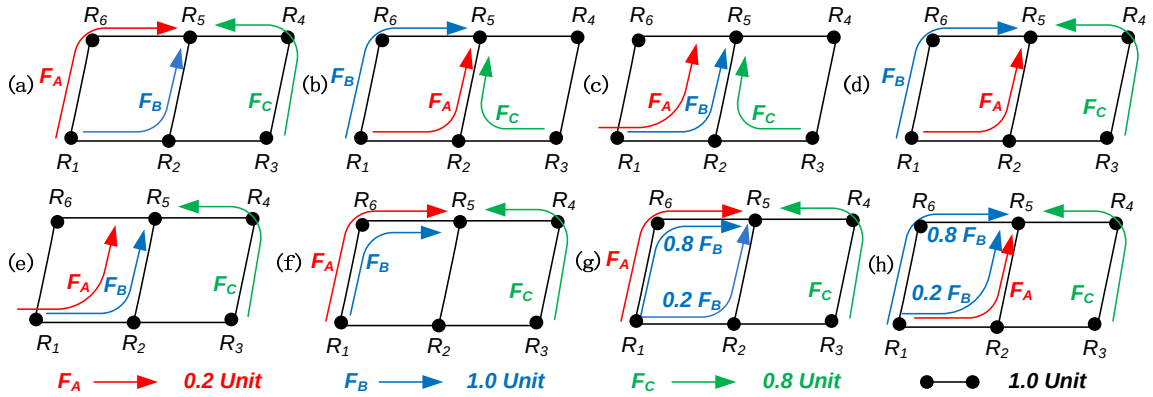


Fig. 1. An example of network update where transient congestion may happen.

TABLE 3
Key notations in this paper.

F_{sp}	The set of flows routed with single path
F_{mp}	The set of flows routed with multipath
F	The set of flows $F = F_{sp} \cup F_{mp}$
V	The set of switches v
E	The set of links e
G	The directed network graph $G = (V, E)$
S	The set of stages the routing update is performed
C_e	The capacity of link e
$P(f)$	The set of possible paths for flow f
d^f	The demand of flow f
n	The number of update stages. $n = \|S\|$
ρ	The number of intermediate stages. $\rho = \|S\| - 2$
k	The number of switches in the network. $k = \|V\|$
σ	The maximum transient congestion.
μ_e^s	The maximum link congestion relative to its capacity during update from stage s to $s+1$.

divided flows in the network into two categories, in accordance to application requirements. F_{sp} represents the set of unsplittable flows that must use single path routing during update; F_{mp} represents the set of flows that can use multipath routing. Each flow f is associated with a demand d^f, routed through a possible path $p \in P(f)$ between its source and destination. For convenience, we summarize important notations in Table 3.

Problem 1. **Minimum Congestion Update Problem (MCUP)**
The set of stages is $S = \{1, 2, \ldots, n-1, n\}$, in which stage 1 and stage n are initial and final stage, respectively. Routing in stage 1 and stage n are known while routing in stages $2, 3, \ldots, n-1$ needs to be determined. The number of intermediate stages ρ is specified by the network operator. The operator may obtain this based on the history of update data and its global view of the network state, which is beyond the scope of this paper. We find routing for each intermediate stage that minimizes the transient congestion.

Problem 2. **Bounded Congestion Update Problem (BCUP)**
The maximum level of transient congestion σ is given. We need to determine the minimum number of intermediate stages ρ and corresponding routing at each stage such that the transient congestion does not exceed σ.

The path set $p \in P(f)$ is pre-computed such that all paths are loop-free. It is true that $P(f)$ would be of exponential size if it contains all possible paths. For fat-tree topologies used in our DCN scenarios, calculating all possible paths can be done in polynomial time. As proved in [6], there are $\left(\frac{k}{2}\right)^2$ paths between each source destination switch pair in the edge level for a k-pod fat-tree. For WAN topologies, we calculate edge-disjoint paths for each source destination switch pair. To further lower the complexity, it is common to use a subset of edge-disjoint paths as the input of TE, such as SWAN [19] and Dionysus [22]. Note that paths of different source destination pairs may use common edges. The resulting path set $P(f)$ are the input of our algorithm. In addition, we assume the two-phase commit protocol proposed in [46] is used to maintain *packet coherence*, i.e., each packet is forwarded either by the old routing prior to the update, or the new routing after the update, but never a mixture of the two.

3.3 Problem Formulation

Based on the above model, we formulate MCUP, i.e. minimum congestion update problem, as a mixed integer linear program (1). We seek to find the optimal routing for all intermediate stages that minimizes the transient congestion from the initial stage to the final stage.

$$\text{minimize} \quad \max_{e \in E, s \in \{1,2,\ldots,n-1\}} \mu_e^s \tag{1}$$

$$\text{subject to} \quad \sum_{f \in F_{sp} \cup F_{mp}} d^f \sum_{p \in P(f): e \in p} \max(x_{f,p}^s, x_{f,p}^{s+1}) \le \mu_e^s C_e,$$
$$\forall e \in E, \forall s \in S - \{n\}, \tag{1a}$$
$$\sum_{p \in P(f)} x_{f,p}^s = 1,$$
$$\forall f \in F_{sp} \cup F_{mp}, \forall s \in S - \{1, n\}, \tag{1b}$$
$$x_{f,p}^s \in \{0, 1\},$$
$$\forall f \in F_{sp}, \forall p \in P(f), \forall s \in S - \{1, n\}, \tag{1c}$$
$$x_{f,p}^s \ge 0,$$
$$\forall f \in F_{mp}, \forall p \in P(f), \forall s \in S - \{1, n\}, \tag{1d}$$
$$\mu_e^s > 0, \forall e \in E, \forall s \in S - \{n\}. \tag{1e}$$

We define transient congestion as the maximum link congestion relative to its capacity μ_e^s during the update

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2016.2631519, IEEE Transactions on Services Computing

across the network, as shown in the objective function of (1). The optimization variable $x^s_{f,p}$ indicates whether flow f is routed through path p in stage s. Constraint (1a) characterizes transient congestion for link e during transition. For example, as illustrated in Fig. 1, during the transition from Fig. 1(a) to Fig. 1(b), *i.e.*, from stage 1 to stage 2, the maximum load of the link (R_2, R_5) is $0.2 \times \max(0,1) + 1.0 \times \max(1,0) + 0.8 \times \max(0,1) = 2$, which describes the case shown in Fig. 1(c). Constraint (1b) is the flow demand conservation constraint. Constraint (1c) represents the single path routing constraint for unsplittable flows, and (1d) represents the multipath routing constraint for splittable flows.

Now we formulate the bounded congestion update problem (BCUP) as an optimization program. The goal is to find the minimum number of intermediate stages such that maximum transient congestion is no larger than σ.

$$\text{minimize} \quad |S| \tag{2}$$

$$\text{subject to} \sum_{f \in F_{sp} \cup F_{mp}} d^f \sum_{p \in P(f): e \in p} \max(x^s_{f,p}, x^{s+1}_{f,p}) \le \sigma C_e, \quad \forall e \in E, \forall s \in S - \{n\}, \tag{2a}$$

$$(1b), (1c), (1d), (1e).$$

The formulation of the bounded transient congestion update problem is shown in (2). The objective aims to minimize the number of elements in set S. The optimization variables $\{x^s_{f,p}\}$ are the same as MCUP (1). Constraint (2a) characterizes that the load of link e cannot be larger than $\sigma \cdot C_e$ during transition.

Because of the max function, constraints (1a) and (2a) in program (1) and (2) are not linear. By introducing auxiliary variables $\{y^s_{f,p}\}$ and $\{z^s_{f,p}\}$, we can transform the constraints to the following linear constraints. The auxiliary variable $y^s_{f,p}$ ($z^s_{f,p}$) is equal to one when unsplittable (splittable) flow f is routed through path p either in stage s or $s+1$, and equals zero otherwise.

$$\sum_{f \in F_{sp}} d^f \sum_{p \in P(f): e \in p} y^s_{f,p} + \sum_{f \in F_{mp}} d^f \sum_{p \in P(f): e \in p} z^s_{f,p} \le \mu^s_e C_e, \quad \forall e \in E, \forall s \in \{1,2,\dots,n-1\}, \tag{3a}$$

$$y^s_{f,p} \ge x^s_{f,p}, \quad \forall f \in F_{sp}, \tag{3b}$$

$$y^s_{f,p} \ge x^{s+1}_{f,p}, \quad \forall f \in F_{sp}, \tag{3c}$$

$$z^s_{f,p} \ge x^s_{f,p}, \quad \forall f \in F_{mp}, \tag{3d}$$

$$z^s_{f,p} \ge x^{s+1}_{f,p}, \quad \forall f \in F_{mp}, \tag{3e}$$

$$\tag{3f}$$

3.4 Hardness Analysis

We establish the hardness of MCUP and BCUP below.

Theorem 1. MCUP is NP-hard.

Proof: Consider a special case of MCUP with only one intermediate stage. We construct a polynomial reduction from the set partition problem [13] to it. Consider a partition instance $\mathbb{A}$ consisting of m items, each with a value a_i, $a_i \in \mathbb{R}, i \in \{1,2,\dots,m\}$. The objective is to partition $\mathbb{A}$ into two subsets $\mathbb{A}_1$ and $\mathbb{A}_2$ ($\mathbb{A}_1 \cup \mathbb{A}_2 = \mathbb{A}$ and $\mathbb{A}_1 \cap \mathbb{A}_2 = \emptyset$)

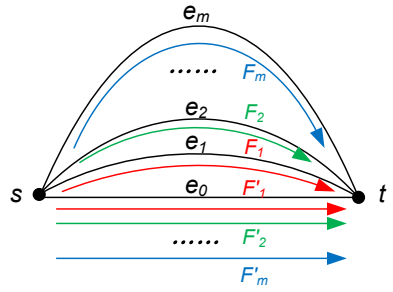


Fig. 2. Reduction from Partition to MCUP.

such that $|A_1 - A_2|$ is minimized, where A_1 and A_2 denote the sums of the elements in each of the two subsets $\mathbb{A}_1$ and $\mathbb{A}_2$. Accordingly, for each item in set $\mathbb{A}$ we introduce two flows F_i and F'_i with demands $d_{F_i} = d_{F'_i} = a_i$. There are m items in total and thus we introduce $2m$ flows. The instance of MCUP is constructed as shown in Fig. 2. There are multiple edges between the source node s and destination node t as shown in Fig. 2. Each edge represents a path in the real network. We ignore the intermediate nodes for simplicity. There are $2m$ flows from source s to destination t in the initial stage, in which flows $F_1, F_2, ..., F_m$ are routed through links $e_1, e_2, \dots, e_m$, respectively, and flows $F'_1, F'_2, \dots, F'_m$ are routed through a single link e_0. The final stage is that flows $F'_1, F'_2, \dots, F'_m$ are routed through links $e_1, e_2, ..., e_m$ and flows $F_1, F_2, ..., F_m$ are routed through link e_0. Note that the flow swap operation between F_i and F'_i captures the definition of transient congestion described in constraint (1a) and the transient congestion only happens during the transition. . The link capacities are $C_{e_i} = a_i$, and $C_{e_0} = \sum_{i=1}^m a_i$ for all $i \in \{1,2,\dots,m\}$. None of the flows can be split during update. Therefore, any partition with minimum difference between set A_1 and A_2 corresponds to MCUP with only one intermediate stage (the partition results indicate that which flows should be swapped firstly and which flows swapped secondly), and vice versa.

□

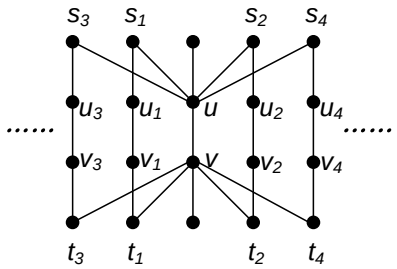


Fig. 3. Reduction from Bin Packing to BCUP.

Theorem 2. BCUP is NP-hard.

Proof: Given maximum transient congestion σ ($1 < \sigma < 2$), we reduce from bin packing [50] to our problem. Consider a set $I = \{1,2,\dots,m\}$ of items, where item $i \in I$ has size a_i, $a_i \in \mathbb{R}$ and a set $B = \{1,2,\dots,m\}$ of bins with capacity b, where $b = (\sigma - 1)C_{u,v}$. For each item we introduce two flows F_i and F'_i with demands $d_{F_i} = d_{F'_i} = a_i$ and the instance of BCUP is constructed as shown in Fig. 3. In the initial stage, flows F_i and F'_i

are routed through two disjoint paths: $\langle s_i, u_i, v_i, t_i\rangle$ and $\langle s_i, u, v, t_i\rangle$. All F_i' have the common link $\langle u, v\rangle$. The final stage swaps F_i and F_i' for each i and all F_i have the common link $\langle u, v\rangle$. All the flows cannot be split during update. We need to take advantage of slack capacity $(\sigma-1)C_{u,v}$, which is equal to the bin's capacity in the common link $\langle u, v\rangle$ to perform swap operations for flows F_i and F_i'. Therefore, the problem of finding an assignment $I \rightarrow B$ such that the number of non-empty bins is minimal is equivalent to the bounded congestion update problem. The number of non-empty bins corresponds to the number of introduced intermediate stages. The items in each non-empty bin are equivalent to the swapped flows in each stage. □

4 A Rounding Algorithm

We now design a rounding based approximation algorithm [50] to tackle the NP-hard MCUP (1).

Algorithm 1 Randomized Rounding

Input: The optimal fractional solution $\{\tilde{x}^s_{f,p}\}$ to the relaxed LP of (1).
Output: A solution $\{\hat{x}^s_{f,p}\}$ to (1).
1: **for** $s = 2$ to $n-1$ **do**
2: **for** each $f \in F_{mp}$ **do**
3: **for** each $p \in P(f)$ **do**
4: $\hat{x}^s_{f,p} = \tilde{x}^s_{f,p}$
5: **end for**
6: **end for**
7: **for** each $f \in F_{sp}$ **do**
8: $P'(f) = \emptyset$
9: **for** each $p \in P(f)$ and $p \notin P'(f)$ **do**
10: $\hat{x}^s_{f,p} = 0$
11: $P'(f) = P'(f) \cup p$
12: $l^s_{f,p} = \sum_{p' \in P'(f)} \tilde{x}^s_{f,p'}$
13: **end for**
14: Generate a number r in (0,1] uniformly at random
15: Find $\hat{p}$ such that $r \leq l^s_{f,\hat{p}}$ and $l^s_{f,\hat{p}} - r$ is minimum
16: $\hat{x}^s_{f,\hat{p}} = 1$
17: **end for**
18: **end for**

The mixed integer program (1) can be relaxed to a linear program by replacing the constraint (1c) $x^s_{f,p} \in \{0,1\}$ with $x^s_{f,p} \geq 0$. Since constraint (1b) holds, $\{x^s_{f,p}\}$ are in fact real numbers between 0 to 1. The optimal fractional solutions $\{\tilde{x}^s_{f,p}\}$ of the relaxed LP can be obtained in polynomial time using standard solvers.

As shown in Algorithm 1, for $f \in F_{mp}$, $\{\tilde{x}^s_{f,p}\}$ is already the feasible solution (lines 2–6). For $f \in F_{sp}$, we apply randomized rounding to obtain an integer solution $\{\hat{x}^s_{f,p}\}$ (lines 7–17). We do not show the process of rounding auxiliary variables $\{\tilde{y}^s_{f,p|f\in F_{sp}}\}$, which can be readily obtained from the integer solutions $\{\hat{x}^s_{f,p|f\in F_{sp}}\}$. To ensure that only one path is chosen for a flow $f \in F_{sp}$ in stage s, the optimal fractional solution can be viewed as partitioning the interval $[0,1]$ to intervals of lengths $\{\tilde{x}^s_{f,p|f\in F_{sp}}\}$ (lines 9–13). A real number is generated uniformly at random in $(0,1]$ and the interval in which it lies determines the path (lines 14–16).

Before analyzing the performance of Algorithm 1, we introduce the following definition.

Definition 1. Let $\tilde{\mu}$ be the optimal fractional solution to (1). Let μ^* be the optimal solution to (1), which gives a lower bound of transient congestion.

Theorem 3. If $\mu^* > 1, \forall e \in E$, Algorithm 1 outputs a feasible solution with transient congestion bounded by $\mathcal{O}(\log k)\mu^*$ from any stage s to $s+1$ with probability $1-\frac{1}{k^2}$, where k is the number of switches in the network.

The proof can be found in Appendix A.

5 A Greedy Improvement Algorithm

In this section we develop a greedy algorithm to improve the solution of the rounding algorithm. In spite of its guaranteed approximation ratio, the randomized algorithm is still not efficient as it may occasionally produce a bad solution. The greedy algorithm improves upon the solution by greedily rerouting each flow to a better path. It has the same approximation ratio as the rounding algorithm in general topologies, and has a constant approximation ratio of 4 in fat-tree topology.

5.1 Algorithm Design

Let us introduce two notations first.

Definition 2. **The $\vee$ operator**: Let $\{\alpha^{s_1}_{f,p}\}$ and $\{\beta^{s_2}_{f,p}\}$ be two routing configurations in stages s_1 and s_2. The result of $\{\alpha^{s_1}_{f,p}\} \vee \{\beta^{s_2}_{f,p}\}$ is a flow distribution $\{D_{f,e}\}$ in network G, where $D_{f,e} = \max(\{\alpha^{s_1}_{f,p}\}, \{\beta^{s_2}_{f,p}\})$. If flow $f \in F_{sp}$, $D_{f,e} \in \{0,1\}$. If flow $f \in F_{mp}$, $D_{f,e} \in [0,1]$.

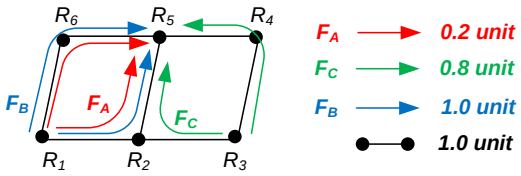


Fig. 4. The result of $\vee$ operator over routing configurations in Fig. 1(a) and Fig. 1(b).

The $\vee$ operator maps routing configurations of different stages onto the same network, which is convenient for further optimization. Fig. 4 shows the $\vee$ operator applied over routing configurations in Fig. 1(a) and Fig. 1(b).

Algorithm 2 Congestion Calculation Function ϕ

Input: Flow distribution $\{D_{f,e}\}$.
Output: Maximal congestion λ.
1: **for** each $e \in E$ **do**
2: $\eta_e, \delta_e = 0$
3: **for** each $f \in F$ **do**
4: **if** $D_{f,e} > 0$ **then**
5: $\eta_e = \eta_e + d^f \cdot D_{f,e}$
6: **end if**
7: **end for**
8: $\delta_e = \frac{\eta_e}{C_e}$
9: **end for**
10: $\lambda = \arg\max_{e\in E} \delta_e$

The congestion calculation function ϕ is rigorously described in Algorithm 2. It takes the result of $\vee$ operator $\{D_{f,e}\}$ as input and calculates the maximal link congestion λ. η_e denotes the load and δ_e the load relative to its capacity in link e (lines 3-8). When Algorithm 2 stops, λ represents the maximal link congestion. Take flow distribution in Fig. 4

as the input of function ϕ, the result is 2.0, which represents the transient congestion transitioning from Fig. 1(a) to Fig. 1(b).

Property 1. For any routing configuration $\{a_{f,p}\}$ in the network, $\phi(\{a_{f,p}\} \vee \{a_{f,p}\}) = \phi(\{a_{f,p}\})$.

Property 2. Let $\{a_{f,p}\}$, $\{b_{f,p}\}$ and $\{c_{f,p}\}$ be three routing configurations in the same network. If $\phi(\{a_{f,p}\}) \geq \phi(\{b_{f,p}\})$ and $\phi(\{a_{f,p}\}) \geq \phi(\{c_{f,p}\})$, then $2 \cdot \phi(\{a_{f,p}\}) \geq \phi(\{b_{f,p}\} \vee \{c_{f,p}\})$.

Theorem 4. Let $\{\alpha^1_{f,p}\}$ and $\{\gamma^n_{f,p}\}$ be the initial and final routing configurations. If intermediate routing configurations $\{\beta^2_{f,p}\}, \{\beta^3_{f,p}\}, \ldots, \{\beta^{n-1}_{f,p}\}$ are not optimal, then the maximum transient congestion is greater than or equal to $\frac{1}{n-1} \cdot \phi(\{\alpha^1_{f,p}\} \vee \{\gamma^n_{f,p}\})$.

The proof can be found in Appendix B.

Algorithm 3 Greedy Improvement

Input: The optimal fractional solution $\{\tilde{x}^s_{f,p}\}$ to the relaxed LP of (1).
Output: An optimized solution $\{\hat{x}^s_{f,p}\}$ to (1).

1: Run Algorithm 1 and obtain a solution $\{\hat{x}^s_{f,p}\}$
2: $\lambda = +\infty$
3: **for** $s^* = n-1$ to 2 **do**
4: $\quad \{\beta^{s^*}_{f,p}\} = \{\hat{x}^{s^*}_{f,p}\}$
5: $\quad$ **for** each $f^* \in F_{sp}$ **do**
6: $\quad\quad \{D^*_{f,e}\} = \{\hat{x}^{s^*+1}_{f,p}\} \vee \{\beta^{s^*}_{F-\{f^*\},p}\}$
7: $\quad\quad$ **for** each $p \in P(f^*)$ **do**
8: $\quad\quad\quad \beta^{s^*}_{f^*,p} = 1$
9: $\quad\quad\quad \lambda^* = \left|\phi\left(\{D^*_{f,e}\} \vee \beta^{s^*}_{f^*,p}\right) - \phi\left(\{\tilde{x}^{s^*}_{f,p}\} \vee \{\tilde{x}^{s^*+1}_{f,p}\}\right)\right|$
10: $\quad\quad\quad$ **if** $\lambda^* < \lambda$ **then**
11: $\quad\quad\quad\quad \{\beta^{s^*}_{f,p}\} = \{\beta^{s^*}_{F-\{f^*\},p}\} \vee \beta^{s^*}_{f^*,p}$
12: $\quad\quad\quad\quad \lambda = \lambda^*$
13: $\quad\quad\quad$ **end if**
14: $\quad\quad\quad$ **if** $\lambda^* = \lambda$ and $\beta^{s^*}_{f^*,p} = \hat{x}^{s^*+1}_{f^*,p}$ **then**
15: $\quad\quad\quad\quad \{\beta^{s^*}_{f,p}\} = \{\beta^{s^*}_{F-\{f^*\},p}\} \vee \beta^{s^*}_{f^*,p}$
16: $\quad\quad\quad$ **end if**
17: $\quad\quad$ **end for**
18: $\quad$ **end for**
19: $\quad \{\hat{x}^{s^*}_{f,p}\} = \{\beta^{s^*}_{f,p}\}$
20: **end for**

We are now ready to describe our greedy algorithm shown in Algorithm 3. We first run Algorithm 1 and obtain an initial solution $\{\hat{x}^s_{f,p}\}$ (line 1), λ represent the transitioning congestion and initiated as positive infinity (line 2). We consider intermediate stages $n-1$ to 2 and greedily change the routing configuration $\{\beta^{s^*}_{f,p}\}$ flow by flow to improve transient congestion from the initial stage $\{\hat{x}^1_{f,p}\}$ to the final stage $\{\hat{x}^n_{f,p}\}$ (lines 4-19). $\{\beta^{s^*}_{f,p}\}$ represent the routing of intermediate stage s^* (line 4). For each flow f^*, we first calculate $\{D^*_{f,e}\}$, which is the result of $\vee$ operator over routing in all stages except f^* in stage s^* (line 6). Then we move f^* onto a different potential path p in order to find a better routing (line 8). If the new routing $\beta^{s^*}_{f^*,p}$ for f^* results in less congestion, we update $\{\beta^{s^*}_{f,p}\}$ and λ (lines 10-13). Further, if f^* is routed through the final path p in stage s^*+1 and does not increase congestion, we update $\{\beta^{s^*}_{f,p}\}$ as well (lines 14-16). When all flows are rerouted in stage s^*, we update $\{\hat{x}^{s^*}_{f,p}\}$ and enter the next stage (line 19).

Note that the congestion upper bound of the rounding algorithm is $\mathcal{O}(\log k)$. The greedy algorithm improves upon it whenever possible. Thus its performance is at least as good as that of rounding. We have the following theorem:

Theorem 5. Algorithm 3 achieves an approximation ratio no more than that of Algorithm 1 in a general topology.

5.2 Approximation Ratio for Fat-tree

We now consider a particular DCN topology, fat-tree [6], an example of which is shown in Fig. 9(a). Fat-tree is commonly used in production data centers [5]. We analyze the approximation ratio of Algorithm 3 in a fat-tree.

Definition 3. Let $\{\alpha^1_{f,p}\}$ and $\{\gamma^n_{f,p}\}$ be the initial and final routing.

$$\tilde{\mu} = \max\left(\phi(\{\alpha^1_{f,p}\} \vee \{\tilde{\beta}^2_{f,p}\}), \ldots, \phi(\{\tilde{\beta}^{n-1}_{f,p}\} \vee \{\gamma^n_{f,p}\})\right)$$

$$\mu^* = \max\left(\phi(\{\alpha^1_{f,p}\} \vee \{\hat{\beta}^2_{f,p}\}), \ldots, \phi(\{\hat{\beta}^{n-1}_{f,p}\} \vee \{\gamma^n_{f,p}\})\right)$$

where $\{\tilde{\beta}^2_{f,p}\}, \{\tilde{\beta}^3_{f,p}\}, \ldots, \{\tilde{\beta}^{n-1}_{f,p}\}$ are the optimal fractional intermediate stages and $\{\hat{\beta}^2_{f,p}\}, \{\hat{\beta}^3_{f,p}\}, \ldots, \{\hat{\beta}^{n-1}_{f,p}\}$ are optimal intermediate stages, respectively for a fat-tree. From the definition, we have $\tilde{\mu} \leq \mu^*$.

Lemma 1. Independent of the rerouting order, when rerouting any flow f^* in stage s^* in Algorithm 3, if $\phi(\{D^*_{f,e|e\in p}\} \vee \beta^{s^*}_{f^*,p}) \geq 4\tilde{\mu}$, there must exist another path p' such that $\phi(\{D^*_{f,e|e\in p'}\} \vee \beta^{s^*}_{f^*,p'}) < 4\tilde{\mu}$, where $p, p' \in P(f^*)$ and $p \neq p'$.

Lemma 1 focuses on fat-tree topology. It implies that if transient congestion is greater than 4 times the optimal solution, there must exist another path for a certain flow whose corresponding transient congestion is less than 4 times the optimal solution. The proof can be found in Appendix C. Now we can show that the greedy algorithm has a constant approximation ratio in fat-tree networks.

Theorem 6. Algorithm 3 approximates MCUP in fat-tree networks with a factor of 4.

Proof: By Lemma 1, for any flow f^*, if $\phi(\{D^*_{f,e|e\in p}\} \vee \beta^{s^*}_{f^*,p}) \geq 4\tilde{\mu}$, there must exist another path $p' \in P(f^*)$ such that $\phi(\{D^*_{f,e|e\in p'}\} \vee \beta^{s^*}_{f^*,p'}) < 4\tilde{\mu}$. When all flows have been rerouted in all stages, there must exist intermediate routing configurations $\{\beta^2_{f,p}\}, \{\beta^3_{f,p}\}, \ldots, \{\beta^{n-1}_{f,p}\}$ such that $\max(\phi(\{\alpha^1_{f,p}\} \vee \{\beta^2_{f,p}\}), \phi(\{\beta^2_{f,p}\} \vee \{\beta^3_{f,p}\}), \ldots, \phi(\{\beta^{n-1}_{f,p}\} \vee \{\gamma^n_{f,p}\})) \leq 4 \cdot \max(\phi(\{\alpha^1_{f,p}\} \vee \{\tilde{\beta}^2_{f,p}\}), \phi(\{\tilde{\beta}^2_{f,p}\} \vee \{\tilde{\beta}^3_{f,p}\}), \ldots, \phi(\{\tilde{\beta}^{n-1}_{f,p}\} \vee \{\gamma^n_{f,p}\})) = 4 \cdot \tilde{\mu}$. From Definition 3, we have $4 \cdot \tilde{\mu} \leq 4 \cdot \max(\phi(\{\alpha^1_{f,p}\} \vee \{\hat{\beta}^2_{f,p}\}), \phi(\{\hat{\beta}^2_{f,p}\} \vee \{\hat{\beta}^3_{f,p}\}), \ldots, \phi(\{\hat{\beta}^{n-1}_{f,p}\} \vee \{\gamma^n_{f,p}\})) = 4\mu^*$. Hence, when Algorithm 3 stops, the maximal transient congestion is less than or equal to $4\mu^*$.

□

6 A Heuristic Algorithm for BCUP

In this section we develop a heuristic algorithm to solve BCUP.

We now explain the high level working of Algorithm 4. At first, the number of intermediate stages ρ is zero (line 1) and the transient congestion is calculated assuming the

Algorithm 4 Iterative insertion

Input: Network topology $G = (V, E)$; congestion parameter σ; initial routing $\{x^1_{f,p}\}$ and final routing $\{x^s_{f,p}\}$.
Output: A solution $\{\hat{x}^s_{f,p}\}$.

1: $\rho = 0$
2: **repeat**
3: $\lambda = 0$
4: $count = \rho + 1$
5: **for** $s = 1$ to $count$ **do**
6: **if** $\phi(\{x^s_{f,p}\}, \{x^{s+1}_{f,p}\}) > \sigma$ **then**
7: Apply Algorithm 5 to obtain one intermediate routing $\{\beta_{f,p}\}$ between $\{x^s_{f,p}\}$ and $\{x^{s+1}_{f,p}\}$.
8: **if** $\{\beta_{f,p}\} = \emptyset$ **then**
9: **return**
10: **else**
11: Insert one intermediate stage $\{\beta_{f,p}\}$ between $\{x^s_{f,p}\}$ and $\{x^{s+1}_{f,p}\}$.
12: $\lambda^s = \max\{\phi(\{x^s_{f,p}\} \vee \{\beta_{f,p}\}), \phi(\{\beta_{f,p}\} \vee \{x^{s+1}_{f,p}\}$
13: **if** $\lambda < \lambda^s$ **then**
14: $\lambda = \lambda^s$
15: **end if**
16: $\rho = \rho + 1$
17: **end if**
18: **end if**
19: **end for**
20: **until** $\lambda \leq \sigma$

network is transitioned directly from the initial to the final stage. If transient congestion is larger than σ, an additional stage is inserted between them. The corresponding routing configurations are obtained by Algorithm 5, which is discussed below. Next we sequentially examine every two adjacent stages and try to insert an intermediate stage between them, to further reduce transient congestion (line 7). If the intermediate routing obtained by Algorithm 5 is an empty set, Algorithm 4 stops, which is the case that transient congestion cannot be further reduced to the given congestion threshold σ (lines 8-9). Otherwise, we insert an intermediate stage and recalculate the maximum transient congestion λ^s (lines 11-12). λ records the new maximum transient congestion corresponding to current update sequence (line 14). We iteratively insert intermediate stages until transient congestion is less than or equal to σ (line 20) or the result of Algorithm 5 is an empty set (line 9).

Algorithm 5 Obtain one intermediate stage

Input: Initial routing $\{\alpha_{f,p}\}$; final routing $\{\gamma_{f,p}\}$
Output: The intermediate routing $\{\beta_{f,p}\}$

1: $\lambda = \phi(\{\alpha_{f,p}\} \vee \{\gamma_{f,p}\})$
2: Insert one intermediate stage $\{\beta_{f,p}\}$ between $\{\alpha_{f,p}\}$ and $\{\gamma_{f,p}\}$
3: Apply Algorithm 3 to obtain the solution of $\{\beta_{f,p}\}$.
4: **if** $\max(\phi(\{\alpha_{f,p}\} \vee \{\beta_{f,p}\}), \phi(\{\beta_{f,p}\} \vee \{\gamma_{f,p}\})) < \lambda$ **then**
5: **return** $\{\beta_{f,p}\}$
6: **else**
7: **return** $\emptyset$
8: **end if**

Algorithm 5 describes how to determine the routing of an intermediate stage, which is the special case of Algorithm 3 with only one intermediate stage. In Algorithm 5, we first calculate λ, the congestion when transitioning directly from the initial to the final stage (line 1). If the transient congestion is reduced after we insert one intermediate stage, this intermediate routing is returned (line 5); otherwise, an empty set is returned (line 7). It is important to note that if σ is selected small enough such that the demand of certain unsplittable flow is greater than $\sigma \times C_e$, the algorithm may stop in a certain iteration and the final transient congestion may not be reduced to σ.

7 Large-scale Simulations

We conduct extensive simulations and experiments to evaluate our algorithms. In this section we report our performance evaluation using large-scale simulations. In the next section we present our Mininet implementation results.

7.1 Setup

We consider two large-scale topologies.

- A 8-pod fat-tree [6] for the DCN scenario. The edge and aggregation layers have 64 switches each in all pods. The network has 8 core switches each with 8 10GbE ports, resulting in a non-blocking network.
- A synthetic scale-free topology randomly produced by the `scale_free_graph` function in [3], which is referred to as WAN scenario. There are 100 switches and 586 10 Gbps links in total. This topology is also used in [41].

We consider both single path and multipath routing [9], [16] in the DCN and WAN scenarios. For DCN, we use 64 long-lived flows in the background similar to [30]. For the WAN scenario, we consider tunnel based multipath routing [19]. The concept of flow defined by Openflow is a 10-tuple [37]; thus for each source-destination switch pair, the number of flows can be more than one. In both settings, we leave 5%–10% link capacity vacant on each link for SWAN. Flows in the network are generated randomly [2], and we change the flow demand to simulate traffic variations. We calculate the initial and final routing to maximize link utilization given the demand [14].

7.2 Benchmark Schemes

We evaluate the following schemes:

One Shot: Transition directly from the initial to the final stage.

RR: Our randomized routing algorithm as in Algorithm 1.

GI: Our greedy improvement algorithm as in Algorithm 3.

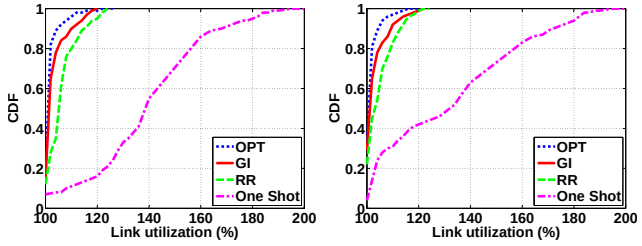
I^2: Our heuristic iterative insertion algorithm as in Algorithm 4.

OPT: The optimal solution of the MCUP integer program (1) obtained using branch and bound.

SWAN: State-of-the-art congestion-free update algorithm [19]. As discussed in §1, this heuristic works by iteratively solving a series of LPs until a congestion-free update plan is found, and does not take the number of intermediate stages as input. Thus it cannot be used to solve (1), and we only include it for comparing the maximum number of rules and update time.

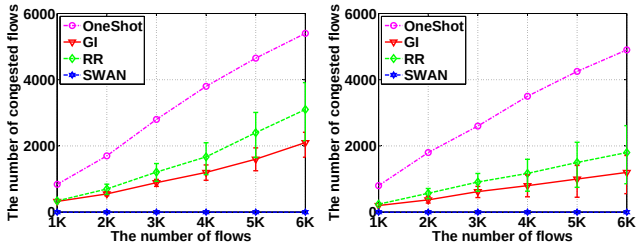
Note that in our simulation, the results of **One Shot** and **SWAN** are deterministic. The algorithms of **RR**, **GI** and **I**^{2} are based on randomization, whose results are the average of at least 10 runs.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2016.2631519, IEEE Transactions on Services Computing



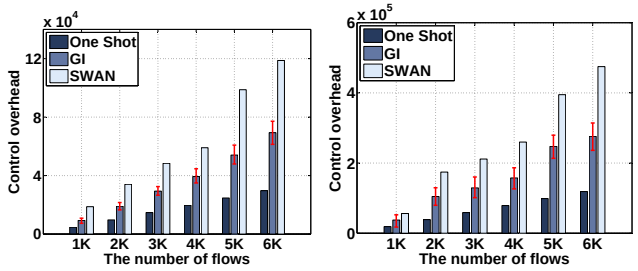
(a) DCN scenario (b) WAN scenario

Fig. 5. Maximum link congestion comparison.



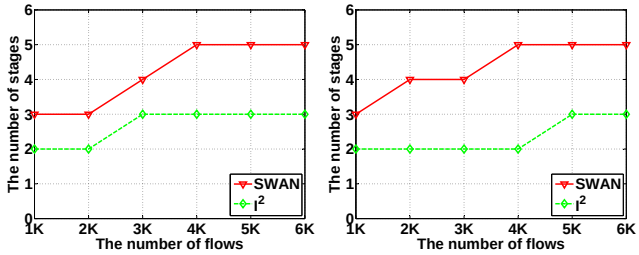
(a) DCN scenario (b) WAN scenario

Fig. 6. The number of congested flows.



(a) DCN scenario (b) WAN scenario

Fig. 7. Control overhead during update.



(a) DCN scenario (b) WAN scenario

Fig. 8. The number of intermediate stages.

7.3 Basic Performance

We first look at algorithms for MCUP. We study the maximum link utilization during update generated by One Shot and the performance of our algorithms—RR and GI—in minimizing congestion comparing to OPT. Fig. 5(a) and Fig. 5(b) show the measured maximum link utilization. Congestion happens when the value of x-axis is larger than 100%, and a larger value indicates severer congestion. For this simulation, we fix the number of flows at 2K for DCN and 4K for WAN. The results of RR, GI and OPT are produced with 3 and 4 intermediate stages for DCN and WAN. Both GI and RR can effectively decrease congestion, particularly between 0.7 and 1.0: GI and RR decrease link congestion by 32.3% and 29.1% respectively compared to One Shot. Furthermore, GI consistently outperforms RR by up to 5%, and provides near-optimal performance compared to OPT.

Fig. 6 shows the number of congested flows during the entire update process. We can see that, as the number of flows increases, One Shot yields significantly more congested flows compared to GI and RR. Specifically, in Fig. 6(a), the number of congested flows for One Shot, RR and GI is 3710, 1670 and 1100, respectively, in the DCN scenario when the number of flows is 4K. Looking more closely into Fig. 6(a) and Fig. 6(b), the improvement for GI is significant: it decreases the number of congested flows by 23% from RR on average. This demonstrates that GI takes full advantage of the richly connected network topology and significantly mitigates congestion by rerouting flows onto less congested paths.

Fig. 7 shows the comparison of control overhead during update. We define control overhead as the number of rules that need to be accessed (added/removed/modified) during the update. Essentially this measures the number of operations, as well as the number of flow table entries required to perform the update. One Shot does not introduce intermediate stages and needs the least update operations. We observe that SWAN induces more control overhead than GI. In Fig. 7(b), when the number of flows is 6K, the control overhead of SWAN is almost twice as that of GI. The reason is that SWAN usually takes more stages to transition the state without any congestion. In contrast, GI uses less intermediate stages with a small extent of congestion and saves a lot of update operations. Note that these results become inaccurate for switches that apply longest prefix matching or wild-card rules. However, such rules are increasingly being replaced with exact match rules in SDN [12], [22], [44].

We now focus on the heuristic algorithm I^2 for BCUP. Fig. 8 shows the number of introduced intermediate stages for I^2 and SWAN with different numbers of flows. We set the congestion threshold σ at 1.15. We observe that I^2 can save the number of intermediate stages compared with SWAN. In Fig. 8(b), the number of introduced intermediate stages for I^2 is only 50% of SWAN with 4K flows. In addition, the algorithm of finding the update plan for I^2 and SWAN is different: For I^2, if the final update plan has ρ intermediate stages, I^2 only needs to solve ρ intermediate stages using Algorithm 3; for SWAN, it needs to solve $\frac{\rho(\rho+1)}{2}$ intermediate stages as described in [19] which significantly aggregates the update time overhead.

8 Implementation

Besides simulation, we develop a prototype of our algorithms using Mininet 2.0 [28]. We use Floodlight 1.0 controller [1] running on a PC with an Intel i5-2400 quad-core processor. Switches run Openflow v1.3, and the forwarding rules are installed and updated via Floodlight's REST API.

We now describe how to perform network update in our implementation. The procedure is shown in Algorithm 6. We first obtain a solution using Algorithm 3 or Algorithm 4

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2016.2631519, IEEE Transactions on Services Computing

Algorithm 6 Performing Network Update

Input: Network topology $G = (V, E)$; initial routing $\{x^1_{f,p}\}$ and final routing $\{x^n_{f,p}\}$.
Output: Update sequence of switch rules.

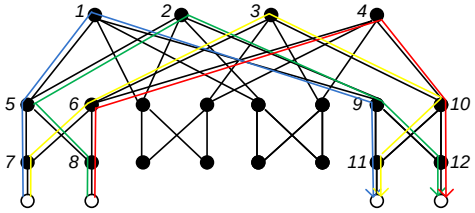
1: Apply Algorithm 3 or Algorithm 4 and obtain solutions $\{\hat{x}^s_{f,p}\}$
2: **for** $s = 1$ to $n-1$ **do**
3: $V' = \emptyset$
4: **for** each $f \in F$ **do**
5: **for** each $p \in P(f)$ **do**
6: **if** $\hat{x}^{s+1}_{f,p} \neq \hat{x}^s_{f,p}$ and $f \in F_{sp}$ **then**
7: **for** each switch v in path p **do**
8: Add new rules to flow table corresponding to flow f in switch v. The new rules use new VLAN tag corresponding to stage $s+1$ to match packets.
9: $V' = V' \cup v$
10: **end for**
11: **end if**
12: **if** $\hat{x}^{s+1}_{f,p} \neq \hat{x}^s_{f,p}$ and $f \in F_{mp}$ **then**
13: For source switch v in path p, change splitting weight to $\frac{\hat{x}^{s+1}_{f,p'}}{\sum_{p' \in P(f)} \hat{x}^{s+1}_{f,p'}}$
14: **end if**
15: **end for**
16: **end for**
17: **if** switch $v \in V'$ is connected by source host **then**
18: Modify the rules in switch v such that it can stamp every incoming packet with a new VLAN tag corresponding to stage $s+1$.
19: **end if**
20: **end for**

(line 1). Next we sequentially examine every stage s of the solution and determine what forwarding rules should be added to which switches by comparing the routing in stage s to stage $s+1$. For a splittable flow, we also need to modify its splitting ratios at its ingress switch (lines 12–14). To ensure consistency, we adopt the two-phase commit protocol proposed in [46], which uses VLAN ID to index stages. In the first phase of transition from s to $s+1$, new rules whose matching fields use the new VLAN ID corresponding to stage $s+1$ (lines 7–10) are added. During this phase, flows are still forwarded according to existing rules as packets are still stamped with the VLAN ID of stage s. Once the update is done for all switches, the protocol enters the second phase when we stamp every incoming packet with the new VLAN ID (lines 17–19). At this point the new rules become functional, and old rules are removed by the controller.

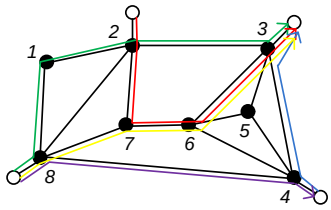
Mininet Setup. We consider two realistic topologies in Mininet shown in Fig. 9. Specifically,

- A 4-pod fat-tree is used for the DCN scenario. The network has 4 core switches and the edge and aggregation layer has 4 switches in each pod. For simplicity, we only use 2 pods in our experiments, which is shown in Fig. 9(a). Each switch has 4 80Mbps ports.
- A realistic WAN topology for interconnecting Microsoft's data centers [22], which is illustrated in Fig. 9(b). There are 8 switches and 14 80Mbps links.

Table 4 shows how multipath routing is done in our implementation using the ingress switch R_8 and egress switch R_3 for the WAN shown in Fig. 9(b) as an example. The



(a) A 4-pod fat-tree DCN topology.



(b) Microsoft's inter-data center WAN topology.

Fig. 9. Realistic network topologies and flows used in our experiments. Each flow is depicted in a different color. Black solid nodes represent switches, while white hollow nodes represent the source and destination hosts that generate test flows.

TABLE 4
Flow table and group table at ingress switch R_8 and egress switch R_3 for splittable flows in Fig. 9(b).

Flow table at R_8

Match Field				Action
InPort	SrcPfx	DstPfx	Tag	
host 1	—	—	—	Gr 1.1

Group table at R_8

Identifier	Type	Action Buckets
Gr 1.1	Select	Weight: 1; Push vlan(0x200); Output: link $\langle R_8, R_1 \rangle$ Weight: 1; Push vlan(0x201); Output: link $\langle R_8, R_7 \rangle$ Weight: 1; Push vlan(0x202); Output: link $\langle R_8, R_4 \rangle$

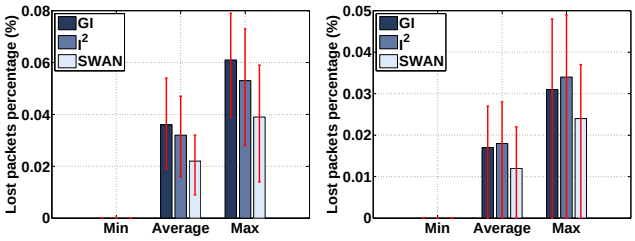
Flow table at R_3

Match Field				Action
InPort	SrcPfx	DstPfx	Tag	
—	—	10.0.0.5	0x200	Pop vlan; Output: host 5
—	—	10.0.0.6	0x200	Pop vlan; Output: host 6
—	—	10.0.0.7	0x201	Pop vlan; Output: host 7
—	—	10.0.0.8	0x201	Pop vlan; Output: host 8
—	—	10.0.0.9	0x202	Pop vlan; Output: host 9
—	—	10.0.0.10	0x202	Pop vlan; Output: host 10

action of R_8 points to the group table Gr 1.1 of type `select`. Gr 1.1 performs multipath routing over three tunnels, and stamps packets with three VLAN IDs. Egress switch R_3 first pops the VLAN header and then forwards the packet according to its destination IP. For simplicity, we do not show the forwarding rules for ARP packets in Table 4. ARP packets are flooded to all output ports.

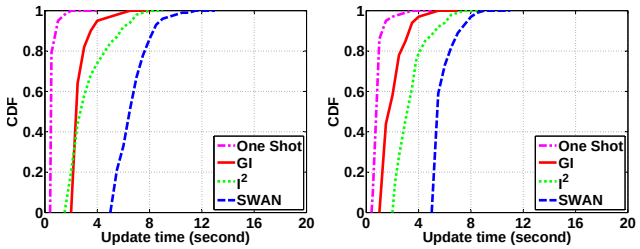
In Fig. 10, we measure packets drop ratio during updates from `iperf`. All the measurements are repeated for at least 30 times. We generate 120 1Mbps flows for a duration of 60 seconds. We leave 25% link capacity vacant for SWAN and the congestion threshold of l^2 is 1.125. The introduced intermediate stages for GI is 3. When all the flows start, we run Floodlight's REST API to modify rules using different update schemes. The x-axis in Fig. 10 represents minimum, maximum and average packet drops among 120 flows. The average packet drop ratio is the result that the total packet

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2016.2631519, IEEE Transactions on Services Computing



(a) DCN scenario (b) WAN scenario

Fig. 10. Percentage of lost packets comparison.



(a) DCN scenario (b) WAN scenario

Fig. 11. Update time comparison.

drops divide the number of flows with packet drops during update. We observe that SWAN, the congestion-free update, still has a small extent of packet drops. The reason is that the hash based flow splitting in OpenvSwitch is imperfect due to the probabilistic nature. The packets drops in Fig. 10(a) is slightly larger than the that of Fig. 10(b). This is because the number of congested links in Fig. 9(a) is larger than that in Fig. 9(b) during update. We also notice that the performance of GI and I^2 are very close to SWAN, because it defines transient congestion for the worst-case scenario during the update, which may not always occur in reality. As shown in Fig. 10(b) for the WAN scenario, the introduced number of intermediate stages is 4 for SWAN, and only 2 for I^2, 3 for GI. Thus our algorithm is able to offer equivalent performance to SWAN without any vacant capacity reserved at links using fewer intermediate stages. Finally we note that One Shot is orders of magnitude longer than I^2, GI and SWAN, and we do not include it here.

In the second experiment, we perform network update to handle device failures in both DCN and WAN. The topologies used here are the same as illustrated in Fig. 9(a) and Fig. 9(b). We generate 200 flows with an average size of 5Mb. We use the `link down` command in Mininet to simulate link failures. We run our algorithms and SWAN, respectively, in the controller to update routing, and measure the total update time T. It includes two parts: time for generating an update plan T_{gen} and time for updating forwarding rules T_{update}. We measure T_{update} using OpenFlow barrier messages [4], which are implemented by floodlight's `OFBarrierRequest` and `OFBarrierReply` class. Specifically, from stages s to $s+1$, we first record the starting time T_s, then send the update messages, and finally send the barrier request message. Upon receiving the barrier response message, we obtain the finish time T_{s+1}. In addition, we measure the average delay between the controller and the switch T_{delay} using the `hello` messages [4], which is subtracted from the calculation.

$$T = T_{gen} + T_{update} = T_{gen} + \sum_{s=1}^{n-1}(T_{s+1} - T_s - T_{delay})$$

Fig. 11 shows the update time results in response to failures. In DCN, 80^{th} percentile updates using GI and I^2 finish within three seconds and five seconds, respectively, while SWAN takes eight seconds. In WAN, GI and I^2 use three seconds and four seconds for 80^{th} percentile updates, while SWAN takes seven. One Shot, as the lower bound of update time, is also implemented in our experiments. T_{gen} is equal to zero for One Shot since it does not require solving LP. We observe that the update time of GI is close to One Shot especially in WAN scenario.

9 Conclusion

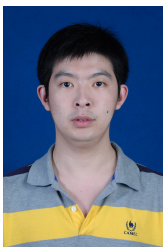
In this paper, we studied congestion-minimizing network update. We proposed two problems of finding an update plan with minimum transient congestion, and finding an update plan with minimum intermediate stages given the transient congestion threshold. We formulated them as optimization programs, and proposed three algorithms to solve the NP-hard problems. Experimental and simulation results show that our algorithms mitigate transient congestion, save control overhead, and reduce update time significantly.

Acknowledgments

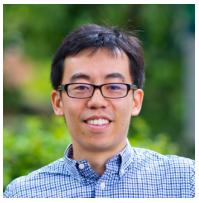
References

[1] Floodlight. http://floodlight.openflowhub.org/.
[2] Fnss. http://fnss.github.io/.
[3] Networkx. https://networkx.github.io/.
[4] Openflow switch specification. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.
[5] A. Agache, R. Deaconescu, and C. Raiciu. Increasing Datacenter Network Utilisation with GRIN. In *Proc. USENIX NSDI*, 2015.
[6] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, pages 63–74, 2008.
[7] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *SIGCOMM*, 2010.
[8] S. Amiri, A. Ludwig, J. Marcinkowski, and S. Schmid. Transiently consistent sdn updates: Being greedy is hard. In *SIROCCO*, 2016.
[9] T. Benson, A. Anand, A. Akella, and M. Zhang. Microte: fine grained traffic engineering for data centers. In *CoNEXT*, page 8, 2011.
[10] S. Brandt, K. Förster, and R. Wattenhofer. Augmenting anycast network flows. In *ICDCN*, pages 24:1–24:10, 2016.
[11] S. Brandt, K.-T. Forster, and R. Wattenhofer. On consistent migration of flows in sdns. In *INFOCOM*, 2016.
[12] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A retrospective on evolving sdn. In *HotSDN*, pages 85–90, 2012.
[13] S. Chopra and M. R. Rao. The partition problem. *Math. Program.*, 59:87–115, 1993.
[14] R. Cohen, L. Lewin-Eytan, J. Naor, and D. Raz. On the effect of forwarding table size on sdn network utilization. In *INFOCOM*, 2014.
[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.
[16] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: scaling flow management for high-performance networks. In *SIGCOMM*, pages 254–265, 2011.
[17] P. Francois, O. Bonaventure, B. Decraene, and P.-A. Coste. Avoiding disruptions during maintenance operations on bgp sessions. *IEEE Transactions on Network and Service Management*, pages 1–11, 2007.

[18] J. Gettys and K. M. Nichols. Bufferbloat: dark buffers in the internet. *Communication of the ACM*, 55(1):57–65, 2012.
[19] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven wan. In *SIGCOMM*, pages 15–26, 2013.
[20] J. Hua, X. Ge, and S. Zhong. Foum: A flow-ordered consistent update mechanism for software-defined networking in adversarial settings. In *INFOCOM*, 2016.
[21] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: experience with a globally-deployed software defined wan. In *SIGCOMM*, pages 3–14, 2013.
[22] X. Jin, H. H. Liu, X. Wu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *SIGCOMM*, pages 539–550, 2014.
[23] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. E. Anderson, and A. Venkataramani. Consensus routing: The internet as a distributed system. (best paper). In *NSDI*, pages 351–364, 2008.
[24] S. Kandula, I. Menache, R. Schwartz, and S. R. Babbula. Calendaring for wide area networks. In *SIGCOMM*, pages 515–526, 2014.
[25] S. Kandula, S. Sengupta, A. G. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *IMC*, pages 202–208, 2009.
[26] N. P. Katta, J. Rexford, and D. Walker. Incremental consistent updates. In *HotSDN*, pages 49–54, 2013.
[27] N. Kushman, S. Kandula, D. Katabi, and B. Maggs. R-bgp: Staying connected in a connected world. In *4th USENIX Symposium on Networked Systems Design and Implementation*, Cambridge, MA, April 2007.
[28] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *HotNets*, page 19, 2010.
[29] N. Laoutaris, M. Sirivianos, X. Yang, and P. Rodriguez. Inter-datacenter bulk transfers with netstitcher. In *SIGCOMM*, 2011.
[30] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. A. Maltz. zupdate: updating data center networks with zero loss. In *SIGCOMM*, pages 411–422, 2013.
[31] A. Ludwig, S. Dudycz, M. Rost, and S. Schmid. Transiently secure network updates. In *SIGMETRICS*, 2016.
[32] A. Ludwig, J. Marcinkowski, and S. Schmid. Scheduling loop-free network updates: It's good to relax! In *PODC*, pages 13–22, 2015.
[33] A. Ludwig, M. Rost, D. Foucard, and S. Schmid. Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies. In *HotNets*, pages 1–7, 2014.
[34] S. Luo, H. Yu, L. Luo, and L. Li. Arrange your network updates as you wish. In *IFIP Networking*, 2016.
[35] J. McClurg, H. Hojjat, P. Cerný, and N. Foster. Efficient synthesis of network updates. In *SIGPLAN*, pages 196–207, 2015.
[36] R. McGeer. A safe, efficient update protocol for openflow networks. In *HotSDN*, 2012.
[37] N. McKeown, T. Anderson, H. Balakrishnan, G. M. Parulkar, L. L. Peterson, J. Rexford, S. Shenker, and J. S. Turner. Openflow: enabling innovation in campus networks. *Computer Communication Review*, 38(2):69–74, 2008.
[38] T. Mizrahi and Y. Moses. Software defined networks: It's about time. In *INFOCOM*, 2016.
[39] T. Mizrahi, O. Rottenstreich, and Y. Moses. Timeflip: Scheduling network updates with timestamp-based TCAM ranges. In *INFOCOM*, pages 2551–2559, 2015.
[40] T. Mizrahi, E. Saat, and Y. Moses. Timed consistent network updates. In *SOSR*, pages 21:1–21:14, 2015.
[41] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti. Officer: A general optimization framework for openflow rule allocation and endpoint policy enforcement. In *INFOCOM*, Apr. 2015.
[42] A. Noyes, T. Warszawski, P. Cerný, and N. Foster. Toward synthesis of network updates. In *SYNT*, pages 8–23, 2014.
[43] Z. A. Qazi, C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simple-fying middlebox policy enforcement using sdn. In *SIGCOMM*, pages 27–38, 2013.
[44] B. Raghavan, M. Casado, T. Koponen, S. Ratnasamy, A. Ghodsi, and S. Shenker. Software-defined internet architecture: decoupling architecture from infrastructure. In *HotNets*, pages 43–48, 2012.
[45] S. Raza, Y. Zhu, and C.-N. Chuah. Graceful network state migrations. *IEEE/ACM Trans. Netw.*, 19(4):1097–1110, 2011.
[46] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *SIGCOMM*, pages 323–334, 2012.
[47] L. Vanbever, S. Vissicchio, C. Pelsser, P. François, and O. Bonaventure. Lossless migrations of link-state igps. *IEEE/ACM Trans. Netw.*, 20(6):1842–1855, 2012.
[48] S. Vissicchio and L. Cittadini. Flip the (flow) table: Fast lightweight policy-preserving sdn updates. In *INFOCOM*, 2016.
[49] W. Wang, W. He, J. Su, and Y. Chen. Cupid: Congestion-free consistent data plane update in software defined networks. In *INFOCOM*, 2016.
[50] D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.
[51] W. Zhou, D. K. Jin, J. Croft, M. Caesar, and P. B. Godfrey. Enforcing customizable consistency properties in software-defined networks. In *NSDI*, pages 73–85, 2015.
[52] X. Zhu, Q. Li, W. Mao, and G. Chen. Online vector scheduling and generalized load balancing. *Journal of Parallel and Distributed Computing*, 74(4):2304–2309, 2014.



Jiaqi Zheng is currently a Ph.D. candidate from Department of Computer Science and Technology, Nanjing University, China. He was a Research Assistant in the City University of Hong Kong in 2015, and a Visiting Scholar in Temple University in 2016. His research interests include data center networks, software defined networks and cloud computing. He received the best paper award from IEEE ICNP 2015.



Hong Xu is an assistant professor the Department of Computer Science, City University of Hong Kong. He received his M.A.Sc. and Ph.D. degrees from the Department of Electrical and Computer Engineering, University of Toronto. His research interests include data center networking, SDN, NFV, and cloud computing. He was the recipient of an Early Career Scheme Grant from the Research Grants Council of the Hong Kong SAR, 2014. He also received the best paper awards from IEEE ICNP 2015 and ACM CoNEXT Student Workshop 2014. He is a member of ACM and IEEE.

Guihai Chen is a distinguished professor of Shanghai Jiao Tong University. He earned BS degree in computer software from Nanjing University in 1984, ME degree in computer applications from Southeast University in 1987, and PhD degree in computer science from the University of Hong Kong in 1997. He has a wide range of research interests with focus on parallel computing, wireless networks, data centers, peer-to-peer computing, high-performance computer architecture and data engineering.

Haipeng Dai is a research assistant in the Department of Computer Science and Technology in Nanjing University, Nanjing, China. He received the B.S. degree in the Department of Electronic Engineering from Shanghai Jiao Tong University, Shanghai, China, in 2010, the Ph.D. degree at the Department of Computer Science and Technology in Nanjing University, Nanjing, China, in 2014. His research interests are mainly in the areas of network measurement, mobile computing, and internet of things.

Jie Wu is the Associate Vice Provost for International Affairs at Temple University. He also serves as the Chair and Laura H. Carnell professor in the Department of Computer and Information Sciences. Prior to joining Temple University, he was a program director at the National Science Foundation and was a distinguished professor at Florida Atlantic University. His current research interests include mobile computing and wireless networks, routing protocols, cloud and green computing, network trust and security, and social network applications. Dr. Wu is a CCF Distinguished Speaker and a Fellow of the IEEE. He is the recipient of the 2011 China Computer Federation (CCF) Overseas Outstanding Achievement Award.