Link-Based Fine Granularity Flow Migration in SDNs to Reduce Packet Loss

Yang Chen and Jie Wu
Center for Networked Computing, Temple University, USA
Email: {yang.chen, jiewu}@temple.edu

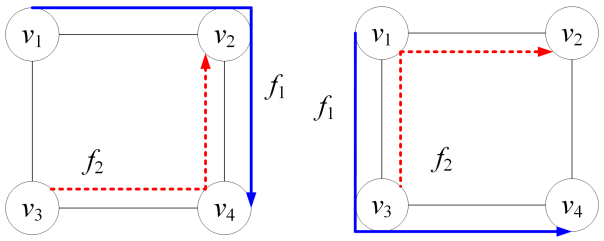
***Abstract*—To maximize the data center network utilization, the SDN control plane needs to frequently update the data plane via flow migration as the network conditions change dynamically. Since each switch updates its flow table independently and asynchronously, the network state transition may result in serious link congestion and packet loss if it is done directly from the initial to the final stage. Deadlocks among flows and links may also block update processes. In this paper, we novelly migrate flows in a finer granularity of links, which is more likely to find a deadlock-free update plan. We prove that it is NP-hard to check the feasibility of a consistent flow migration. We also propose an efficient heuristic for allocating link resources to avoid deadlocks. We show the necessary and sufficient conditions for the deadlock existence in special situations. Extensive simulations show that our solution achieves a much higher probability of a consistent flow migration than prior methods.**

***Index Terms*—Software Defined Networks (SDNs), consistent flow migration, link-based, deadlock.**

I. Introduction

With the development of SDNs [1], there are many causes for a network update: (1) changes in security policies [2] (e.g., traffic from one subnetwork may have to be rerouted via a firewall before entering another subnetwork); (2) traffic engineering in the network [3] (to minimize the maximal link load, an operator may decide to reroute parts of the traffic along different links); (3) network maintenance works [4, 5] (e.g., in order to replace a faulty router, it may be necessary to temporarily reroute traffic); and (4) reactions to link failures [6] (e.g., fast network update mechanisms are required to react quickly to link failures and determine a failover path). Key challenges come from the fact that some unexpected events during the flow migration may happen. These events consist of unpredicted, long switch update times, and abnormal communication delays between the controller and the switch. There are multiple reasons for those chaotic situations, such as the imperfect clock synchronization and the transient controller-data plane disconnection.

However, high performance network requirements are becoming more and more intense [7]. For example, data centers usually claim that packet loss rates to be around 2% [8], while the requirements for wide area networks (WANs) and carrier-grade networks are much higher [9]. Specifically, carrier-grade performance is often associated with the term "five nines", representing an availability of 99.999%. Finding a migration plan without congestion, packet loss and deadlocks is increasingly important. In this paper, we study the consistent flow migration problem, which requires moving network flows from their initial paths to their target ones in a lossless way.



(a) Initial flow routing paths. (b) Final flow routing paths.

Fig. 1: A motivating example.

Fig. 1 is a toy example to illustrate the superiority of a link-based update scheme in avoiding deadlocks. Consider the network in the graphs, in which the nodes represent switches and the edges are bidirectional links. There are two flows, f_1 and f_2, whose initial and final routing paths are shown in Fig. 1a and Fig. 1b, respectively. Assume that each directional link has a capacity of 1 Gbps and each flow has a demand for 1 Gbps bandwidth. All path-based plans require that each link along a flow's final path has enough capacity before migrating the flow from its initial route to the final one. Then neither f_1 nor f_2 can be directly moved. This is because the initial routing path of f_1 overlaps the final routing path of f_2, and vice versa. Such a deadlock is challenging in terms of path-based flow migrations. However, we find that migrating link by link is able to avoid the deadlock. Specifically, each flow requests each link capacity resource along its final path in sequence and releases its occupying link resource of its initial route one-by-one. Our insight is to fully utilize the time difference between occupation and release. We assume that the times to occupy and release one link resource is equal as one time step (TS). In this example, in the first TS, f_1 occupies the link v_1 to v_3 and frees the link v_1 to v_2, while f_2 occupies the link v_3 to v_1 and frees the link v_3 to v_4. In the second TS, f_1 is able to use the link v_3 to v_4, which has been released by f_2 in the first TS. f_2 occupies the link v_1 to v_2, which has been released by f_1 in the first TS. No deadlocks exist. A consistent migration is completed with the fine link granularity scheduling.

In this paper, we study the consistent flow migration problem, and innovatively apply a link-based approach to avoid deadlocks. We extend our previous work [10] to multi-unit flow capacities. We prove the feasibility of a lossless flow migration plan is NP-hard and propose a greedy algorithm

to decide a flow update order. A graph reduction method is introduced to transform the original resource dependency graph into a stuck graph. Then, we prove the necessary and sufficient conditions for the deadlock existence. With unavoidable deadlocks, we try to use an alternative path as an intermediate state to resolve the deadlocks in a lossless way. Extensive simulations show the efficiency and effectiveness of our proposed methods. Because flow splitting may not be feasible for certain applications that are sensitive to TCP packet reordering, such as video applications, we assume all flows are unsplittable in this paper.

We make four main contributions in this paper:

- We analyze current popular network update methods' advantages and disadvantages.
- We propose a link-based update plan to achieve fine granularity network control and resilience to the migration procedure with fewer deadlocks.
- We extensively analyze the necessity and sufficiency of deadlock detection under the unit flow capacities.
- We do a comprehensive performance evaluation of our algorithms in both DCN and WAN scenarios using production topologies.

The remainder of this paper is organized as follows. Section II surveys related works. Section III describes the model and formulates the problem. Section IV introduces the theorems and the flow migration algorithms. Section V includes the experiments. Finally, Section VI concludes the paper.

II. Related Work

There are two basic mainstream methods for flow migration implementations: ordering [11, 12] and two-phase [13, 14]. The ordering strategy usually updates the forwarding tables of switches one-by-one in a specified order that is carefully calculated to preserve required properties, like being loop-free and blackhole-free. However, this order might not exist when both forwarding and policy demands must be guaranteed. The two-phase scheme installs both the initial and final rules on all switches and tags packets with a rule's version number. This method ensures the success of update, but it doubles the number of rules on every switch, which wastes expensive and power-hungry Ternary Content Addressable Memory (TCAM) resource. This paper performs the two-phase commit using version numbers for flow migrations. The major drawback of the basic ordering and two-phase methods of flow migration is that they can not guarantee consistency. Because congestion may exist during flow migrations.

State-of-art work strives to find a path-based, congestion-free update scheme that ensures there will be no congestion independent of the update order. However, most congestion-free update plans require part of the link capacity to be left vacant and decrease utilization of the expensive network infrastructure. As a typical link capacity reservation approach, SWAN [15] has two main results. First, if a fraction s of capacity is guaranteed to be free on each link for the old and new flows, SWAN can update the network in $\lceil 1/s \rceil - 1$ steps. Second, in order to solve the problem efficiently, they use linear programming to check whether a solution with bounded steps exists. However, when there is no slack on some edges, this algorithm may not halt in certain steps, which will lead to a high computation complexity. Moreover, SWAN always involves solving a series of linear programmings (LPs) that is slow and does not scale well. A representative of the intermediate state-involvement approach, ZUpdate [4] attempts to compute and execute a sequence of steps to migrate flows in a congestion-free way. However, it stretches the update time, which makes the chaos of traffic migration last longer. Another kind of time-awareness plan is based on time synchronization technology. The timed consistent strategy [16, 17] utilizes time-triggered network updates to achieve consistency. However, this scheme asks too much of time synchronization. Even with a straggling switch, the whole following process is likely to be in total disorder.

We are aware that there is little network update research on deadlock tackling. Dionysus [18] mentions rate limiting a random number of flows until all the deadlocks are resolved. When a link does not have enough remaining bandwidth for several flows to update at the same time, Dionysus utilizes the migration completion time as the default order of flow priority. This kind of opportunistic scheduling is likely to cause deadlocks where no progress can be made. MCUP [19] proposes a migration approach to minimize transient congestion during the update procedure when a congestion-free update order does not exist, or specifically, when there are deadlocks among update-awaiting flows.

III. Overview

A. Motivation and Background

It is a routine task to update the configurations in SDNs in order to achieve better network performance in failure recovery, transmission latency, and bandwidth utilization. Flow migration is one of the most important and frequent configuration changes. However, deadlocks may frequently occur and cause severe traffic loss even when the current and the desired network flow configurations are both valid and congestion-free. Network administrators do not take flow path overlapping information into consideration when reallocating flow routes. We notice that almost all current research leaves out or looks down upon the problem of avoiding and handling deadlocks. Due to the restrictive demand for low packet loss rates in networks, it is essential to elaborately handle deadlocks and migrate flows while making the best effort to preserve the consistency. An inappropriate migration order of flows can also lead to deadlocks [18]. To the best of our knowledge, our prior work [10] is the first one to focus on the efficient deadlock resolution problem in network update. We show the sufficient condition of the existence of a consistent update. We demonstrate that even if there are multiple consistent migration plans, finding the optimal one that occupies the fewest leisure bandwidth resources is NP-hard. As a result, a greedy algorithm with a theoretical performance guarantee is generated, which is still a path-based flow migration plan.

In this paper, our key observation is that link-based update scheduling works better to avoid deadlocks than the current path-based methods, inspired by the toy example in Fig 1. Since switches are now deeply buffered [20], it is practical to accomplish fine granularity control over the update procedure. Furthermore, we wish to schedule flows in an order that avoids as many deadlocks as possible. To resolve unavoidable deadlocks, we involve the intermediate state in the form of spare paths. Migrating flows to their spare paths can vacate the link resources of initial paths and break the deadlocks. We use the two phase commit protocol to complete the modification of the forwarding table in the switches. With the help of protocol, each packet is stamped with a version number and is routed to the path with the same number. Adding a new path means inserting a new entry with the tag of the new version number in each forwarding table of the switches along the route. In order to guarantee per-packet consistency, the entries with the old version number are only deleted after all the packets with the old version tag arrive at the destination. Considering the limited and expensive Ternary Content-Addressable Memory (TCAM) in the routing tables, it is better to use as few links for the spare paths as possible to cause less redundancy in the network. By migrating fewer flows, we can control the network with less temporary disruption and congestion.

B. Network Model and Problem Formulation

Before formulating the problem, we first present our model of the directed network $G = (V, E)$, where V is a set of vertices (i.e., switches) and $E \subseteq V^2$ is a set of directed edges (i.e., links). We use v_i to denote the i-th vertex and e_{ij} to denote the edge from v_i to v_j. Each edge is capacitated, and we use c_{ij} and r_{ij} to denote the bandwidth of e_{ij} and its remaining capacity respectively. The network G includes a set of the given unsplittable flows F. We use f_k to denote the k-th flow, and its fixed assigned bandwidth during the whole update is d_k, which is equal to its demand. The initial and final routing paths of f_k are denoted by p_k and p'_k, respectively. A path is an ordered set of edges. For example, in Fig. 1, we have $p_1 = \{e_{12}, e_{24}\}$ and $p'_1 = \{e_{13}, e_{34}\}$. e_{24} is the second edge in f_1's initial path, denoted as $p_1(2) = e_{24}$, and its first to second edges are denoted as $p_1(1 \sim 2)$.

This paper studies consistent flow migrations requiring that the bandwidth demands of all flows be satisfied without any link congestion during the entire migration process. A time step-by-step manner is used to migrate flows from their initial to final routing paths. Initially, each flow occupies all the link resources in its initial path, whose amount is equal to its demand. In each time step, each flow can only request one link resource in order along its final path and release one link resource in order along its initial path. Both requesting and releasing start from each flow's first link. The amount that the flow requests and releases is always equal to its demand. If one link request is satisfied, the flow can release one link in its initial path in the same time step. It makes sense that when a flow is migrating from its initial to its final path, it occupies the link resources in order from the head of the final path. At the same time, no more packets are transmitted from the source in its initial path. However, the previous packets along the initial path still need time to complete the transmission. Since we assume the transmission rate is identical, the time it takes a packet to flow through one link is the same. It is reasonable to suppose that when one flow is satisfied with a new assigned link resource along its final path, it will release one link in its initial route. For example, in the first time step, f_1 occupies $p_1(1 \sim 2)$ and requests $p'_1(1) = e_{13}$. e_{13} has enough remaining bandwidth to satisfy f_1's request. Then, f_1 is assigned the request resource and releases $p_1(1)$.

Let b_{ij}^{τ} denote the bandwidth usage of e_{ij} in TS τ, which equals to the total bandwidth demands of its passing flows. We have T time steps in total, i.e., $0 \leq \tau \leq T$. Our problem, extended from our prior work [10], is similar to the Klotski game [21]. Based on the above network model, we formulate the consistent flow migration problem as:

$$\text{minimize} \quad \sum_{e_{ij} \in E} [\max_{0 \leq \tau \leq T} b_{ij}^{\tau}] \tag{1}$$

$$\text{s.t.} \quad b_{ij}^{\tau} \leq c_{ij} \qquad \forall\, 0 \leq \tau \leq T, e_{ij} \in E \tag{2}$$

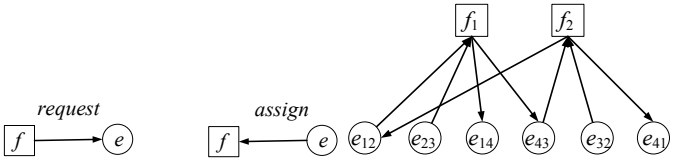
$$\left|\{\, f_k \,|\, p_k^{\tau} \neq p_k^{\tau+1}\}\right| = 1 \qquad \forall\, 0 \leq \tau \leq T \tag{3}$$

$$p_k^0 = \emptyset \text{ and } p_k^T = p'_k \qquad \forall\, f_k \in F \tag{4}$$

In Eq. 1, the maximum bandwidth usage of e_{ij} among all rounds is $\max_{\tau} b_{ij}^{\tau}$. The objective is to minimize the total maximum bandwidth usage among all edges during flow migrations. We aim to use minimum spare bandwidth resources to migrate flows without any link congestion. Two constraints are involved. Eq. 2, meaning that the bandwidth usage of e_{ij}, must be smaller than or equal to its capacity, c_{ij}. Eq. 3 means that only one link of each flow is requested in each TS when migrating its routing path. Note that $\{\, f_k \,|\, p_k^{r} \neq p_k^{\tau+1}\}$ is the set of flows that have changed their routing paths from TS τ to TS $\tau + 1$. Meanwhile, $|\cdot|$ denotes set cardinality. Eqs. 2 and 3 represent the consistency requirement during flow migrations. However, the consistency may not always be satisfied, leading to the feasibility problem. Finally, Eq. 4 represents the constraint that each flow is migrated from its initial paths to final ones.

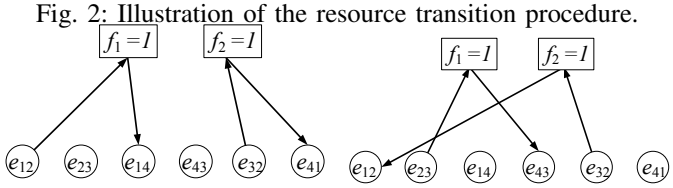
C. Resource Dependency Relationship

The primary challenge is to tractably describe the relationships among flows and resources in the initial and final states. In this paper, we leverage a directed graph to illustrate the dependency relationship. We start with resource dependency in flow migrations. There are two kinds of relationships between flows and links: request and assign. In the Fig. 2a, we use a directed link from the flow node to the link node to represent the request. The reverse represents the assignment. The capacity of the request and assignment is equal to the flow demand, which is the value of the flow node. The link node has the value of its remaining capacity in the current TS. Then we have defined the following:



(a) Flow-link relationship scheme. (b) RDG of a path-based schedule.

Fig. 2: Illustration of the resource transition procedure.



(a) TS 1 of our link-based schedule. (b) TS 2 of our link-based schedule.

Fig. 3: Illustration of the link-based update procedure.

***Definition** 1:* Resource Dependency Graph (RDG) is defined by mapping flows and link resources as nodes and flows' requests and assignments on links as directed edges.

RDG is generated to describe the relationships among flows and links. There are two types of nodes in our dependency graph: link nodes and flow nodes. Link nodes represent link resources and are labelled with the amount of current residue capacity. Flow nodes correspond to different flows marked with demands. Example graphs, which correspond to our toy example in the first section, are shown in Fig. 3. Each link along a flow's initial path is occupied by the flow, unless the link is released. For all links of the final path, one flow is able to request one link in each TS. Each resource node can be requested only after all its predecessor edges have been assigned to flows.

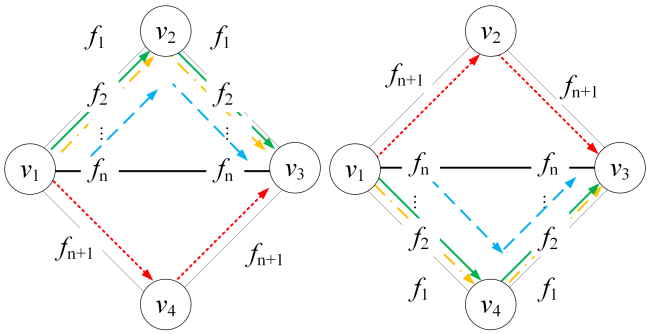
***Definition** 2:* In RDGs, a deadlock is defined as a directed cycle without repeated flow or link nodes and all inside link nodes' remaining capacities are impossible to satisfy any request in the current TS.

In our toy example, with no resource left in e_{12} and e_{43}, neither f_1 nor f_2 is able to be migrated, which is a condition of a deadlock. Fig. 2b is the RDG of a path-based schedule. There is a deadlock among f_1, e_{12}, f_2, and e_{43}. With our link-based schedule, we leverage the time difference among the request and release. Through two TSs in Figs. 3a and 3b, the flows are migrated consistently without any deadlocks because the resources e_{12} and e_{43} are released before their next requests.

First, we study the feasibility problem, i.e., whether a consistent flow migration exists or not with no assistance from spare paths. If it is not feasible, there must be deadlocks among flows and links. We start with the hardness proof:

***Theorem** 1:* The feasibility of a consistent flow migration is NP-hard to check.

Proof: A general proof is introduced in [22]. We can also prove this using a special case of a network consisting of four switches and $n+1$ flows. Our proof is completed using a polynomial reduction from the partition problem [23]. Let us assume a partition problem with n items, each with a random value w_i, $w_i \in \mathbb{R}$, $i \in \{1, 2, ..., m\}$. We assume $\varphi = \sum_{i=1}^{n} w_i$. For each item, we introduce one flow f_i with a demand w_i.



(a) Initial flow routing paths. (b) Final flow routing paths.

Fig. 4: An example to explain NP-hardness.

Algorithm 1 *Link-based Flow Migration Schedule (LifMig)*

Input: Network G and flow set F;
Output: The number of time steps TS;

1: $TS = 0$;
2: RDG =*DependencyGraphGeneration*(G, F);
3: **while** !*MigrationFinish*(RDG) **do**
4: $TS = TS + 1$;
5: EG =*ExpedientGraphReduction*(RDG);
6: (D, DDG) =*DeadlockDetection*(EG);
7: **if** *!empty*(D) **then**
8: DRG =*DeadlockResolution*(DDG);
9: *UpdateGraph*(DRG)
10: **return** TS;

Additionally, there is one more flow f_{n+1} with a demand of φ. We construct a network as shown in Fig. 4. The capacity of all links is 2φ. There are n flows f_1, f_2,..., f_n that have an initial path that is the upper route through v_2. One flow, f_{n+1}, has an initial path that is the lower route through v_4. The target stage is that flows f_1, f_2,..., f_n are routed in the lower path and f_{n+1} is routed in the upper path. Flows are unsplittable. The existence of a feasible consistent flow migration is equivalent to a partition of flows, whose total demands are equal to φ. As a result, any feasible partition of the items corresponds to a feasible consistent flow migration plan, and vice versa. Since the set partition problem is NP-complete and reduces to our problem in polynomial time, our problem is also NP-hard. ■

IV. Link-based flow migration scheme

This section introduces the details of our ***Li***nk-based ***f***low ***Mig***ration scheme, **LifMig**. The process can be accomplished with several update functions, all of which will be discussed in the following. First, we create a dominant algorithm.

The dominant algorithm in Alg. 1 shows how LifMig migrates flows in every time step. Given the network configuration and flow information in the initial and final states, we primarily generate the original resource dependency graph (line 2). Then as long as the migration procedure does not finish, we enter the scheduling phase in a new time step. (lines 4-9). We record the current TS in order to mark the request

Algorithm 2 *DependencyGraphGeneration*

Require: Network G and flow set F;
Ensure: Resource Dependency Graph RDG

1: Initiate a directed graph $RDG = empty$;
2: **for** each link e_{ij} in the network **do**
3: Add a new resource node n with its residue capacity $r_{ij} = c_{ij} - \sum_{e_{ij} \in p_k} d_k$ as the value;
4: **for** each flow $f_k \in F$ **do**
5: Add a new flow node f_k with the value of its demand d_k;
6: Add a directed link from f_k to the first link of p'_k;
7: Add a directed link from the first link of p_k to f_k;
8: **return** Resource Dependency Graph RDG;

position along flows' final paths (line 4). Then we need to elaborately allocate remaining resources in order to reduce the graph to *the stuck state* (line 5). All deadlocks can be detected (line 6) and resolved (lines 7-9) with the stuck graph if they exist. Otherwise, we can continue our next step.

A. Dependency Graph Generation

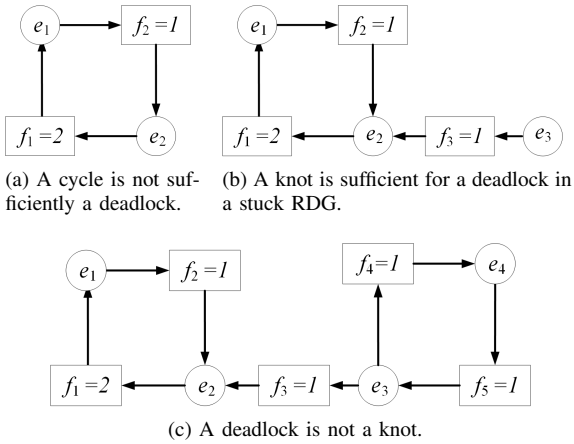
Alg. 2 is used to construct the original dependency graph. First, we add the link nodes with the value of their remaining bandwidth resources (lines 2-3). Then, we add the flow nodes with the value of their demands (line 5). We add a directed link from the flow to the first link resource in its final path (line 6) and a directed link from the first link node in its initial path to the flow node (lines 7-8). The reason for adding only one link from the flow node is that in each TS, we can only release the headmost link in the flow's initial path. Moreover, this simplifies the graph and decreases the complexity of the following processes. We return the accomplished graph.

B. Deadlock Avoidance and Detection

Even with the fine granularity link-based schedule, it is possible to encounter deadlocks. An inappropriate order for the remaining link allocation and the flow migration will lead to deadlocks. Consequently, it is important to avoid unnecessary deadlocks and detect all unavoidable deadlocks if any exists. Although our problem is NP-hard, we have the following theorems derived from [24]:

***Theorem** 2:* A cycle in the RDG is a necessary condition for deadlocks. If the RDG does not include a cycle, a feasible solution can use the topological order of flow nodes in the RDG to migrate flows.

Proof: This theorem is quite straightforward and we can prove it by contradiction. If the graph does not contain a cycle, each flow's request can be satisfied with enough remaining link bandwidth because the network's final state is valid. This means that the bandwidth usage of each link is within its capacity. We can easily find a topological order among the flows. As long as all of a flow's predecessors have released the resource, a flow can be assigned with its request resource and release the link along its initial path. After all requests are satisfied, the system can continue the update. ■



(a) A cycle is not sufficiently a deadlock.
(b) A knot is sufficient for a deadlock in a stuck RDG.
(c) A deadlock is not a knot.

Fig. 5: Illustrating examples of cycle and knot.

Theorem 2 discusses the necessary condition for the feasibility that a cycle is not sufficiently a deadlock. For example, Fig. 5a is a cycle with $c_1 = c_2 = 3$. f_2's request can be satisfied first as there is enough bandwidth in e_2, and it will release its occupying resource in e_1. Then, f_1 can get its request resource of e_1. Thus, the nodes $f_1,\ e_1,\ f_2$, and e_2 form a cycle, but not a deadlock. It is difficult to find the sufficient condition because of the feasibility's NP-hardness. Consequently, we introduce a special state of an RDG and show some interesting properties of it.

***Definition** 3:* An RDG is in the stuck state when all link nodes' remaining capacity fail to satisfy any of the requests in the current TS.

***Definition** 4:* The reachable set of a node a is the set of all nodes b such that a path is directed from a to b. A knot is a non-empty set K of nodes such that the reachable set of each node in K is exactly set K.

***Theorem** 3:* In a stuck RDG, a knot is a sufficient condition for the existence of a deadlock.

Proof: In definition 4, a knot only can have outside nodes request resources in the knot. In the stuck graph, all flow nodes are waiting for resources in the knot, while all edge nodes lack the bandwidth necessary to satisfy any of the requests. The edge nodes in the knot can have their available bandwidth increased only through the release of the flow nodes in the knot since there are no outer links to the nodes that are not in the knot. Thus, there are deadlocks in the knot. Take Fig. 5b as an example. Suppose $c_1 = c_2 = c_3 = 2$ and that the RDG is stuck. The nodes $f_1,\ e_1,\ f_2$, and e_2 form a knot which is also a deadlock. ■

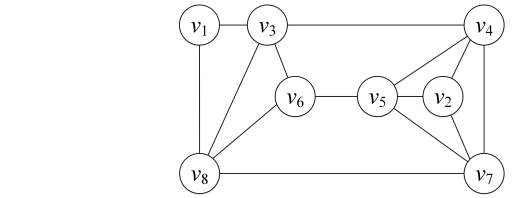
However, a knot is not a necessary condition. Take the RDG in Fig. 5c as a simple counterexample. Suppose $c_1 = c_2 = c_3 = 2$ and $c_4 = 1$. Then, the RDG is in the stuck state. The cycle $e_3,\ f_4,\ e_4$, and f_5 is a deadlock, but not a knot.

***Theorem** 4:* In a stuck RDG with unit demands for all flows, a knot is a necessary and sufficient condition for the existence of a deadlock.

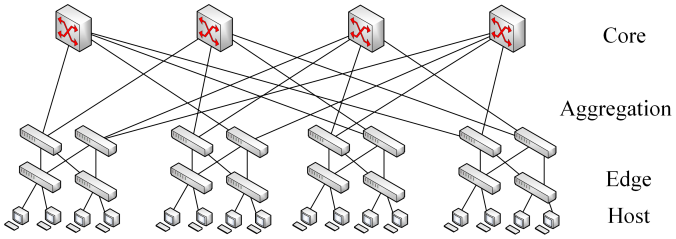
Algorithm 3 ExpedientGraphReduction

Input: Resource Dependency Graph RDG;
Output: Expedient Graph EG;

1: *UpdateSafeFlows(RDG).*
2: Compute $w_k = d_k * \max_{l_k \in L_k} |l_k|$ of each flow f_k involved in cycles;
3: **for** each e_{ij} in any existing cycle of RDG **do**
4: **while** $r_{ij} \geq \min_{f_k \in R_{ij}} d_k$ **do**
5: Find the set S of all flows with the demand less than the remaining capacity;
6: Assign f^* with d^* bandwidth ($w^* = \max w$);
7: $r_{ij} = r_{ij} - d^*$ and delete $f^* \rightarrow e_{ij}$;
8: $r_{kl} = r_{kl} + d^*$ and delete $e_{kl} \rightarrow f^*$ (e_{kl} is the headmost occupying link in p_k).
9: *UpdateSafeFlows(RDG).*
10: **return** Expedient Graph $EG = RDG$;



(a) WAN topology.



(b) Fat-tree topology of data center.

Fig. 6: Topology setting.

Proof: Since Theorem 3 states the sufficient condition proof, we only need to show that when there are only unit demands, a knot is necessary condition for deadlock. In a stuck RDG of all unit flows, there are no edge nodes with a remaining capacity of more than one unit. If the graph does not contain any knot, there is a path from any requesting flow node to a occupying flow node, whose request can be satisfied. If any such flow node exists, the graph is not stuck, which violates our assumption. Hence, when there is no knot, no flow nodes are deadlocked. ■

It is important to first allocate the remaining link resources to reduce the RDG to the stuck state. We propose a greedy heuristic algorithm, Alg. 3, to elaborately allocate resources based on the flow benefit value w. In Alg. 3, we present a graph reduction algorithm to transform an RDG into a stuck state for the following deadlock detection procedure. Alg. 3 updates all the *safe flows*, flows whose father link node has enough remaining bandwidth to satisfy all requests, with the function *UpdateSafeFlows(RDG)*. The function completes the flow's request link assignment and occupying link release by deleting the links of the flow node and renewing its predecessor and successor's value (line1). The details are omitted in this paper. Then we calculate all the leftover flows' benefit values (line2). We use the benefit value $w = d * \max_{l \in L} |l|$ to measure the importance of a flow. L is the set containing all the cycles the flow is involved in, and l is a cycle in L whose size $|l|$ is equal to the number of inside nodes. Lines 3-9 describe the process of exhaustively allocating each cycle-involving edge node's remaining bandwidth to flows sequenced in decreasing order of their benefit values. In each TS, because each flow requests only one link resource, as long as the request is satisfied, the flow is no longer in any cycle. As long as each edge node satisfies all its requests from flows, all directed links towards the node are deleted and the node is not in any cycle. To maximize avoiding the deadlocks, we use the function *UpdateSafeFlows(RDG)* to update some new, safe flows. We do the allocation until there are no cycles or until no more possible assignments exist.

If the EG is a directed acyclic graph (DAG), there must exist the topological order among all the unscheduled flows, and no deadlocks may exist . After the graph reduction to the stuck state, finding deadlocks utilizing the theorem 3 is simple. Since a knot is only a sufficient condition for deadlocks, there are some deadlocks that we do not find.

C. Deadlock Resolution

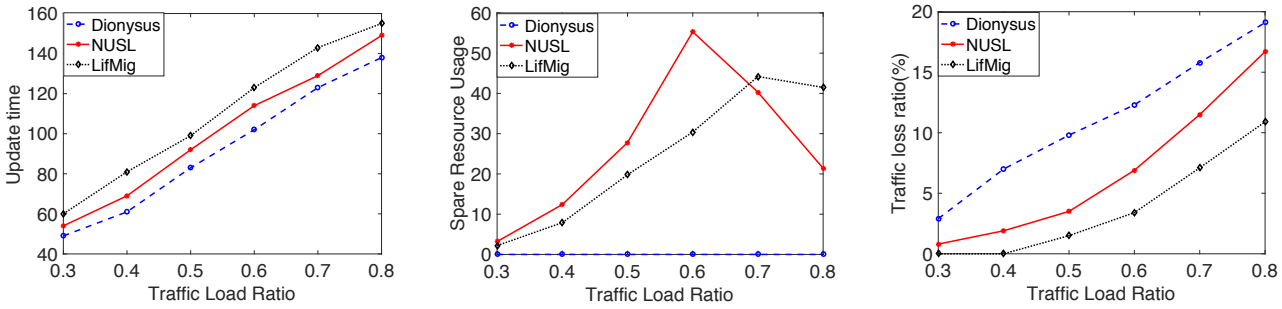
In order to resolve the deadlocks, we primarily utilize the leisure link capacity resources as spare paths to release the links involved in deadlocks. The details of finding and applying the spare paths to break deadlocks are included in our previous work [10]. If there are still unresolved deadlocks, we try to rate limit some flows as discussed in our work [25].

V. Experiment

Simulated experiments are conducted to evaluate the performances of the proposed algorithms. After presenting the network and flow settings, the results are shown from different perspectives to provide insightful conclusions.

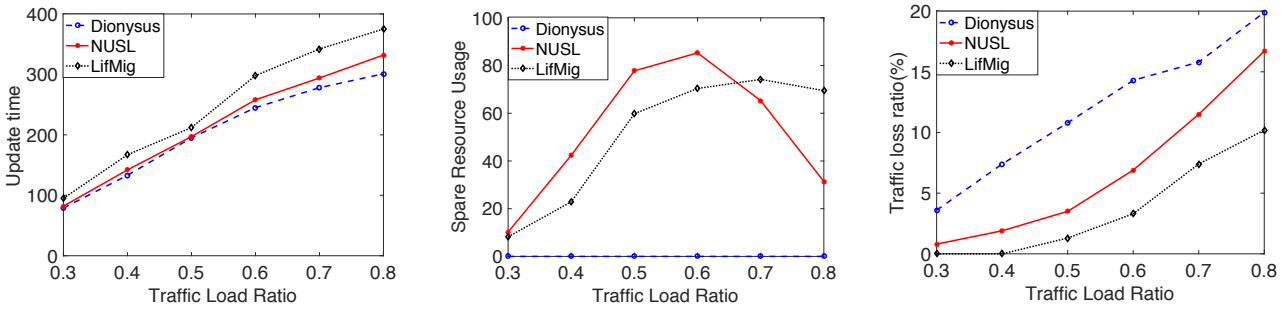
A. Real Traffic Traces and Settings

We evaluate various aspects under two different topologies in Fig. 6a and Fig. 6b. Our experiments study the relationships between the traffic load and three metrics: the number of rate-limiting flows (when a consistent migration plan does not exist), spare resource usage (bandwidth), and traffic loss (the total number of lost packets). We change the traffic load ratio from 30% to 80% to simulate independent variables. Flows in the network are generated randomly at the granularity of 1Mbps. We assume the initial and final states of the network are all valid. There are no more new flows coming into the network during the update. No flow paths have any loops, and each link load is within its capacity.



(a) Updating time. (b) Spare resource usage. (c) Traffic loss ratio.

Fig. 7: Performance in the WAN topology.



(a) Update time. (b) Spare resource usage. (c) Traffic rate ratio.

Fig. 8: Performance in the Fat-tree topology.

B. Benchmark Schemes

There are two comparison algorithms in our simulations: Dionysus [18] and NUSL [10]. Dionysus has already been introduced in our related work section. It builds the dependency graph to describe the relationships among different flow states and uses topological order to migrate flows. As for deadlocks, it opportunistically rate limit flows to zero until all deadlocks are resolved. NUSL is a path-based consistent update strategy that avoids deadlocks with the help of spare paths.

C. Evaluation Results for the WAN

The experimental results in the WAN topology are shown in Fig. 7. Experiments focus on our algorithm's performance with respect to different traffic loads' ratios. Different colors of lines represent the aforementioned mentioned schemes.

Fig 7 shows that our algorithm LifMig achieves a satisfying result compared to Dionysus and NUSL. LifMig limits the fewest flows and maintains the highest throughput with the least traffic loss in Fig. 7(c). When the load ratio is 80%, LifMig limits 44% less flows than Dionysus and 26% less flows than NUSL. LifMig does, however, have a longer update time as shown in Fig. 7(a). It takes about 31% more time than NUSL with a load ratio of 70%. LifMig and NUSL introduce the intermediate state of the update flows. They utilize the leisure bandwidth resource to reduce traffic loss during the update process by migrating some flows to their alternate links. In this way, they vacates some competing link resources to break more deadlocks among flows. As LifMig is link-based, it is more likely to find a consistent update plan. Additionally, it needs only several link resources instead of whole path resources. As a result, LifMig uses fewer spare resources, as shown in Fig. 8(b). It is clear that Dionysus does not introduce intermediate states so its spare resource usage is a constant: 0. As shown in Fig. 8(a), Dionysus leads in extra update time. As long as there is no chaos during the update, it is acceptable to stretch the time when the controller orderly operates flows one by one. Dionysus performs fine, but it does not have a good strategy for breaking deadlocks, and thus experiences more traffic loss than LifMig, as shown in Fig. 7(c). Even in update time, Dionysus has no obvious advantage over our approach.

D. Evaluation Results for the Data Center

The experimental results for the Fat-tree topology in the data centers are shown in Fig. 8. From the figures, we can see that the basic tendencies and relationships of the Fat-tree topology are almost the same as those of the WAN topology. We focus on the differences between Fat-tree and WAN. It should be first noted that the Fat-tree can hold more than five times the flows the WAN topology can hold under the same traffic load ratio. In contrast, the update time, as shown in Fig. 8(a), and traffic loss, as shown in Fig. 8(c), of LifMig is better than in the WAN. This may be because of Fat-tree's excellent load balancing property. The performance of our algorithms is also better than that of the other two. In the update time, the difference between LifMig and NUSL is smaller. The data center is able to achieve a relative faster update because all flows have a fixed path length. Fig. 8(a) reveals there is actually little difference among the three approaches. LifMig more frequently uses spare paths under the Fat-tree topology than WAN, as shown in Fig. 8(b). Because Fat-tree always

distributes traffic in a more balanced manner. It is easier to find spare links to assist in consistently updating flows. What's more, when the traffic is heavy, it is more difficult to find spare paths under the balancing network. However, the number of flows to be migrated to spare links is also much smaller, which indirectly proves the advantage of a regular topology.

Fig. 7 and Fig. 8 show that there is comparatively, less traffic loss in data centers that have the regular topology and balanced traffic. This proves the sensibility of applying a regular topology in the data centers. The number of the flows involved in deadlocks is larger because if one link is busy, it will affect a large number of flows competing for bandwidth resource. It is harder to be intertwined with other flows that are routed in a tidy pattern. It is also more likely to find a feasible update plan using the Fat-tree topology; with the same load ratio, the probability is about 24% higher. Thus, LifMig is well suited for the patterned topology.

VI. Conclusion

The SDN control plane needs to frequently update the data plane via flow migration as network conditions change. In this paper, we creatively propose a link-based flow migration approach that updates the network in a fine granularity. We prove the NP-hardness of the feasibility of finding a consistent flow migration plan. Consequently, we propose an efficient heuristic to allocate the remaining resources to update flows to avoid deadlocks. Several important theorems related to deadlocks are introduced based on the concepts of knot and cycle. Our extensive experiments' results show that our algorithm LifMig can reduce more transient congestions, save more link resources and lose less traffic.

VII. Acknowledgment

This research was supported in part by NSF grants CNS1629746, CNS 1564128, CNS 1449860, CNS 1461932, CNS 1460971, CNS 1439672, CNS 1301774, and ECCS 1231461.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *Proceedings of the 2008 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 2008)*.

[2] A. Ludwig, S. Dudycz, M. Rost, and S. Schmid, "Transiently secure network updates," in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science (SIGMETRICS 2016)*.

[3] V. Jalaparti, I. Bliznets, S. Kandula, B. Lucier, and I. Menache, "Dynamic pricing and traffic engineering for timely inter-datacenter transfers," in *Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 2016)*.

[4] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zupdate: Updating data center networks with zero loss," *Proceedings of the 2013 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 2013)*.

[5] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter, "Traffic engineering with forward fault correction," in *Proceedings of the 2014 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 2014)*.

[6] J. Zheng, H. Xu, X. Zhu, G. Chen, and Y. Geng, "We've got you covered: Failure recovery with backup tunnels in traffic engineering," in *IEEE 24th International Conference on Network Protocols (ICNP 2016)*.

[7] H. Zheng, W. Chang, and J. Wu, "Coverage and distinguishability requirements for traffic flow monitoring systems," in *IEEE IWQoS*, 2016.

[8] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Zhao, and H. Zheng, "Packet-level telemetry in large datacenter networks," in *Proceedings of the 2015 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 2015)*.

[9] T. Mizrahi and Y. Moses, "Time4: Time for sdn," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, 2016.

[10] Y. Chen, H. Zheng, and J. Wu, "Flow migrations in software defined networks: Consistency, feasibility, and optimality," in *the 15th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2017)*.

[11] P. Francois and O. Bonaventure, "Avoiding transient loops during the convergence of link-state routing protocols," *IEEE/ACM Transactions on Networking*, vol. 15, no. 6, 2007.

[12] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure, "Seamless network-wide igp migrations," in *Proceedings of the 2011 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2011.

[13] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent updates for software-defined networks: Change you can believe in!" in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks (HotSDN 2011)*.

[14] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proceedings of the 2012 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2012.

[15] C. Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," in *Proceedings of the 2014 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2014.

[16] T. Mizrahi, E. Saat, and Y. Moses, "Timed consistent network updates in software-defined networks," *IEEE/ACM Transactions on Networking*, 2016.

[17] T. Mizrahi, O. Rottenstreich, and Y. Moses, "Timeflip: Scheduling network updates with timestamp-based tcam ranges," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, 2015.

[18] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," in *Proceedings of the 2014 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 2014)*.

[19] J. Zheng, H. Xu, G. Chen, and H. Dai, "Minimizing transient congestion during network update in data centers," in *2015 IEEE 23rd International Conference on Network Protocols (ICNP)*, 2015.

[20] J. Gettys and K. Nichols, "Bufferbloat: Dark buffers in the internet," *Queue*, vol. 9, no. 11, 2011.

[21] [Online]. Available: https://en.wikipedia.org/wiki/Klotski

[22] S. Brandt, K. T. Frster, and R. Wattenhofer, "On consistent migration of flows in sdns," in *The 35th Annual IEEE International Conference on Computer Communications (INFOCOM 2016)*.

[23] S. Chopra and M. R. Rao, "The partition problem," *Mathematical Programming*, vol. 59, no. 1, 1993.

[24] R. C. Holt, "Some deadlock properties of computer systems," *ACM Computing Surveys (CSUR)*, vol. 4, no. 3, 1972.

[25] Y. Chen and J. Wu, "Max progressive network update," in *IEEE International Conference on Communications (ICC 2017)*.