Efficient and Consistent Flow Update for Software Defined Networks

Kun-Ru Wu, *Member, IEEE*, Jia-Ming Liang, *Member, IEEE*, Sheng-Chieh Lee, and Yu-Chee Tseng, *Fellow, IEEE*

***Abstract*— *Software defined network (SDN)* provides flexible and scalable routing by separating control plane and data plane. With centralized control, SDN has been widely used in traffic engineering, link failure recovery, and load balancing. This work considers the flow update problem, where a set of flows need to be migrated or rearranged due to change of network status. During flow update, *efficiency* and *consistency* are two main challenges. Efficiency refers to how fast these updates are completed, while consistency refers to prevention of *blackholes*, *loops*, and *network congestions* during updates. This paper proposes a scheme that maintains all these properties. It works in four phases. The first phase partitions flows into shorter routing segments to increase update parallelism. The second phase generates a global dependency graph of these segments to be updated. The third phase conducts actual updates and then adjusts dependency graphs accordingly. The last phase deals with deadlocks, if any, and then loops back to phase three if necessary. Through simulations, we validate that our scheme not only ensures freedom of blackholes, loops, congestions, and deadlocks during flow updates, but is also faster than existing schemes.**

***Index Terms*— Computer network, OpenFlow, protocol, SDN, switching, routing.**

I. Introduction

SDN is appealing because it provides flexible and dynamic routing by separating control and data planes. It has a global view on network topology due to its centralized control and is thus widely applied to traffic engineering [1], [2], link failure recovery [3]–[5], and load balancing [6]–[9]. As a network scales up, the frequency of flow updates, where a set of flows need to be migrated to new routes, also increases. During flow update, *efficiency* and *consistency* are two main challenges. Efficiency refers to how fast a set of given flow updates are completed, while consistency refers to prevention of *blackholes*, *loops*, *network congestions*, and *deadlocks* during updates. Blackhole-free means that there is no tentative flow leading to a deadend [10]. Loop-free means that there is no tentative circular route in the network. Congestion-free means the sum of flows on a link should not exceed its link capacity [11]. Since a SDN can control its update order, it is expected to maintain these properties by computing a suitable update schedule.

In previous works, [12]–[15] investigate the optimization of transmission paths in a SDN which needs to update a large amount of flows. References [16] and [17] propose dynamic flow update schemes. In [16], a global dependency graph is generated to dynamically adjust update schedules. However, its update speed is not fast enough. Reference [17] uses critical nodes and local dependency graph to speed up updating, but it incurs congestions in some cases. It also does not have a well-designed mechanism for handling deadlocks.

In this paper, we design a dynamic algorithm for flow update in a SDN by considering both consistency and efficiency. It contains four phases. In the first phase, each flow to be updated is partitioned into multiple segments, if possible, to increase update parallelism. In the second phase, we generate a global dependency graph from these flow segments to construct a global view. In the third phase, we start to update rules of flow segments and adjust the dependency graph dynamically for efficiency while maintaining network consistency. In the last phase, we handle those deadlocks that are not yet solved in the third phase.

The third and the forth phases can be repeated until the desired network state is reached. Our simulation results validate that the proposed algorithm is more efficient than existing schemes and can guarantee consistency properties during the update.

The rest of this paper is organized as follows. Related work is discussed in Section II. The problem statement is in Section III. Section IV presents our proposed algorithm. Simulation results are in Section V. Conclusions are drawn in Section VI.

II. Related Work

In the literature, [1] and [2] present their experiences in building a global SDN network and address dynamic traffic engineering mechanisms. Reference [3] proves that combining SDN with traditional network leads to faster link failures recovery. References [1]–[3] show that the cost of flow updates is a key issue influencing efficiency of SDN and then point out that deciding optimized routing paths is critical as SDN scales up. References [12]–[15] investigate the opti-

Manuscript received September 30, 2017; revised February 5, 2018; accepted February 27, 2018. Date of publication March 12, 2018; date of current version May 21, 2018. This work was supported in part by MOST under Grant 102-2218-E-182-008-MY3, Grant 105-2221-E-182-051, Grant 106-2221-E-182-015-MY3, Grant 105-2745-8-182-001, Grant 106-2221-E-024-004, Grant 105-2221-E-009-100-MY3, Grant 105-2218-E-009-029, Grant 105-2923-E-009-001-MY2, and Grant 104-2221-E-009-113-MY3, in part by the MoE ATU Plan, in part by Delta Electronics, in part by ITRI, in part by the Institute for Information Industry, in part by Academia Sinica under Grant AS-105-TP-A07, and in part by Chang Gung Memorial Hospital, Taoyuan. *(Corresponding authors: Jia-Ming Liang.)*

K.-R. Wu, S.-C. Lee, and Y.-C. Tseng are with the Department of Computer Science, National Chiao Tung University, Hsinchu 30010, Taiwan (e-mail: kunruwu@cs.nctu.edu.tw; sclee840310@cs.nctu.edu.tw; yctseng@cs.nctu.edu.tw).

J.-M. Liang is with the Department of Computer Science and Information Engineering, Chang Gung University, Taoyuan 33302, Taiwan, and also with the Department of General Medicine, Chang Gung Memorial Hospital, Taoyuan 33378, Taiwan (e-mail: jmliang@mail.cgu.edu.tw).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSAC.2018.2815458

0733-8716 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

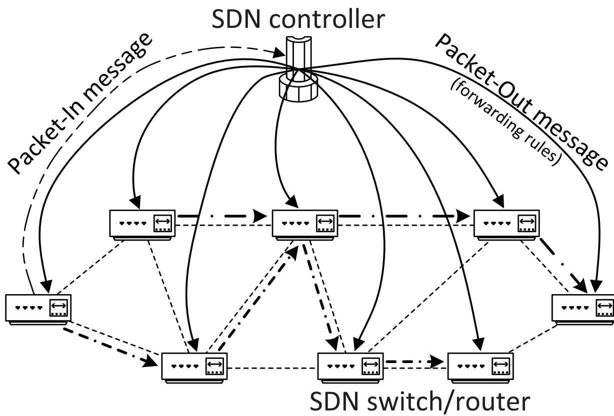


Fig. 1. SDN network architecture.

mization of transmission paths in SDN. However, the above researches do not address the potential data transfer delay during flow updates. Studies [18]–[20] propose a static scheme for scheduling flow updates that can decrease transition delay. However, static schemes arrange update order in advance and the update order can not be adjusted during update, which is not suitable for network with frequent update needs. When encountering compicated and dynamic flow updates, these schemes may encounter the congestion problem. Dynamic flow update methods are proposed in [16] and [17]. Reference [16] uses a global dependency graph to dynamically adjust update schedules and thus prevents the congestion problem. However, it does not perform well in terms of efficiency and may lead to dropping packets during update. Reference [17] improves [16] by identifying some critical nodes that may hurt performance. It generates local dependency graphs based on these critical nodes for improving efficiency. However, it can not ensure consistency during flow update. Transmitted packets may experience loss during update. Besides, in [16] and [17], the way that deadlocks are handled may lead to long latency.

III. Problem Statement

A network architecture can be divided into control plane, data plane, and management plane. Data plane forwards packets by looking up flow tables. Control plane configures and updates flow tables for data plane. Management plane is responsible for monitoring and configuring network devices. In SDN, it separates control plane software and data plane hardware. As shown in Fig. 1, one controller can monitor and manage multiple routers or switches. Network devices thus only need to forward packets. The controller is executed by pure software and commands data planes by protocols such as OpenFlow. In this way, new communication protocols can be more easily realized with little hardware constraint. Therefore, SDN can save hardware upgrade costs and adapt to ever-increasing software needs.

A. Efficient and Consistent Flow Update

A flow update example is shown in Fig. 2, where flow $f_1 = A \rightarrow B \rightarrow E$ in Fig. 2(a.1) needs to be migrated to flow $f_1' = A \rightarrow C \rightarrow D \rightarrow E$ in Fig. 2(a.2). To do so, control plane first sends control messages carrying update rules to switches on target paths. During updates, packets are still being transmitted in the network. Therefore, it is critical to maintain the efficiency and consistency properties. Efficiency refers to how fast a set of given flow updates are completed, while consistency refers to prevention of blackholes, loops, network congestions, and deadlocks during updates.

1) Blackhole problem: The existence of blackholes leads to dropping packets at certain nodes, thus decreasing network throughput. In the example of Fig. 2(a), the network needs to make the following changes: i) deleting the old rules from $switch_A$ and $switch_B$, and ii) adding the new rules to $switch_A$, $switch_C$ and $switch_D$. However, when only partial updates are done, blackholes may appear. In Fig. 2(a.3), the rule in $switch_B$ has been deleted already, but $switch_A$ is not updated yet. Therefore, packets sent to $switch_B$ will be buffered or even dropped. In Fig. 2(a.4), both $switch_A$ and $switch_D$ are updated, but $switch_C$ is not updated yet, so $switch_C$ is a blackhole.

2) Loop Problem: To prevent blackholes, it is straightforward to add new rules first and then delete old rules. However, it might form a loop during the transition. Flow $f_1 = A \rightarrow B \rightarrow E \rightarrow C \rightarrow D$ in Fig. 2(b.1) needs to be updated to $f_1' = A \rightarrow C \rightarrow B \rightarrow E \rightarrow D$ in Fig. 2(b.2). In Fig. 2(b.3), if $switch_A$ and $switch_C$ are updated, but $switch_E$ still uses the old rule. A loop $C \rightarrow B \rightarrow E \rightarrow C$ appears. Note that since the priority of new rule is higher than the priority of old rule, $switch_C$ will send packets to $switch_B$ rather than $switch_D$.

3) Congestion Problem: During updates, no link should experience tentative bandwidth overflow. In Fig. 2(c), there are two flows f_1 and f_2 with rates 0.7 and 0.8 Mbits/s, respectively. Assuming each link capacity to be 1 Mbits/s, if we update f_1 first, the load of $link_{AD}$ will tentatively become $0.7 + 0.8 > 1$ Mbits/s during transition.

4) Deadlock Problem: In Fig. 2(d), flow f_1 and f_2 need to be switched. In this case, each flow waits for the other flow to be removed before it can be migrated. Therefore, none of them can be updated, causing deadlock.

B. Problem Formulation

We consider a SDN with N switches $s_i, 1 \le i \le N$. Switches are connected by a set of links and the link between s_j and s_k is denoted by $l_{j,k}$. The link capacity of $l_{j,k}$ is denoted by $C_{j,k}$. In the network, we are given a set F of flow pairs. Each flow pair $(f_i, f_i') \in F$ implies the need of migrating the transmission flow f_i to a new path f_i'. A flow f_i (or f_i') is written as a sequence $s_{i_1} \rightarrow s_{i_2} \rightarrow s_{i_3} \rightarrow \ldots$, such that $l_{i_j, i_{j+1}}$ is a link. The required transmission bandwidth of both f_i and f_i' is b_i (Mbits/s). During flow update, we have to maintain the four consistency properties. Our goal is to schedule an updating order $\hat{R} = R_1 \rightarrow R_2 \rightarrow \ldots$, where R_i refers to the ith update round. Each round R_i consists of a set of segment pairs denoted by $\{(g_1, g_1'), (g_2, g_2'), \ldots\}$, where each segment pair (g_i, g_i') is a partial flow pair in F and represents migrating g_j to g_j'. We will show how to

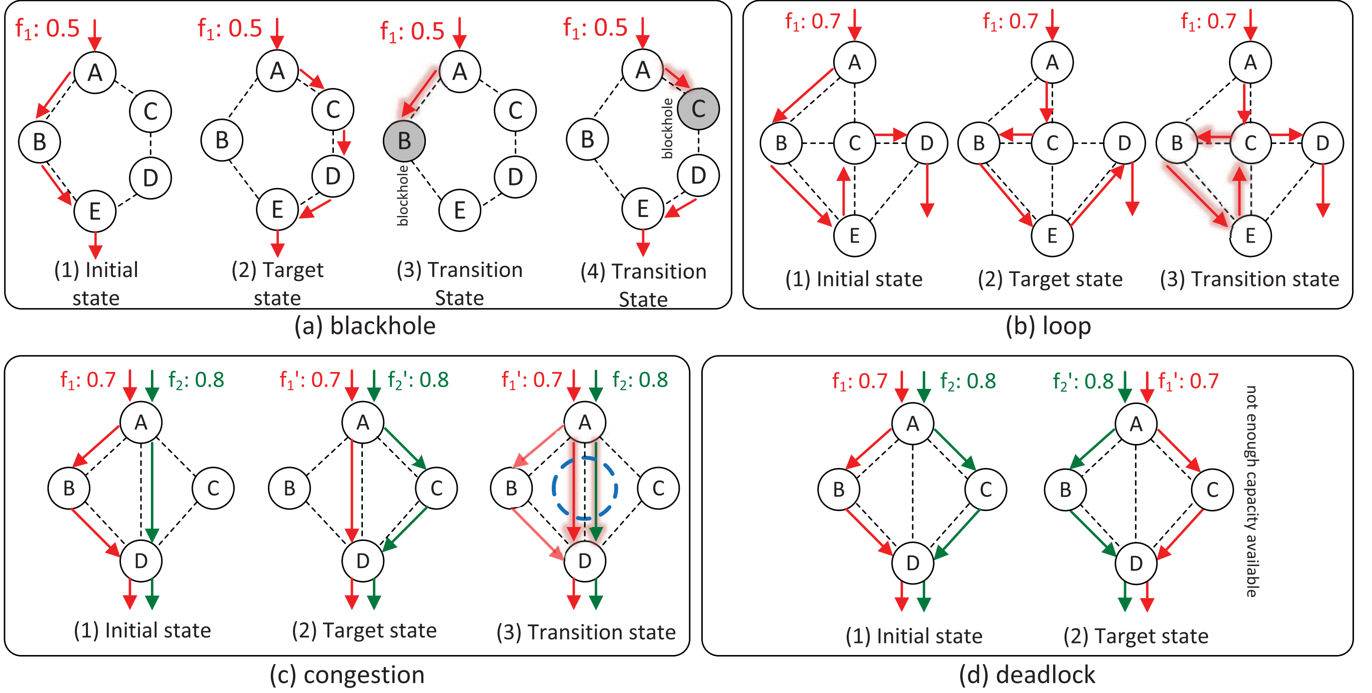


Fig. 2. Blackhole, loop, congestion, and deadlock problems.

partition a flow into segments. All segment pairs in R_i can be updated in parallel. Therefore, the update time of R_i is $T(R_i) = \max_{(g_j, g'_j) \in R_i} \{|g_j| \times t_D + |g'_j| \times t_A\}$, where t_D and t_A are the time of deleting and adding a rule at a switch, respectively. The total update time of $\hat{R}$ is $T(\hat{R}) = \sum_{R_i of \hat{R}} T(R_i)$.

Our goal is to find an update order $\hat{R}$ for updating F with the minimal update time $T(\hat{R})$ while guaranteeing consistency during the whole updating process.

IV. Proposed Scheme

Updating flow f to f' includes adding new rules to f' and deleting old rules from f. However, updating rules on these switches in an arbitrary order may cause problems. To prevent blackholes and loops, [17] proposes a *reverse order update* scheme. The controller adds new rules with reverse order and then deletes old rules with forward order. Taking Fig. 2(b) as an example, rules should be added to f' with the order $s_D \rightarrow s_E \rightarrow s_B \rightarrow s_C \rightarrow s_A$, and be deleted from f with the order $s_A \rightarrow s_B \rightarrow s_E \rightarrow s_C \rightarrow s_D$. For switch s_E, once its new rule is added (which has a higher priority than old rules), the packets transmitted to it via f_1 will be forwarded to s_D by the new rule. Thus, the loop problem in Fig. 2(b.3) will not appear. On the other hand, the blackhole problem in Fig. 2(b.3) is prevented since we add rule at s_D before s_C.

In this work, we adopt the scheme in [17]. Our proposed scheme is shown in Algorithm 1. It mainly contains four functions. Function *ParFlow()* tries to partition flow pairs into segment pairs. We then generate a dependency graph from these segment pairs by function *GenDepGraph()*. Function *GenRnd()* is to schedule a sequence of non-deadlock rounds from $\hat{G}$. If there is no deadlock in the function, it is expected to handled all flow updates. Otherwise, *RsvDead()* is called to resolve one deadlock. Then this is repeated until $\hat{G}$ becomes null.

Algorithm 1

```
1:  S = φ
2:  // Partition flows into segments
3:  for each (f, f') ∈ F do
4:      S = S ⋃ ParFlow(f, f')
5:  end for
6:  // Dependency graphs generation
7:  Ĝ = GenDepGraph(S)
8:  while Ĝ ≠ Null do
9:      // Generate non-deadlock rounds
10:     GenRnd(Ĝ)
11:     // Resolve deadlock
12:     if Ĝ ≠ Null then
13:         RsvDead(Ĝ)
14:     end if
15: end while
```

A. Segmenting Flow Pairs

To increase update parallelism, function $ParFlow(f, f')$ tries to partition a flow pair (f, f') into segment pairs $\{(g_1, g'_1), (g_2, g'_2), \ldots\}$ such that g_j and g'_j are partial paths of f and f', respectively, and g_j and g'_j share the same start and end nodes. Fig. 3(a.1) shows an example, where (f, f') is divided into $\{(g_1, g'_1), (g_2, g'_2)\}$. Both g_1 and g'_1 start at A and end at B; both g_2 and g'_2 start at C and end at D. The common parts (which do not need to be updated) are not included. Fig. 3(a.2) shows a more complicated example, where (f, f') is divided into $\{(g_1, g'_1), (g_2, g'_2), (g_3, g'_3)\}$

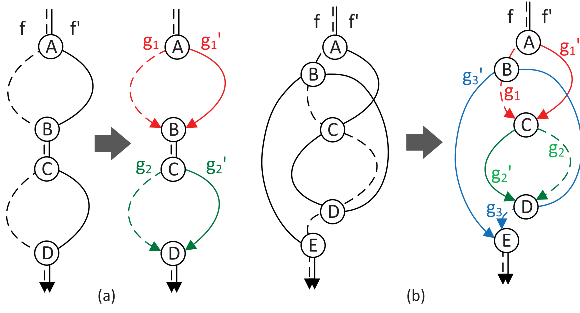


Fig. 3. Examples of partitioning flow pairs into segment pairs.

(to be explained later). Note that except start and end nodes, a segment pair never overlaps.

How to partition (f, f') into segments pairs is shown in Algorithm 2. Set M is initially empty and will contain final segment pairs at the end. Node S is an index to f' that helps us to traverse f'. Function $Start(f')$ returns the first node of f', while $End(f')$ returns the last node of f'.

Algorithm 2 $ParFlow(f, f')$

```
1: M = ∅
2: s = Start(f')
3: while s ≠ End(f') do
4:     if s uses different rule in f and f' then
5:         tail = FindTail(s)
6:         g = sub-flow of f from s to tail
7:         g' = sub-flow of f' from s to tail
8:         M = M ∪ {(g, g')}
9:         s = tail
10:    else
11:        s = Next(s)
12:    end if
13: end while
14: return (M)
```

In the while-loop, we traverse f' in the forward direction using index s. If s tail has different routing rules for f and f', a new segment pair (g, g') starting from s and ending at tail is generated. Here, s is a splitting node that f and f' go into different directions and $FindTail(s)$ returns the node next to s in f' that also appears in the rest part of f. Then we move index s to tail. Otherwise, we move index s to $Next(s)$, the node immediately next to s in f'. At the end, M contains all these generated segment pairs and serves as the returned value of this function.

In Fig. 3(b), $f = A \to B \to C \to D \to E$ and $f' = A \to C \to D \to B \to E$. Since the search is based on node order in f', from A, the next common node of f and f' is C, so the first segment pair is $(g_1, g_1') = (A \to B \to C, A \to C)$. From C, the next pair is $(g_2, g_2') = (C \to D, C \to D)$. From D, the next pair is $(g_3, g_3') = (D \to E, D \to B \to E)$. So totally three pairs are returned.

Note that Algorithm 2 only partitions one flow pair. After each call to $ParFlow()$, the returned set is unioned with set S in Algorithm 1. Finally, S will include all generated segment pairs.

B. Generating Dependency Graph

Next, we derive the dependency relations among segment pairs of S. This helps prevent potential congestions and deadlocks. We will write $l \to (g, g')$ if the migration of g to g' depends on the availability of bandwidth on link l. We will write $(g, g') \to l$ if the migration of g to g' will release some bandwidth to link l.

Our goal is to first generate a dependency graph G and then extend it to an augmented graph $\hat{G}$. From each segment pair $(g, g') \in S$, we construct $m + n + 1$ nodes, where m and n are the numbers of links in g and g', respectively. The (g, g') itself generates one node, called *segment node*, also denoted by (g, g') for convenience. Each link of g and g' generates one node, called *link node*, also denoted by the link's label. This gives $m + n$ link nodes. From each link node l of g', we generate a directed edge $l \to (g, g')$. For each link node of g, we generate a directed edge $(g, g') \to l$. Fig. 4(a) shows an example with 6 segment pairs. Fig. 4(b) shows the subgraphs derived from (g_1, g_1') and (g_2, g_2'). After having all subgraphs, we can combine them into a dependency graph by merging all link nodes belonging to the same link into one node. Fig. 4(c) is the final G after merging all link nodes.

Next, we augment G to $\hat{G}$ by adding information to each link and segment node. A link node $l_{j,k}$ has one extra element.

- $AvlBw$: available bandwidth, which is the total bandwidth of $l_{j,k}$ subtracted by the sum of bandwidths of all flows passing it.

A segment node (g, g') has two extra elements.

- $TxBw$: transmission bandwidth b_i of g and g'.
- $PotUpTm$: Potential update time, which is the longest possible time to update all nodes depending on (g, g').

Fig. 4(d) shows $\hat{G}$.

Next, we show how to calculate $PotUpTm$ for $\hat{G}$. This element reflects the accumulated update time of a dependency sequence. From each segment node r, we construct a breadth-first spanning tree in G with r as root. Then a tentative value $tm(x)$ is computed for each segment/link node x in a bottom-up manner on the tree. If x is a segment node, we set

$$tm(x) = T(x) + \max_{\text{y is a child of x}} \{tm(y)\},$$

where $T(x) = |g| \cdot t_D + |g'| \cdot t_A$ such that $x = (g, g')$. If x is a link node, we set

$$tm(x) = \max_{\text{y is a child of x}} \{tm(y)\}.$$

Intuitively, a segment node will add its update time on itself, while a link node will not. For a leaf node y, if it has no outgoing dependency, its $tm(y) = 0$; otherwise, its $tm(y) =$ 'L', which means "Large value". The label 'L' will remain there when we go up recursively. Finally, $PotUpTm(r)$ is set to $tm(r)$.

In Fig. 5(a), the update time of link node $link_1$ is $max\{tm(B), tm(C)\} = 70$, where B and C are children of $link_1$. If dependency relations form a loop, the leaf node will have its $PotUpTm =$'L'. In Fig. 5(b), node D has 'L', its parent will carry 'L', too. Note that the above process is for

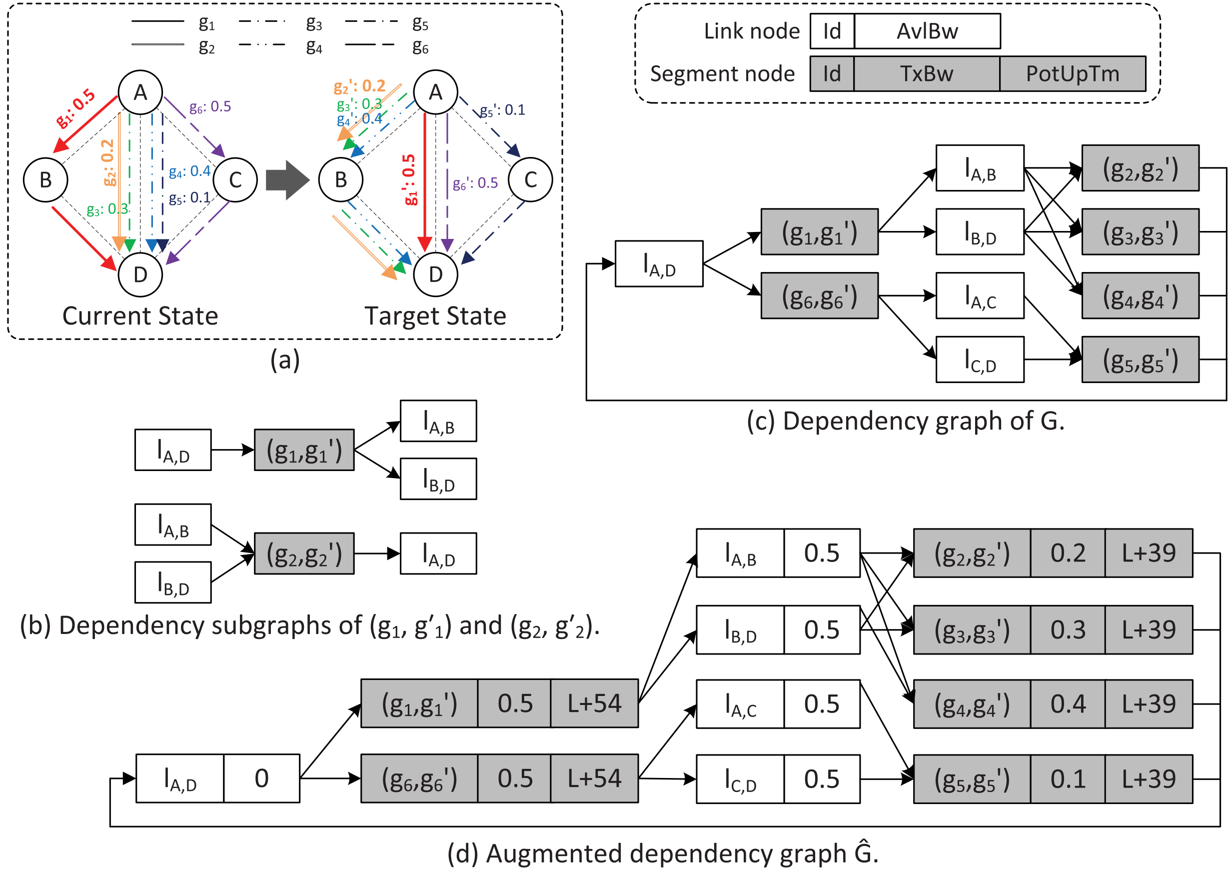


Fig. 4. Dependency graphs G and $\hat{G}$.

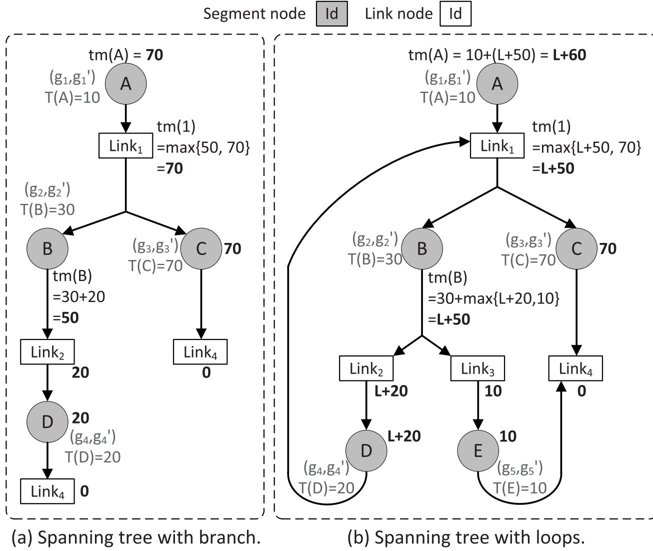


Fig. 5. Examples of calculating $PotUpTm$.

one segment node. It should be repeated for each segment node to find its $PotUpTm$.

C. Generating Non-Deadlock Rounds

Now, $\hat{G}$ contains all dependency information. Algorithm $GenRnd$ is to schedule flow updates round by round until finishing all updates or encountering unsolvable deadlocks. We explain its steps as follows:

- line 1: The while loop will potentially generate a round after each iteration.
- lines 2-3: This is to initialize a new empty round R. Segment nodes are sorted by their $PotUpTm$ in a descending order, and then by their $TxBw$ in a descending order.
- lines 5-10: If all links on g' have enough bandwidths for updating (g, g'), we update their $AvlBw$ and append (g, g') to set R.
- lines 15-18: If the set R is not empty, we update the segment pair (g, g') and release bandwidth to link l.
- line 19: After updating, the segment pair (g, g') is removed in the set R.

Consider the example in Fig. 4(d), the sorted list is: $[(g_1, g_1'), (g_6, g_6'), (g_4, g_4'), (g_3, g_3'), (g_2, g_2'), (g_5, g_5')]$. To find the first round, we check the sorted segment pairs one by one. Only (g_4, g_4') and (g_5, g_5') can acquire enough bandwidths to update. Node (g_4, g_4') takes 0.4 bandwidth from $l_{A,B}$ and $l_{B,D}$ and then node (g_5, g_5') takes 0.1 bandwidth from $l_{A,C}$ and $l_{C,D}$. So $R = \{(g_4, g_4'), (g_5, g_5')\}$. The network state after taking out these bandwidths is shown in Fig. 6(a). The state after finishing updating R is shown in Fig. 6(b). Note that the field $PotUpTm$ needs to be recalculated for all segment pairs. $PotUpTm$ of (g_1, g_1') is reduced to $L + 27$ and that of (g_6, g_6') is reduced to 15. In this example, all dependencies can be resolved, and the final sequence of round is $S = R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow R_4 = \{(g_4, g_4'), (g_5, g_5')\} \rightarrow \{(g_1, g_1')\} \rightarrow \{(g_2, g_2'), (g_3, g_3')\} \rightarrow \{(g_6, g_6')\}$.

Algorithm 3 *GenRnd*

1: **while** True **do**
2: $R = \phi$
3: Sort the segment nodes by $PotUptm$ and $TxBw$
4: **for** each (g, g') in the sorted list **do**
5: **if** $\forall l \rightarrow (g, g') : AvlBw(l) \geq TxBw(g, g')$ **then**
6: **for** each $l \rightarrow (g, g')$ **do**
7: $AvlBw(l) = AvlBw(l) - TxBw(g, g')$
8: **end for**
9: $R = R \cup \{(g, g')\}$
10: **end if**
11: **end for**
12: **if** $R = \phi$ **then**
13: break **while**
14: **else**
15: Conduct flow update in R
16: **for** $\forall (g, g') \rightarrow l, (g, g') \in R$ **do**
17: $AvlBw(l) = AvlBw(l) + TxBw(g, g')$
18: **end for**
19: Remove all $(g, g') \in R$ from $\hat{G}$
20: **if** $\hat{G}$ has no segment node, **then** break **while**
21: **end if**
22: **end if**
23: **end while**

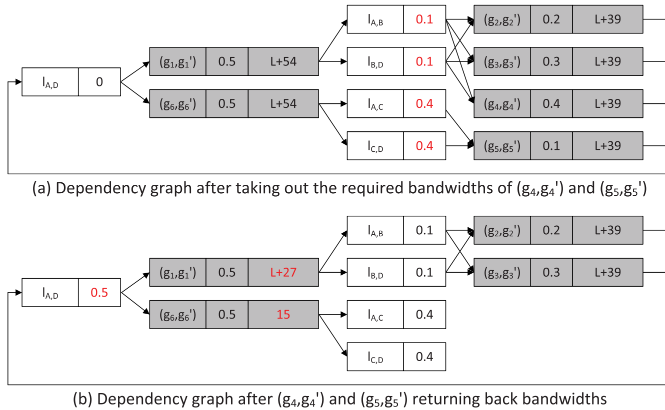


Fig. 6. Change of dependency graph during a round.

D. Resolving Deadlock

After the above steps, if there is any segment pair remaining in $\hat{G}$, function $RsvDead$ is called. It is impossible to resolve a deadlock unless some party in the loop is willing to yield. Therefore, we will identify a pair and reduce its bandwidth requirement so as to migrate it. The first for-loop calculates the *yield ratio* of each $(g, g') \in \hat{G}$ by the following equation:

$$yr(g, g') = \frac{TxBw(g, g') - \min_{l \rightarrow (g, g')} \{AvlBw(l)\}}{TxBw(g, g')}$$

This ratio refers to the percentage of throughput loss during updating. We can restore its bandwidth after the whole migration is done. In line 4-5, we favor the one with the lowest yield ratio and update its $TxBw$. Then the rest of these steps are to conduct the update of (g, g') and reflect the update in $\hat{G}$. Calling this function contributes one round consisting of only one segment pair. Then we will loop back to line 8 of Algorithm 1.

Algorithm 4 HandleDeadlock

1: **for** each $(g, g') \in \hat{G}$ **do**
2: calculate $yr(g, g')$
3: **end for**
4: select (g, g') with the minimal $yr(g, g')$
5: $TxBw(g, g') = TxBw(g, g') \times (1 - yr(g, g'))$
6: **for** each $l \rightarrow (g, g')$ **do**
7: $AvlBw(l) = AvlBw(l) - TxBw(g, g')$
8: **end for**
9: Conduct update for (g, g')
10: **for** $\forall (g, g') \rightarrow l$ **do**
11: $AvlBw(l) = AvlBw(l) + TxBw(g, g')$
12: **end for**
13: $R = R \cup \{(g, g')\}$ and Remove (g, g') from $\hat{G}$

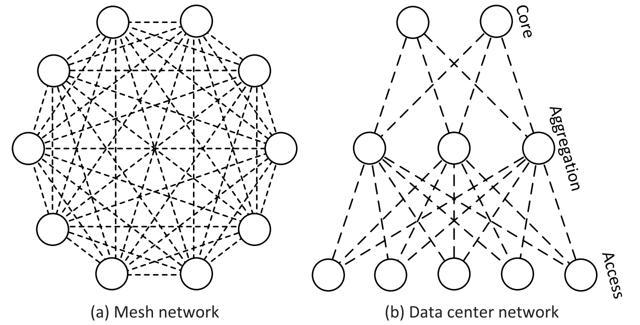


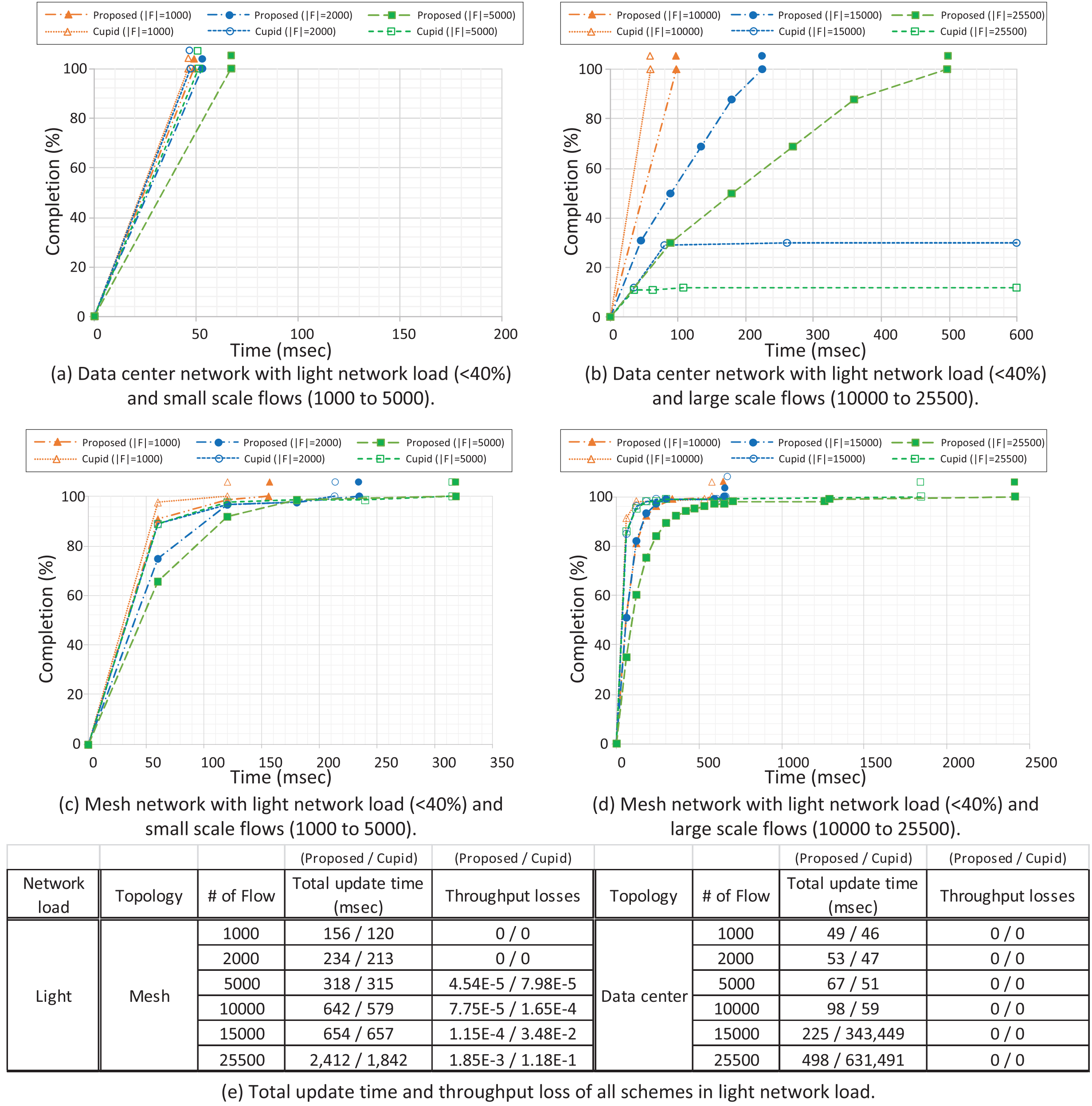
Fig. 7. Network topology (10 switches as examples).

E. Analysis of Time Complexity

In the proposed scheme, it works in four phases. The first phase costs $O(M \times N)$ to partition $|F|$ flow pairs (we let $M = |F|$ for ease of presentation) into shorter routing segments to increase update parallelism. Here, a flow can be partitioned into at most $O(N)$ shorter segments.

The second phase generates a global dependency graph of the segments to be updated. Thus, it costs $O(M \times N)$ because there are M flow pairs and each flow pair has at most $(N - 1)$ segments which connect to/from at most $(N - 1)$ links. Then, it recursively calculates the potential update time for all segments and links. Since there are at most $(N - 1)$ segment nodes and $(N - 1)$ link nodes to be updated, it costs $O(N)$ to traverse all nodes and links in the spanning tree. Thus, the second phase costs totally $O(M \times N) + O(N) = O(M \times N)$.

The third phase conducts actual updates and then adjusts dependency graphs accordingly. To generate the non-deadlock schedules, it first costs $O(MNlogMN)$ to sort at most MN segment nodes based on the potential update time and transmission bandwidth. Then, these segments cost $O(MN \times N)$ to check the available bandwidth of all the links belonging to them and costs $O(MN \times N)$ to conduct flow update



Network load	Topology	# of Flow	Total update time (msec) (Proposed / Cupid)	Throughput losses (Proposed / Cupid)	Topology	# of Flow	Total update time (msec) (Proposed / Cupid)	Throughput losses (Proposed / Cupid)
Light	Mesh	1000	156 / 120	0 / 0	Data center	1000	49 / 46	0 / 0
		2000	234 / 213	0 / 0		2000	53 / 47	0 / 0
		5000	318 / 315	4.54E-5 / 7.98E-5		5000	67 / 51	0 / 0
		10000	642 / 579	7.75E-5 / 1.65E-4		10000	98 / 59	0 / 0
		15000	654 / 657	1.15E-4 / 3.48E-2		15000	225 / 343,449	0 / 0
		25500	2,412 / 1,842	1.85E-3 / 1.18E-1		25500	498 / 631,491	0 / 0

(e) Total update time and throughput loss of all schemes in light network load.

Fig. 8. Update completion efficiency under different traffic flows in light network loads.

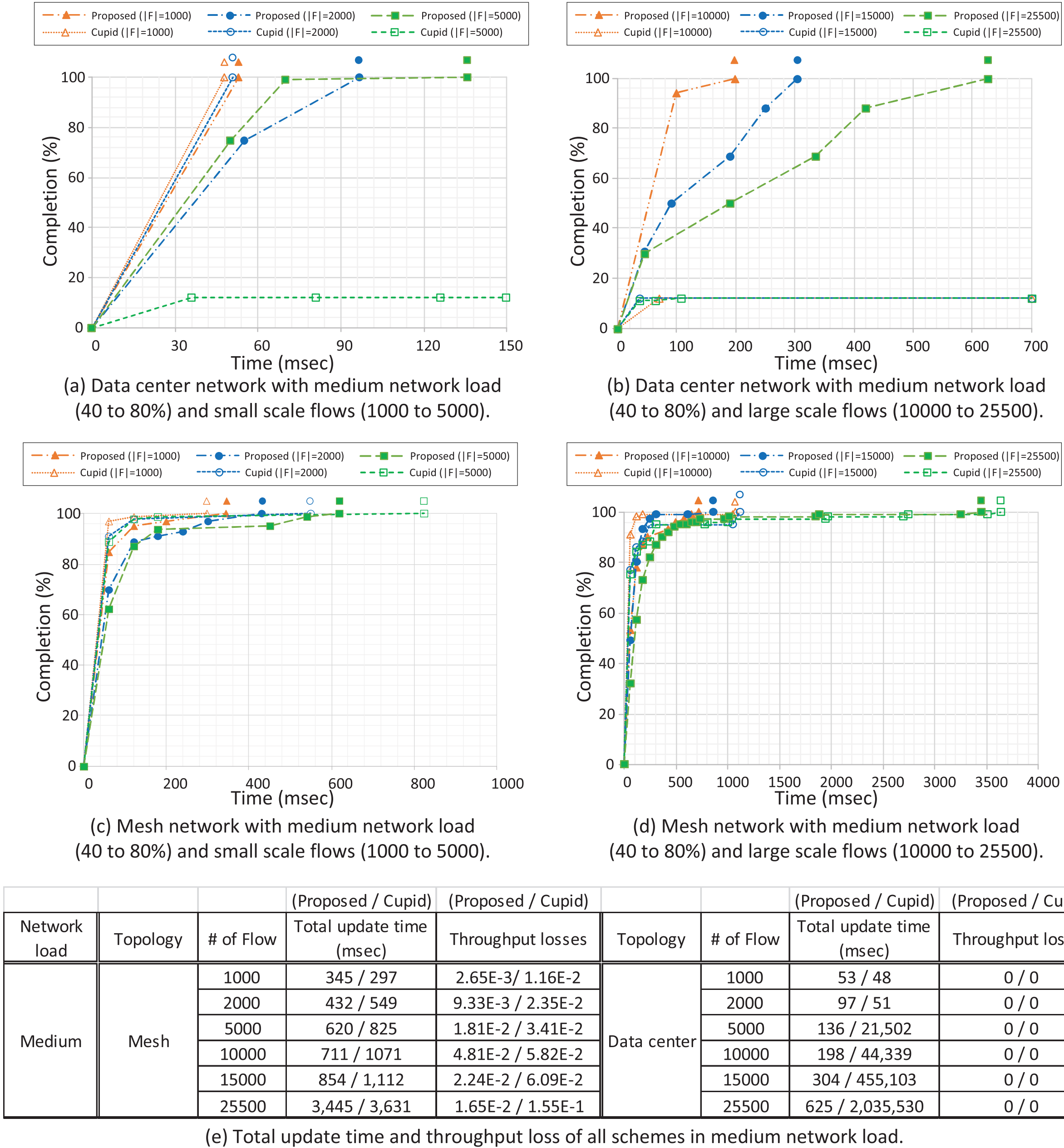
accordingly. Thus, the third phase costs $O(MNlogMN) + O(MN \times N) + O(MN \times N) = O(MNlogMN + MN^2)$.

The last phase deals with deadlocks, if any, and then loops back to phase three if necessary. To resolve deadlocks, it first costs $O(MN)$ to calculate the yield ratio for all deadlock segments in $\hat{G}$. Then, it costs $O(MN)$ to select the segment with the minimal value of yield ratio. Next, it costs $O(1)$ to calculate the new transmission bandwidth. It costs $O(N)$ to update the available bandwidth of all links in this selected segment after applying the new transmission bandwidth. Thus, the fourth phase totally costs $O(MN) + O(MN) + O(1) + O(N) = O(MN)$.

Since the third phase schedules at least one segment in each round and the last phase resolves at least one deadlock in each loop, they need to take at most $K = M \times N$ rounds. Therefore, the proposed scheme totally costs $O(M \times N) + O(M \times N) + K \times (O(MNlogMN + MN^2) + O(MN)) = O(KMNlogMN + KMN^2)$. Note that to speed up the algorithm, we can set a constant threshold for 'K' on the number of iterations. Thus, the complexity will become $O(MNlogMN + MN^2)$.

V. Performance Evaluation

In this section, we develop a simulator by Python language and implement our scheme to validate its performance. The simulator includes two types of network topology: (1) mesh network and (2) data center network [17], as illustrated in Fig. 7. In these two topologies, there are 100 switches with the link bandwidth of 1 (Gbps), where the switches are full-duplex and the traffic flows can be transmitted to or from two connecting switches. In addition, the transmission pairs and routes are generated randomly [17]. Note



(a) Data center network with medium network load (40 to 80%) and small scale flows (1000 to 5000).

(b) Data center network with medium network load (40 to 80%) and large scale flows (10000 to 25500).

(c) Mesh network with medium network load (40 to 80%) and small scale flows (1000 to 5000).

(d) Mesh network with medium network load (40 to 80%) and large scale flows (10000 to 25500).

Network load	Topology	# of Flow	(Proposed / Cupid) Total update time (msec)	(Proposed / Cupid) Throughput losses	Topology	# of Flow	(Proposed / Cupid) Total update time (msec)	(Proposed / Cupid) Throughput losses
Medium	Mesh	1000	345 / 297	2.65E-3/ 1.16E-2	Data center	1000	53 / 48	0 / 0
		2000	432 / 549	9.33E-3 / 2.35E-2		2000	97 / 51	0 / 0
		5000	620 / 825	1.81E-2 / 3.41E-2		5000	136 / 21,502	0 / 0
		10000	711 / 1071	4.81E-2 / 5.82E-2		10000	198 / 44,339	0 / 0
		15000	854 / 1,112	2.24E-2 / 6.09E-2		15000	304 / 455,103	0 / 0
		25500	3,445 / 3,631	1.65E-2 / 1.55E-1		25500	625 / 2,035,530	0 / 0

(e) Total update time and throughput loss of all schemes in medium network load.

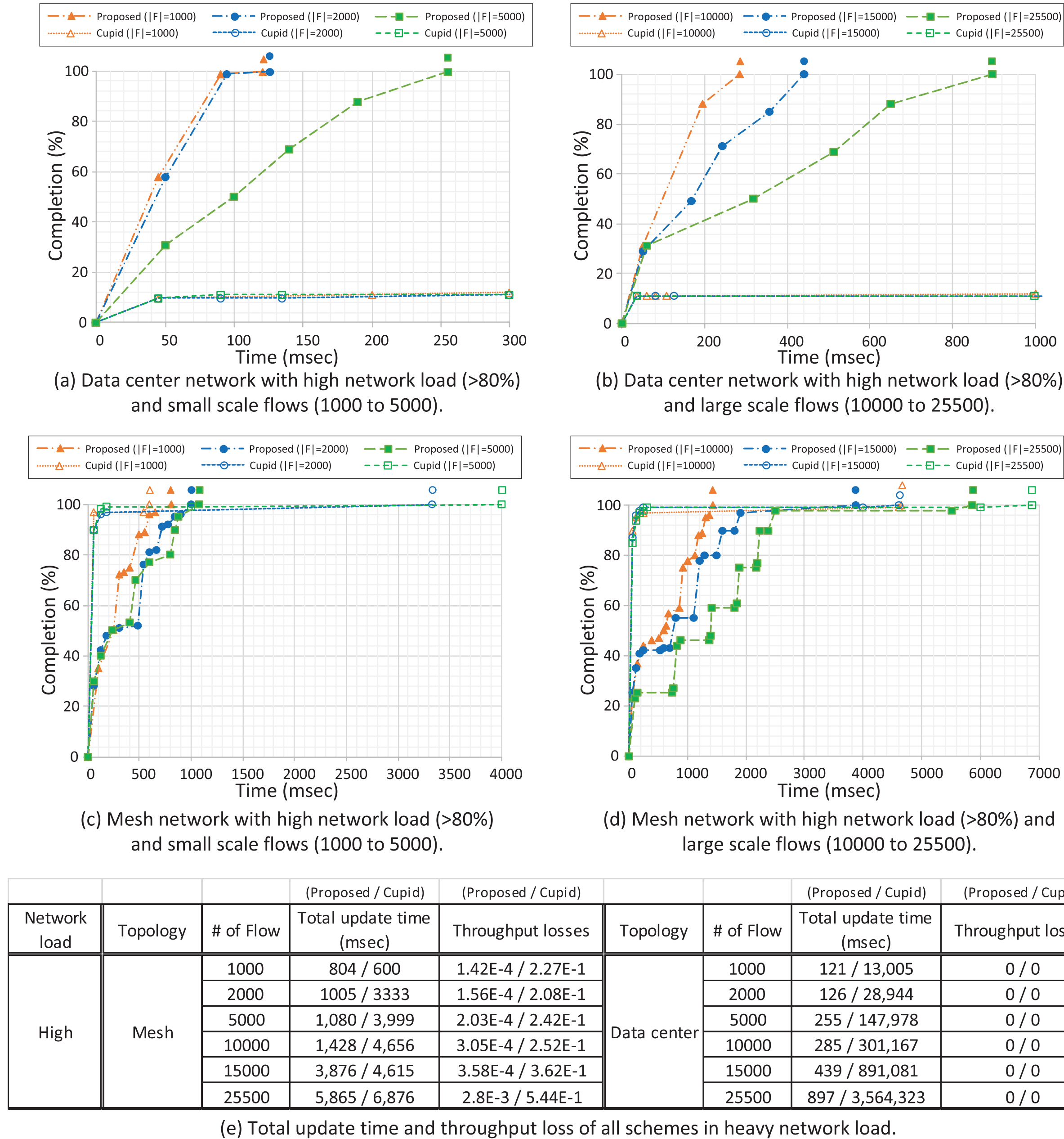
Fig. 9. Update completion efficiency under different traffic flows in medium network loads.

that the data center network contains three layers: core layer (12 switches), aggregation layer (22 switches), and access layer (66 switches). Each data flow can be transmitted through: (1) access layer $\rightarrow$ aggregation layer $\rightarrow$ core layer or (2) core layer $\rightarrow$ aggregation layer $\rightarrow$ access layer, accordingly.

In the simulation, we compare our scheme against Cupid scheme [17]. To simulate a real SDN environment, we add several real parameters into our simulation, such as the latency of real switches. We adopt the measured latencies of off-the-shelf SDN switches in [21], including the time needed for adding a rule to a switch and deleting a rule in rule table. In addition, according to *Open vSwitch* [22] (a software-based SDN switch), which can support a maximum of 255 flows in a switch, is adopted. Thus, the number of flows $|F|$ in our simulation is $=$ 1000, 2000, 5000, 10000, 15000 and 25500 flows. The performance metrics include: 1) update completion efficiency, 2) total update time, and 3) throughput loss. In addition, three levels of network loads are considered: light, medium and heavy, where the loads over the total link capacity are $< 40\%$, 40% to 80%, and $> 80\%$, respectively. Note that in the following Figs. 8, 9 and 10, the proposed scheme is represented by solid patterns and Cupid by hollow patterns.

A. Light Network Load ($< 40\%$)

First, we investigate the number of flows on update completion efficiency when the network load is light. In Fig. 8, we can see that most schemes achieve 100% update completion in both mesh and data center networks except for Cupid in Fig. 8(b). The update efficiency of Cupid in small scale flows is slightly better than ours, such as the results



(a) Data center network with high network load (>80%) and small scale flows (1000 to 5000).

(b) Data center network with high network load (>80%) and large scale flows (10000 to 25500).

(c) Mesh network with high network load (>80%) and small scale flows (1000 to 5000).

(d) Mesh network with high network load (>80%) and large scale flows (10000 to 25500).

			(Proposed / Cupid)	(Proposed / Cupid)			(Proposed / Cupid)	(Proposed / Cupid)
Network load	Topology	# of Flow	Total update time (msec)	Throughput losses	Topology	# of Flow	Total update time (msec)	Throughput losses
High	Mesh	1000	804 / 600	1.42E-4 / 2.27E-1	Data center	1000	121 / 13,005	0 / 0
		2000	1005 / 3333	1.56E-4 / 2.08E-1		2000	126 / 28,944	0 / 0
		5000	1,080 / 3,999	2.03E-4 / 2.42E-1		5000	255 / 147,978	0 / 0
		10000	1,428 / 4,656	3.05E-4 / 2.52E-1		10000	285 / 301,167	0 / 0
		15000	3,876 / 4,615	3.58E-4 / 3.62E-1		15000	439 / 891,081	0 / 0
		25500	5,865 / 6,876	2.8E-3 / 5.44E-1		25500	897 / 3,564,323	0 / 0

(e) Total update time and throughput loss of all schemes in heavy network load.

Fig. 10. Update completion efficiency under different traffic flows in high network loads.

shown in Figs. 8(a) and 8(c), because it does not check all links' capacities for flow pairs. Thus, Cupid incurs more throughput loss when the topology is complicated, such as the mesh network shown in Fig. 8(d). Since Cupid considers only *critical nodes* and *local* dependency graph to perform update, the flows passing through multiple hops encountering congestion cannot be detected. Therefore, Cupid needs to spend more time on resolving deadlocks especially in large-scale flows, e.g., $|F| = 15000 \sim 25500$ in Fig. 8(b). Contrarily, our scheme can detect congestion problems and reduce throughput loss significantly during flow updating, thus taking extra update time.

B. Medium Network Load (40% to 80%)

Next, when the network load increases to medium level, as shown in Fig. 9, the congestion problem is more likely to happen during flow updating. We can see that our scheme outperforms Cupid when the number of flows increases to 10000, 15000 and 25500 in Figs. 9(b) and 9(d). The reason is that more flows result in more deadlocks. Since Cupid addresses deadlocks flow by flow, it takes longer time to resolve deadlocks. Contrarily, our scheme can find all possible schedules and update flows concurrently to solve deadlocks; thus, it can complete in a short time when the network load increases, as shown in Fig. 9(e).

C. Heavy Network Load (> 80%)

Finally, when the network is under heavy load, as shown in Fig. 10, there are potentially more congestions and deadlocks. Thus, all schemes take more time to update. We can see that our scheme still outperforms Cupid in terms of total

update time and throughput loss when the number of flows is larger than 1000. Since more flows in a data center network need to pass through access layer, aggregation layer, and then core layer, the bottleneck between core layer and aggregation layer should be addressed first. Similar to Fig. 9(a)(b), Cupid needs more time to handle its deadlock procedure and may not complete in short time, as shown in Fig. 10(a)(b). It is worth noting that our scheme can achieve 100% completion in both network topologies under high network load with massive flows because our scheme can update flows more efficiently and consistently.

VI. Conclusions

In this paper, we have addressed the efficiency and consistency issues for software defined networks in terms of Blackhole problem, Loop problem, and Deadlock problem when migrating flows. We have proposed an efficient flow update scheme which considers both efficiency and consistency by four phases. First, it partitions flows into shorter routing segments to increase update parallelism. Second, it generates a global dependency graph for network status maintenance. Third, it conducts actual updates and adjusted dependency graphs. Finally, it deals with deadlocks when loops exists. Through simulations, we have verified that our scheme can not only ensure freedom of blackholes, loops, congestions, and deadlocks during flow updates, but also run faster than existing schemes. For future directions, one may use SDN software, such as *Open vSwitch (OVS)* [22] to build a prototype. Our simulation is at flow level; it deserves to study packet-level simulations.

References

[1] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined WAN," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 3–14, 2013.

[2] S. Agarwal, M. Kodialam, and T. V. Lakshman, "Traffic engineering in software defined networks," in *Proc. IEEE INFOCOM*, Apr. 2013, pp. 2211–2219.

[3] C.-Y. Chu, K. Xi, M. Luo, and H. J. Chao, "Congestion-aware single link failure recovery in hybrid SDN networks," in *Proc. IEEE INFOCOM*, Apr./May 2015, pp. 1086–1094.

[4] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Enabling fast failure recovery in OpenFlow networks," in *Proc. Design Rel. Commun. Netw. (DRCN)*, 2011, pp. 164–171.

[5] K. Xi and H. J. Chao, "IP fast rerouting for single-link/node failure recovery," in *Proc. Broadband Commun., Netw. Syst.*, Sep. 2007, pp. 142–151.

[6] H. Sufiev and Y. Haddad, "A dynamic load balancing architecture for SDN," in *Proc. IEEE Sci. Elect. Eng. (ICSEE)*, Nov. 2016, pp. 1–3.

[7] Y.-L. Lan, K. Wang, and Y.-H. Hsu, "Dynamic load-balanced path optimization in SDN-based data center networks," in *Proc. IEEE Int. Symp. Commun. Syst., Netw. Digit. Signal Process. (CSNDSP)*, Jul. 2016, pp. 1–6.

[8] J. Liu, J. Li, G. Shou, Y. Hu, Z. Guo, and W. Dai, "SDN based load balancing mechanism for elephant flow in data center networks," in *Proc. IEEE Int. Symp. Wireless Pers. Multimedia Commun. (WPMC)*, Sep. 2014, pp. 486–490.

[9] Y. Hu, W. Wang, X. Gong, X. Que, and S. Cheng, "Balanceflow: Controller load balancing for OpenFlow networks," in *Proc. IEEE Cloud Comput. Intell. Syst. (CCIS)*, Oct./Nov. 2012, pp. 780–785.

[10] C. Cooper, R. Klasing, and T. Radzik, "Searching for black-hole faults in a network using multiple agents," in *Principles of Distributed Systems*. Berlin, Germany: Springer, 2006, pp. 320–332.

[11] R. Schoonderwoerd, O. Holland, and J. Bruten, "Ant-like agents for load balancing in telecommunications networks," in *Proc. Auto. AGENTS*, 1997, pp. 209–216.

[12] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. USENIX Conf. Netw. Syst. Design Implement. (NSDI)*, vol. 10. 2010, p. 19.

[13] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," *Comput. Commun. Rev.*, vol. 41, no. 4, pp. 254–265, Aug. 2011.

[14] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," in *Proc. 7th Conf. Emerg. Netw. Experim. Technol. (CoNEXT)*, 2011, p. 12.

[15] M. Suchara, D. Xu, R. Doverspike, D. Johnson, and J. Rexford, "Network architecture for joint failure recovery and traffic engineering," in *Proc. ACM SIGMETRICS Joint Int. Conf. Meas. Modeling Comput. Syst.*, 2011, pp. 97–108.

[16] X. Jin *et al.*, "Dynamic scheduling of network updates," in *Proc. ACM SIGCOMM*, 2014, pp. 539–550.

[17] W. Wang, W. He, J. Su, and Y. Chen, "Cupid: Congestion-free consistent data plane update in software defined networks," in *Proc. IEEE INFOCOM*, Apr. 2016, pp. 1–9.

[18] C.-Y. Hong *et al.*, "Achieving high utilization with software-driven WAN," in *Proc. ACM SIGCOMM*, 2013, pp. 15–26.

[19] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zUpdate: Updating data center networks with zero loss," in *Proc. ACM SIGCOMM*, 2013, pp. 411–422.

[20] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2012, pp. 323–334.

[21] K. He *et al.*, "Measuring control plane latency in SDN-enabled switches," in *Proc. ACM SIGCOMM Symp. Softw. Defined Netw. Res.*, 2015, pp. 25:1–25:6.

[22] *Open vSwitch*. Accessed: Feb. 6, 2018. [Online]. Available: http://openvswitch.org/