Continuous Network Update with Consistency Guaranteed in Software-Defined Networks

Xin He, *Student Member, IEEE,* Jiaqi Zheng, *Member, IEEE,* Haipeng Dai, *Member, IEEE,* Chong Zhang, *Student Member, IEEE,* Geng Li, *Member, IEEE,* Wanchun Dou, *Member, IEEE,* Wajid Rafique, Qiang Ni, *Senior Member, IEEE,* Guihai Chen, *Senior Member, IEEE*

Abstract—**Network update enables Software-Defined Networks (SDNs) to optimize the data plane performance. The single update focuses on processing one update event at a time, *i.e.*, updating a set of flows from their initial routes to target routes, but it fails to handle continuously arriving update events in time incurred by high-frequency network changes. On the contrary, the continuous update proposed in "Update Algebra" can handle multiple update events concurrently and respond to the network condition changes at all times. However, "Update Algebra" only guarantees the blackhole-free and loop-free update. The congestion-free property cannot be respected. In this paper, we propose Coeus to achieve the continuous update while maintaining consistency, *i.e.*, ensuring the blackhole-free, loop-free, and congestion-free properties simultaneously. Firstly, we establish the continuous update model based on the update operations in update events. With the update model, we dynamically reconstruct the operation dependency graph (ODG) to capture the relationship between update operations and link utilization variations. Then, we develop a composition algorithm to eliminate redundant operations in update events. To further speed up the update procedure, we present a partition algorithm to split the operation nodes of the ODG into a series of suboperation nodes that can be executed independently. The partition algorithm is proven to be optimal. Finally, extensive evaluations show that Coeus can improve the update speed by at least 179% and reduce redundant operations by at least 52% compared with state-of-the-art approaches when the arrival rate of update events equals three times per second.**

Index Terms—**Continuous update, consistency, SDNs, operation dependency graph.**

I. Introduction

Software-Defined Networks (SDNs) outsource the network control function over switches to the logically centralized controller. Benefitting from the global view of the controller and the simplified data plane, SDNs can provide flexible traffic management and fine-grained network monitoring (*e.g.*, traffic engineering [1]–[3] and failures recovery [4], [5]), and have been widely deployed in datacenter networks [6]–[8]. However, the traffic in datacenters is highly volatile and bursty [9], [10]. To respond to the high-frequency network condition variations, network applications produce a series of network configuration updates [11], [12]. For simplicity, we refer to a network configuration update as an update event. Each update event consists of a series of update operations, which assign new routes for flows. Executing an update event means updating flows from their initial routes to target routes. The SDNs controller needs to execute these continuously arriving update events as soon as possible to optimize the data plane performance continuously [13], [14]. Furthermore, the network consistency should be guaranteed, where the consistency means no blackholes, no forwarding loops, and no link congestion during the update [15]–[18].

Most update solutions focus on processing the single update event at a time, *i.e.,* finding the update sequence for an update event to shorten the update time while maintaining consistency [19]–[25]. We collectively refer to these update schemes as the single update. The single update fails to process high-frequency network changes or the burst event in time. For example, the load balancing can react to congestion and produce update events at a microsecond level [26]. However, the single update may complete each update event at a second level [27]. In the single update, the continuously arriving update events are executed in sequence. Each update event cannot be responded to until the previous update events are completed, leading to prolonged update time and poor network performance. Instead of the single update, the current work, "Update Algebra" [28], explores the solution that can execute the continuously arriving update events in parallel. Specifically, "Update Algebra" models update operations in update events as a set of projections and selects a feasible subset of projections to execute. We refer to the solution of executing continuous update events in parallel as the continuous update. Although 'Update Algebra" can rapidly handle continuous update events, it only guarantees the blackhole-free and loop-free properties and ignores the fact that different flows compete for the limited link resources. Without a well-designed update mechanism, transient link congestion may occur, leading to packet loss and network performance degradation.

In this paper, we initiate the study of the continuous update with consistency guaranteed, *i.e.*, handling the continuously arriving update events in time and ensuring consistency. To achieve this goal, we face three main challenges: (1) The

The work was supported in part by the National Key R&D Program of China under Grant 2017YFB1001801, in part by the National Natural Science Foundation of China under Grants 61802172, 61672276, 61872178, 61832005, 61672353, in part by the Natural Science Foundation of Jiangsu Province under Grant BK20181251, and in part by the Royal Society project IEC170324. (Corresponding authors: Wanchun Dou and Guihai Chen)

X. He is with the School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China (xhe@njupt.edu.cn).

J. Zheng, C, Zhang, H, Dai, W. Rafique, W. Dou, G. Chen are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China (e-mail: jzheng@nju.edu.cn, chongzhang@smail.nju.edu.cn, haipengdai@nju.edu.cn, rafiqwajid@smail.nju.edu.cn, douwc@nju.edu.cn, gchen@nju.edu.cn).

G. Li is with the Department of Computer Science, Yale University, New Haven CT 06520, USA (e-mail: ligeng66@aliyun.com).

Q. Ni is with the School of Computing and Communications, Lancaster University, Lancashire LA1 4WA, U.K. (e-mail: q.ni@lancaster.ac.uk).

update operations in continuously arriving update events that act on the same flow may be redundant [28], [29]. For example, one update operation requires modifying a flow forwarding rule. Then another update operation requires deleting this forwarding rule. Actually, executing the delete operation is equivalent to executing these two update operations in sequence. It is challenging to identify and eliminate redundant operations of continuously arriving update events in time because these update events may contain lots of update operations and appear rapidly. Besides, the execution of update operations will incur the link resource variations, and the link resource will conversely affect the execution order of update operations. To produce a consistent update order, the dependency relationship between link resources and update operations should be carefully considered. (2) The number of possible congestion-free update orders is exponential, even for a single update event [27]. For an update event that updates n flows, the number of possible update orders is $O(n!)$. Involving multiple update events makes the problem essentially harder. (3) We need to speed up the continuous update process, *i.e.*, increasing the parallel execution of update operations. We aim to find as many independent operations as possible, which can be executed simultaneously. To address the challenges mentioned above, we make the following contributions.

Firstly, we develop Coeus to achieve the continuous update while ensuring consistency. We illustrate the continuous update problem by using an example (Sec. III) and build the continuous update model (Sec. IV). The continuous update model captures the continuously update events in the control plane and the forwarding actions in the data plane. Besides, we give an overview of Coeus (Sec. V).

Secondly, we propose a set of algorithms to achieve the continuous update with consistency guaranteed (Sec. VI). Specifically, based on the continuous update model, we dynamically reconstruct the operation dependency graph (ODG) to capture the relationship between update operations and link utilization variations. The ODG is a bipartite graph containing the operation nodes and the resource nodes. Then, we develop an operation composition algorithm to represent redundant operations as directed loops and eliminate redundant operations in the ODG. With the ODG and operation composition, we solve the first two challenges. To address the third challenge, we design a partition algorithm to split the operation nodes into a series of independent suboperation nodes that can be updated in parallel. We prove that the partition algorithm is optimal, and the update of suboperation nodes ensures consistency.

Finally, we conduct large-scale simulations on two common topologies (*i.e.*, SWAN [27] and fat-tree [30]) to verify the effectiveness of Coeus (Sec. VII). The simulations show that Coeus can improve the update speed by at least 179% and reduce redundant operations by at least 52% compared with state-of-the-art approaches when the arrival rate of update events is three times per second.

II. Related Work

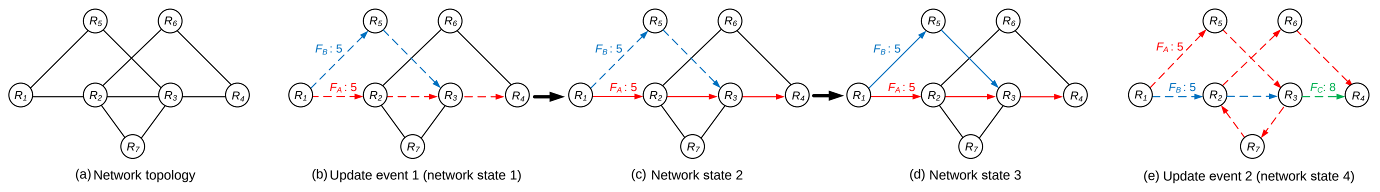
With the advent of SDNs, the update problem has been widely studied in recent years. Reitblatt *et al.* [6] introduced the notion of consistent update in SDNs and proposed the two-phase update protocols to maintain per-packet coherence. To ensure connectivity consistency, Ludwig and Foerster *et al.* [31], [32] achieved the fast blackhole-free and loop-free update by using node-ordering protocols. However, they do not consider link congestion. To ensure the congestion-free condition, zUpdate [7] and SWAN [33] utilized the slack link capacity to produce the static congestion-free update sequence. On these bases, Xin *et al.* [27] and Gandhi *et al.* [34] presented the dynamic update scheduling by utilizing the global resource dependency graph. Wang *et al.* [20] and Wu *et al.* [35] divided the global dependency relationship into the local dependency relationship by dividing flows into segments. Then, independent segments can be updated in parallel while the congestion-free condition is guaranteed. By taking advantage of time synchronization protocols [36]–[38], Zheng *et al.* [39]–[41] designed heuristic algorithms to update a single flow and multiple flows at a specified time with minimum time step while maintaining the congestion-free property. Different with scheduling update commands in a centralized manner, Nguyen *et al.* [42] proposed a decentralized mechanism to achieve the blackhole-free, loop-free, and congestion-free update. Besides, when a congestion-free update plan does not exist, Zheng *et al.* [25], [43] designed a flow migration approach to minimize transient congestion and shorten the makespan. Nevertheless, all of the update solutions mentioned above mainly focus on producing the update order for the single update event. Such an update manner leads to the serial execution of continuous update events, which slows down the makespan significantly.

To the best of our knowledge, Li *et al.* [28] firstly studied the continuous update problem. The authors built the theoretical framework for continuous update events based on abstract algebra and executed updates based on the operation projection. Although "Update Algebra" [28] can generate the blackhole-free and loop-free update order, it cannot guarantee the congestion-free condition, leading to packet loss and network performance degradation. On this basis, we proposed Coeus, which is the first work to handle the continuous update events with blackhole-free, loop-free, and congestion-free properties guaranteed simultaneously.

III. A Motivating Example

In this section, we use a motivating example to illustrate the continuous update problem. For convenience, we summarize important notations in Table I.

Fig. 1(a) shows a network topology containing seven switches $R_1 \sim R_7$, where the capacity of link $\langle R_1, R_5 \rangle$ equals 5 units and others equal 10 units. The update event 1 (UE_1) and the update event 2 (UE_2) arrive sequentially. UE_1 will install forwarding rules of two flows F_A and F_B, where demands of F_A and F_B are both 5 units. UE_2 will install forwarding rules of flow F_C, where the demand of F_C is 8 units. Each update event incurs different network states, where each network state captures the routing information. We use the directed edge to denote the routing state of each flow. The dashed line denotes that the forwarding rules of the flow have not been installed, while the solid line denotes that the forwarding rules of the flow have already been installed. Fig. 1(b) denotes that when UE_1 appears, the controller will



R_1 R_2 R_3 R_4 R_5 R_6 R_7
$F_B: 5$
$F_A: 5$
$F_C: 8$
(a) Network topology
(b) Update event 1 (network state 1)
(c) Network state 2
(d) Network state 3
(e) Update event 2 (network state 4)

Fig. 1. A continuous update example. Fig. 1(a) shows the network topology. The capacity of link $\langle R_1, R_5\rangle$ is assumed to be 5 units while others are assumed to be 10 units. The dashed line denotes that the forwarding rules for the flow will be installed in switches, and the flow will route on its target path. The solid line denotes that the forwarding rules for the flow have been installed in switches and the flow has been routed on its target path. Fig. 1(b) represents that the update event 1 will assign the paths of F_A and F_B, where the demand of F_A and F_B are both 5 units. Fig. 1(c) represents that F_A has been routed on its target path, while the forwarding rules for F_B have not been installed. Fig. 1(d) represents that F_B has been routed on its target path. Fig. 1(e) represents that the update event 2 will inject flow F_C with 8 units in the network. F_A, and F_B need to be rerouted to avoid the congestion on $\langle R_3, R_4\rangle$.

assign the path of F_A and F_B. However, the corresponding forwarding rules have not been installed in switches. Fig. 1(c) shows that in the network state 2 (NS_2), F_A has been updated to its target path while F_B has not been updated[1]. The network state 3 (NS_3) shown in Fig. 1(d) indicates that F_B has been updated. In NS_3, UE_1 is finished since all update operations in UE_1 are executed, *i.e.,* flows in UE_1 are routed on their target paths. Fig. 1(e) shows that when UE_2 occurs after UE_1, the controller will assign the route of F_C and change the routes of F_A and F_B due to the limited capacity of link $\langle R_3, R_4\rangle$.

TABLE I
KEY NOTATIONS

Notation	Meaning
G	Directed network graph $G = (V, E)$
V	Set of switches $\{v\}$
E	Set of links $\{\langle u, v\rangle\}$
$c_{u,v}$	Capacity of link $\langle u, v\rangle$
F	Set of flows $\{f_i\}$
d^{f_i}	Demand of flow f_i
UE	Update event
O_{UE}	Set of update operations in UE.
$\widehat{O}_{UE}$	Set of unexecuted update operations in UE.
NS	Network state
NS_{init}	Initial network state
O_I	Set of executed update operations
O_{NI}	Set of unexecuted update operations
G_D	Operation dependency graph
O^{f_i}	Operation node with a set of unexecuted update operations of f_i in the operation dependency graph
o^{f_i}	Update operation in O^{f_i}
$R_{u,v}$	Resource node in the operation dependency graph
$r_{u,v}$	Residual resources of the resource node

Since the network traffic is highly dynamic, update events may occur randomly and continuously. In the single update, the subsequent update event must wait for the completion of previous update events, *i.e.*, if UE_2 occurs after UE_1, UE_2 cannot be processed until all of the flows in UE_1 are routed on their target paths. This serial update prolongs the makespan. This paper aims to rapidly respond to random and continuous update events while ensuring blackhole-free, loop-free, and congestion-free properties. However, achieving the objective is full of challenges. For example, UE_2 occurs when UE_1 has installed the forwarding rule of F_A in switch R_2 but has not installed the forwarding rule of F_A in switch R_3. UE_1 encourages the controller to install the forwarding rule of F_A in R_3, making R_3 forward F_A to R_4. In the meantime, UE_2 also encourages the controller to install another forwarding rule of F_A in R_3, making R_3 forward F_A to R_7. Therefore, simply installing all of the forwarding rules of continuous update events may lead to chaotic routing. Besides, switches must have forwarding rules of each incoming flow, and the flow should not have transient loops during updates (*i.e.*, the blackhole-free and loop-free properties should be guaranteed). For example, when UE_2 occurs after NS_3, switch R_5 should have the forwarding rule of F_A before F_A reaches. Besides, the new forwarding rules of F_A should be installed in R_3 and R_7 after the original forwarding rule in R_2 is removed. Otherwise, the transient loop $\langle R_2, R_3\rangle \rightarrow \langle R_3, R_7\rangle \rightarrow \langle R_7, R_2\rangle$ appears. To guarantee a congestion-free condition, the update order of multiple flows should be carefully calculated. For example, when UE_2 occurs after NS_3, F_A should not be updated before F_B. Otherwise, link congestion occurs due to the limited resources of $\langle R_1, R_5\rangle$. Similarly, F_C should be updated after F_A. To further speed up the update process, the concurrency of update operations should be taken into account. For each flow, we should carefully consider which update operations can be executed in parallel. For example, when UE_2 occurs after NS_3, installing the forwarding rules of F_A in R_1 and R_3 simultaneously cannot incur inconsistency. However, installing the forwarding rules of F_A in R_2, R_3, and R_7 simultaneously may result in transient loops since the execution of update operations in the data plane is asynchronous [44].

[1] NS_2 is a possible network state. In practice, update commands can be sent by the controller simultaneously and completed by the data plane switches asynchronously, resulting in different network states. Coeus can be applied to all of the possible network states.

IV. Update Model and Problem Definition

In this section, we will introduce the continuous update model and the continuous update problem.

A. Continuous Update Model

The general network can be modeled as a directed graph $G = (V, E)$, where V is the set of switches and $E \subseteq V^2$ is the set of bi-directional links. Each link $\langle u, v\rangle \in E$ has a capacity $c_{u,v}$. F is the set of $s-d$ flows in the network, where s and d are the source and destination of a flow. Each flow $f_i \in F$ is an unsplittable flow with demand d^{f_i}.

In the control plane, continuous update events arrive randomly. Each update event $UE = \{o^{f_i} | \forall f_i \in F\}$ contains a set of update operations $\{o^{f_i}\}$ that assign routes of flows. An update operation o^{f_i} is the operation necessary to change the route of a flow. To be more specific, o^{f_i} acts on a single switch to add, modify, or delete the forwarding rule of f_i. If an update operation o^{f_i} sent by the controller is executed on the switch, the switch will execute the corresponding forwarding action. All of the update operations can be classified into three types $\{add, mod, del\}$. We illustrate the mapping relationship between update operations in the control plane

TABLE II
VALID UPDATE OPERATIONS OF UE_n

Update Event	Update Operation		
UE_m	$o^{f_i}_{add}$	$o^{f_i}_{del}$	$o^{f_i}_{mod}$
UE_n	$o^{f_i}_{del} \parallel o^{f_i}_{mod}$	$o^{f_i}_{add}$	$o^{f_i}_{del} \parallel o^{f_i}_{mod}$

and forwarding actions in the data plane. If o^{f_i} is an *add* operation $o^{f_i}_{add}: add\{fwd\_v\}$ and is executed on switch u, the switch u adds the forwarding rule and forwards f_i from u to v. If o^{f_i} is a *del* operation $o^{f_i}_{del}: del\{fwd\_v\}$ and is executed on switch u, the forwarding rules of f_i is deleted. Similarly, if o^{f_i} is a *mod* operation $o^{f_i}_{mod}: mod\{fwd\_v \rightarrow fwd\_w\}$ and is executed on switch u, the switch u forwards f_i to w instead of v. In fact, $o^{f_i}_{mod}$ can be treated as a delete operation $o^{f_i}_{del}$ that removes the original forwarding rule of f_i, and an add operation $o^{f_i}_{add}$ that adds the new forwarding rule of f_i.

The network state $NS = \{O_I, O_{NI}\}$ characterizes the current routing state. O_I denotes the set of update operations that have been executed and O_{NI} denotes the set of update operations that have not been executed. Suppose there are n continuous update events, O_I and O_{NI} can be expressed as follows.

$$O_I = \{(O_{UE_1} - \widehat{O}_{UE_1}) ... \cup (O_{UE_i} - \widehat{O}_{UE_i}) ... \cup (O_{UE_n} - \widehat{O}_{UE_n})\} \quad (1)$$

$$O_{NI} = \{\widehat{O}_{UE_1} ... \cup \widehat{O}_{UE_i} ... \cup \widehat{O}_{UE_n}\} \quad (2)$$

where O_{UE_i} and $\widehat{O}_{UE_i}$ represent all of the update operations and all of the unexecuted update operations in the i-th arriving update event UE_i, respectively. When update event UE_i occurs, $\widehat{O}_{UE_i} = O_{UE_i}$. With the update procedure, update operations in $\widehat{O}_{UE_i}$ are continuously executed until $\widehat{O}_{UE_i} = \emptyset$. Therefore, operations in O_I and O_{NI} vary with the time.

B. Continuous Update Problem

The network update needs to schedule unexecuted update operations in order to preserve the consistency properties [6]. Instead of executing the continuously arriving update events serially, the continuous update aims to process update events concurrently. To characterize the update process of the continuous update, we define the initial network state as follows.

Definition 1. Initial Network State: *The initial network state is the state that all of the previously arrived update events are completed, i.e., no update operations need to be executed in the data plane at this time.*

In the initial network state, $O_I = \emptyset$. The network will not update routes of flows. With the emergence of newly arriving update events $\{UE\}_{new}$, a series of update operations should be executed to configure routes of flows continuously. Therefore, the continuous update process starts from the initial network state and ends when all of the newly arriving update events $\{UE\}_{new}$ are completed. In the continuously arriving update events, not all of the update operations are valid. We define the validity of update operations as follows.

Definition 2. Valid Update Operations: *For each flow, the valid update operations depend on the previous update operations acting on the same switches. The valid update operations enable each switch to have at most one forwarding rule for each flow, and the incorrect operation is not permitted.*

Specifically, the valid update operations follow the rules shown in Table II, where UE_m is a previous update event, and UE_n is a newly arriving update event. On the specified switch, if UE_m involves an *add* operation $o^{f_i}_{add}$ for f_i, the operation in UE_n should only be a *del* or mod operation (*i.e.*, $o^{f_i}_{del} \parallel o^{f_i}_{mod}$) that deletes or modifies the original forwarding rule of f_i. Otherwise, there is another *add* operation $o^{f_i}_{add} \in UE_n$ for f_i acting on the same switch, leading to chaotic routing. If UE_m involves a *del* operation $o^{f_i}_{del}$, the operation for f_i in UE_n should only be an add operation (*i.e.*, $o^{f_i}_{add}$). It is because that if $o^{f_i}_{del} \in UE_m$ for f_i is executed, there is no forwarding rule that needs to be deleted or modified for f_i on the same switch. Therefore, $o^{f_i}_{mod} \in UE_n$ or $o^{f_i}_{add} \in UE_n$ that appears after $o^{f_i}_{del} \in UE_m$ is an incorrect update operation. If UE_m contains a *mod* operation $o^{f_i}_{mod}$, the operation in UE_n acting on the same switch should only be a del or mod operation (*i.e.*, $o^{f_i}_{del} \parallel o^{f_i}_{mod}$). It is because that if there is an add operation $o^{f_i}_{add} \in UE_n$ that appears after a *mod* operation $o^{f_i}_{mod} \in UE_m$, the switch will have two forwarding rules for f_i.

Definition 3. Valid Update Event: *The valid update event contains a series of valid update operations for each flow.*

In this paper, we assume that all of the continuously arriving update events are valid update events. Now we define the continuous update problem.

Definition 4. Continuous Update Problem: *The continuous update problem aims to design a scheduling scheme to execute valid update events that arrive continuously in time. Meanwhile, the blackhole-free, loop-free, and congestion-free properties should be guaranteed during the update procedure.*

We propose Coeus to solve the continuous update problem. Coeus is mainly focus on valid update events. However, network applications sometimes generate invalid update events due to the highly dynamic network environment [45], [46]. In Sec VI-G, we discuss how to convert invalid update events so as Coeus can also process them.

V. COEUS OVERVIEW

In this section, we will introduce Coeus. Coeus mainly focuses on traffic management applications for the network core (*e.g.*, B4 [1], SWAN [33]). Similar to [20], [27], [28], Coeus assumes that each forwarding rule in the switch matches at most one flow, making Coeus unsuitable for wild-card rules or the longest prefix matching. We make this assumption since a loop-free update order does not exist in networks that use wild-card rules or the longest prefix matching [19].

The entire workflow of Coeus is shown in Fig. 2. In SDNs, update events arrive continuously. To react to these update events in time, the centralized controller judges whether the previous update events have been finished when the new update event arrives. If previous update events have been finished, we build an operation dependency graph for newly arriving update events. Otherwise, we make a further judgment about whether the update operations in newly arriving update events act on new flows or existing flows in the data plane. For the update operations that act on new flows, *i.e.,* executing these operations will add paths for new flows, we build new operation nodes for these flows in the operation dependency graph and construct the corresponding dependency relationship. For the update operations that act on the existing flows,

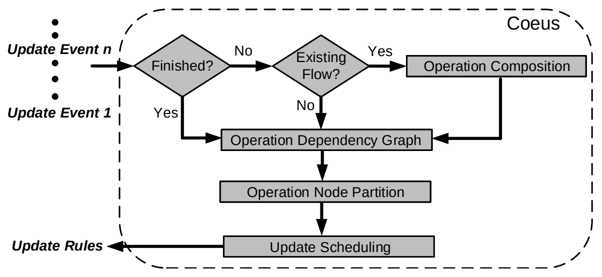


Fig. 2. Overview of Coeus.

i.e., executing these operations will change the routes of flows, we add these operations to existing operation nodes and dynamically reconstruct the dependency relationship between existing operation nodes and link resource nodes. However, the continuously arriving update events may contain redundant update operations, which increases the size of the dependency graph and prolongs the update time. The operation composition module is designed to eliminate redundant operations in the operation dependency graph. With the composited dependency graph, the operation node partition module divides the operation nodes into a series of independent suboperation nodes that can be updated in parallel. Finally, according to the dependency graph with suboperation nodes, the update scheduling module sends a set of update commands to switches, and switches execute these commands until all of the update events are completed.

VI. Coeus Scheduling

In this section, we will present Coeus consisting of a series of algorithms to achieve the continuous update while ensuring consistency.

Before introducing the algorithms in detail, we sketch the problems solved by each algorithm.

The first problem is when update events arrive continuously, how to determine the dependency relationship between link resources and update operations to ensure the congestion-free update. To solve this problem, Algorithm 1 and Algorithm 2 construct the initial dependency graph and dynamically reconstruct the dependency graph when an update event occurs.

The second problem is when we reconstruct the dependency graph, the update operations in update events increase the size of the dependency graph, resulting in complicated dependencies and long update time. However, we do not need to perform all the update operations since some update operations in update events are redundant. How to identify and eliminate redundant operations to reduce the size of the dependency graph? To solve this problem, Algorithm 3 checks and eliminates the directed cycles in the dependency graph so as to combine redundant operations into fewer equivalent ones.

The third problem is that Algorithm 3 only reduces the number of operations that need to be executed. To ensure that each flow does not suffer from the blackholes and forwarding loops, a feasible solution is to perform all update operations of a flow serially in reverse order [28], [31], [42]. To further speed up the update process, we need to investigate how to implement the parallel update. To solve this problem, Algorithm 4 splits the operation nodes into a series of independent suboperation nodes that can be executed in parallel.

The last problem is how to send the update commands to data plane switches in order while ensuring consistency. To solve this problem, Algorithm 5 schedules updates based on the constraints imposed by the operation dependency graph.

Now we introduce our algorithms in detail.

A. Continuous Update Process

Algorithm 1 illustrates the complete continuous update process. Once an update event appears after the initial network state, we construct the initial operation dependency graph G_D (lines 1-5). When the operation dependency graph $G_D \neq \emptyset$, *i.e.*, the update events have not been completed, we check whether a new update event UE arrives (lines 7). Once a new update event occurs, we divide update operations in UE into the update operations for the existing flows and the update operations for the emerging flows. We use $O_{UE}^{f_i}$ to denote a set of update operations for f_i in update event UE. If f_i is an existing flow, we apply Algorithm 3 to composite the update operations in the dependency graph (lines 9-10). If f_i is an emerging flow, we reconstruct the dependency graph by adding a new operation node and building the corresponding dependency relationships (lines 11-14). Then, we apply Algorithm 4 to generate suboperation nodes in the dependency graph (line 15). According to the operation dependency graph with suboperation nodes, we apply Algorithm 5 to produce and send a set of update commands to update the data plane switches until $G_D = \emptyset$ (line 16).

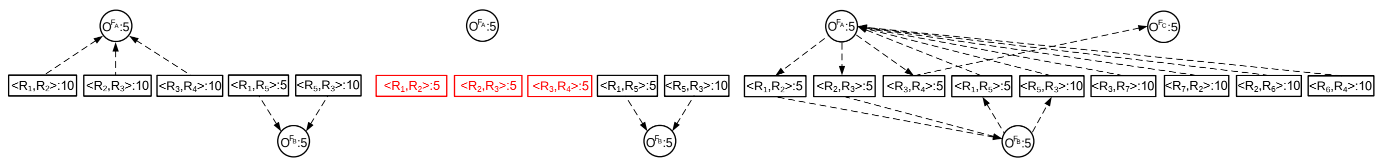
Algorithm 1 Continuous Update Process

```
Input: The continuously arriving update events
Output: A set of update commands
 1: $G_D(O, R, E_{O\dashrightarrow R}, E_{R\dashrightarrow O}) = \emptyset$;
 2: for each updated flow $f_i$ do
 3:     Add $O^{f_i}$ with its demand $d^{f_i}$ in $O$;
 4:     for each operation $o^{f_i} \in O^{f_i}$ do
 5:         Apply Algorithm 2 to obtain the ODG;
 6: for $G_D(O, R, E_{O\dashrightarrow R}, E_{R\dashrightarrow O}) \neq \emptyset$ do
 7:     if the new update event $UE$ arrives then
 8:         for each $O_{UE}^{f_i}$ do
 9:             if flow $f_i$ is the existing flow then
10:                 Apply Algorithm 3 to composite update operations;
11:             else
12:                 Add $O_{UE}^{f_i}$ with its demand $d^{f_i}$ in $O$;
13:                 for each operation $o^{f_i} \in O_{UE}^{f_i}$ do
14:                     Apply Algorithm 2 to build the dependency relationship;
15:         Apply Algorithm 4 to divide an operation node into independent
            suboperation nodes;
16:     Apply Algorithm 5 to schedule update commands;
```

B. Operation Dependency Graph Construction

When update events occur continuously, we construct and adjust the operation dependency graph to capture the relationship between the resource variations and update operations. We define the operation dependency graph as follows.

Definition 5. Operation Dependency Graph (ODG): *The operation dependency graph* $G_D(O, R, E_{O\dashrightarrow R}, E_{R\dashrightarrow O})$ *is a bipartite graph that captures dependency relationships between update operations and link resources, where the two subsets of vertices* O *and* R *denote the set of update operations and the set of links.* $E_{O\dashrightarrow R}$ *is the set of directed edges from vertices in* O *to vertices in* R*, and* $E_{R\dashrightarrow O}$ *is the set of directed edges from vertices in* R *to vertices in* O*.*



(a) Operation dependency graph of update event 1 (b) Operation dependency graph of network state 2 (c) Operation dependency graph of update event 2

Fig. 3. Operation dependency graph of UE_1, NS_2, and UE_2 in Fig. 1.

Specifically, each operation node $O^{f_i} \in O$ with a set of unexecuted update operations $\{o^{f_i}\}$ is labeled with the flow demand d^{f_i}. Each link resource node $R_{u,v} \in R$ is labeled with the residual link resources $r_{u,v}$. The edge $E_{R_{u,v} \dashrightarrow O^{f_i}} \in E_{R \dashrightarrow O}$ denotes that once the update operation $o^{f_i} \in O^{f_i}$ is executed, the link resource will be occupied by f_i. Inversely, the edge $E_{O^{f_i} \dashrightarrow R_{u,v}} \in E_{O \dashrightarrow R}$ denotes that executing the update operation $o^{f_i} \in O^{f_i}$ will release the link resource.

Now we introduce the dependency relationship construction in Algorithm 2. As Algorithm 1 mentioned, we add the operation node O^{f_i} in O (lines 2, 3). Then, in Algorithm 2, we judge the type of each update operation $o^{f_i} \in O^{f_i}$. For each update operation $o^{f_i} \in O^{f_i}$, we build a directed edge between O^{f_i} and $R_{u,v}$ in the ODG (lines 1-13). Specifically, if o^{f_i} is an *add* operation, executing o^{f_i} will add the new forwarding rule in switch u, which forwards f_i to link $\langle u, v\rangle$. Therefore, the link resources of $\langle u, v\rangle$ will be consumed. We add resource node $R_{u,v}$ in the ODG if $R_{u,v}$ does not exist. Then we add directed edge $E_{R_{u,v} \dashrightarrow O^{f_i}}$ in $E_{R \dashrightarrow O}$ (lines 1-4). Similarly, we build the relationship between the operation node and the resource node if o^{f_i} is a del or mod operation (lines 5-13). Note that a mod operation can be treated as a del operation and an add operation, we add $E_{R_{u,v} \dashrightarrow O^{f_i}}$ in $E_{R \dashrightarrow O}$ and $E_{O^{f_i} \dashrightarrow R_{p,q}}$ in $E_{O \dashrightarrow R}$ (lines 9-13).

Algorithm 2 Dependency Relationship Construction

```
Input: The update operation $o^{f_i} \in O^{f_i}$
Output: The dependency relationship between $O^{f_i}$ and $R_{u,v}$
 1: if $o^{f_i}$ is an add operation then
 2:     if corresponding link node $R_{u,v} \notin R$ then
 3:         Add $R_{u,v}$ with its current capacity $c_{u,v}$ in $R$;
 4:     Add $E_{R_{u,v} \dashrightarrow O^{f_i}}$ in $E_{R \dashrightarrow O}$;
 5: if $o^{f_i}$ is a del operation then
 6:     if corresponding link node $R_{p,q} \notin R$ then
 7:         Add $R_{p,q}$ with its current capacity $c_{p,q}$ in $R$;
 8:     Add $E_{O^{f_i} \dashrightarrow R_{p,q}}$ in $E_{O \dashrightarrow R}$;
 9: if $o^{f_i}$ is a mod operation then
10:     Treat $o^{f_i}$ as an add and a del operation;
11:     if corresponding link node $R_{u,v}$ or $R_{p,q} \notin R$ then
12:         Add $R_{u,v}$ with its current capacity $c_{u,v}$ or add $R_{p,q}$ with its
            current capacity $c_{p,q}$ in $R$;
13:     Add $E_{R_{u,v} \dashrightarrow O^{f_i}}$ in $E_{R \dashrightarrow O}$ and $E_{O^{f_i} \dashrightarrow R_{p,q}}$ in $E_{O \dashrightarrow R}$;
```

The ODG maintains the dependency relationship between the unexecuted update operations and the link resources. We illustrate the variation of the ODG when the update operations in O^{f_i} are executed. Once $o^{f_i} \in O^{f_i}$ is executed, we remove o^{f_i} from O^{f_i} and delete the directed edge between O^{f_i} and $R_{u,v}$ established by o^{f_i}. Besides, the link residual resources are also updated. Specifically, if the performed operation is an add operation, f_i is routed on its target path and the corresponding link resources are consumed. The residual link resources can be updated by following Eq. (3). If the performed operation is a del operation, *i.e.,* the forwarding rule of f_i is deleted, the occupied link resources are released. The residual link resources can be updated by following Eq. (4). Similarly, if the performed operation is a mod operation, the resources on the target link of f_i are updated by following Eq. (3), and the resources on the initial link of f_i are updated by following Eq. (4).

$$r^*_{u,v} = r_{u,v} - d^{f_i} \tag{3}$$

$$r^*_{u,v} = r_{u,v} + d^{f_i} \tag{4}$$

Fig. 3 shows the ODG of UE_1, NS_2, and UE_2 in Fig. 1. We assume that UE_2 arises after NS_2. Fig. 3(a) corresponds to the ODG of UE_1 shown in Fig. 1(b), where the operation node O^{F_A} contains a set of unexecuted operations $\{o^{F_A}_{(R_1,add)}, o^{F_A}_{(R_2,add)}, o^{F_A}_{(R_3,add)}\}$ of F_A. Fig. 3(b) corresponds to the ODG of NS_2 shown in Fig. 1(c). Since operations in O^{F_A} has been executed and F_A has been routed on its target path, the corresponding directed edges are removed and the residual link resources are updated. Fig. 3(c) shows the ODG of UE_2. which means update operations in UE_2 will add the route of flow F_C and adjust the routes of F_A and F_B.

C. Operation Composition

We present the graph-based operation composition to reduce the number of redundant operations in continuously arriving update events. Different from the algebra-based rules composition mentioned in [28], the graph-based operation composition is more intuitive and easier to be applied in the ODG.

The input of the algorithm is the update operation in a newly arriving update event UE and the ODG. After the operation composition, the algorithm outputs the composited dependency graph with fewer operations that need to be executed. To transform redundant operations into fewer equivalent ones, we add the update operation in UE to the existing operation node in the ODG. For the added operation, we construct the directed edge between the operation node and the resource node. Then we check whether the addition of the operation will incur a direct cycle between the operation node and the resource node. If a directed cycle forms, we treat the operations that establish the two directed edges of a directed cycle as redundant operations. By eliminating directed cycles and removing the corresponding operations in the ODG, the number of operations that need to be executed is reduced. We will explain the operation composition algorithm in detail and then prove that the composited operations are correct.

Algorithm 3 shows an operation composition process. When an update event UE with a set of update operations occurs, we add each operation o^{f_i} to the corresponding operation node and apply Algorithm 2 to build the dependency relationship (lines 1-2). According to the rules of constructing the ODG, if o^{f_i} is an add operation, we establish the directed edge

$E_{R_{u,v} \dashrightarrow O^{f_i}}$ in the ODG, which means that f_i will route on the link $\langle u, v \rangle$ and consume the link resources. If there exists the directed edge $E_{O^{f_i} \dashrightarrow R_{u,v}}$ in the ODG, which denotes that O^{f_i} has a *del* operation $\bar{o}^{f_i}_{del}$ to delete the forwarding rule of f_i on switch u and the resources of link $\langle u, v \rangle$ will be released. In this case, a cycle forms between O^{f_i} and $R_{u,v}$. We treat the *add* operation o^{f_i} and the *del* operation $\bar{o}^{f_i}_{del}$ which will occupy and release the same link resources successively as redundant operations. Since the target path of the flow will not be changed by discarding these two update operations, we remove two directed edges forming a cycle and $\{o^{f_i}, \bar{o}^{f_i}_{del}\}$ from the ODG (lines 3-6). Similarly, if o^{f_i} is a *del* operation and a cycle forms, according to Definition 2, there is an *add* or a *mod* operation $\bar{o}^{f_i}$ in O^{f_i}. If $\bar{o}^{f_i}$ is an *add* operation, we remove o^{f_i} and $\bar{o}^{f_i}$ directly. Otherwise, we divide $\bar{o}^{f_i}$ into an add operation $\bar{o}^{f_i}_{add}$ and a *del* operation $\bar{o}^{f_i}_{del}$. Then we remove o^{f_i} and $\bar{o}^{f_i}_{add}$ from O^{f_i} (lines 7-12). If o^{f_i} is a *mod* operation and a cycle forms, we split o^{f_i} into $o^{f_i}_{add}$ and $o^{f_i}_{del}$, and eliminate the redundant operations related to the removed edges (lines 13-19). Besides, if the residual operations of o^{f_i} and $\bar{o}^{f_i}$ contain a *del* operation and an *add* operation acting on different links, we combine the residual operations into a new *mod* operation $\tilde{o}^{f_i}$ (line 20). If o^{f_i} does not incur a cycle in the ODG and there exists another update operation $\hat{o}^{f_i}$ acting on the same switch, we combine these two operations into a new *mod* operation $\tilde{o}^{f_i}$ (line 22).

Algorithm 3 Operation Composition

```
Input: The operation in a newly arriving update event and the ODG
Output: The composited dependency graph
 1: Add $o^{f_i}$ in $O^{f_i}$;
 2: Apply Algorithm 2 to build the dependency relationship;
 3: if there is a directed cycle between $O^{f_i}$ and $R_{u,v}$ then
 4:     Remove edges $E_{R_{u,v} \dashrightarrow O^{f_i}}$ and $E_{O^{f_i} \dashrightarrow R_{u,v}}$;
 5:     if $o^{f_i}$ is an add operation then
 6:         $O^{f_i} = O^{f_i} / \{o^{f_i}, \bar{o}^{f_i}_{del}\}$, where $\bar{o}^{f_i}_{del}$ is the operation that established the directed edge $E_{O^{f_i} \dashrightarrow R_{u,v}}$;
 7:     if $o^{f_i}$ is a del operation then
 8:         if $\bar{o}^{f_i}$ is an add operation then
 9:             $O^{f_i} = O^{f_i} / \{o^{f_i}, \bar{o}^{f_i}_{add}\}$, where $\bar{o}^{f_i}_{add}$ is the operation that established the directed edge $E_{R_{u,v} \dashrightarrow O^{f_i}}$;
10:         else
11:             Divide $\bar{o}^{f_i}$ into $\bar{o}^{f_i}_{add}$ and $\bar{o}^{f_i}_{del}$;
12:             $O^{f_i} = O^{f_i} / \{o^{f_i}, \bar{o}^{f_i}_{add}\}$;
13:     if $o^{f_i}$ is a mod operation then
14:         Divide $o^{f_i}$ into $o^{f_i}_{add}$ and $o^{f_i}_{del}$;
15:         if $\bar{o}^{f_i}$ is an add or a del operation then
16:             $O^{f_i} = O^{f_i} / \{o^{f_i}_{del}, \bar{o}^{f_i}_{add}\}$ or $O^{f_i} = O^{f_i} / \{o^{f_i}_{add}, \bar{o}^{f_i}_{del}\}$;
17:         else
18:             Divide $\bar{o}^{f_i}$ into $\bar{o}^{f_i}_{add}$ and $\bar{o}^{f_i}_{del}$;
19:             Remove the divided operations of $o^{f_i}$ and $\bar{o}^{f_i}$ that established the directed edges $E_{R_{u,v} \dashrightarrow O^{f_i}}$ and $E_{O^{f_i} \dashrightarrow R_{u,v}}$;
20:             Combine the residual operations of $o^{f_i}$ and $\bar{o}^{f_i}$ into a new mod operation $\tilde{o}^{f_i}$;
21: else
22:     Combine $o^{f_i}$ and $\hat{o}^{f_i}$ into a new operation $\tilde{o}^{f_i}$, where $\hat{o}^{f_i}$ and $o^{f_i}$ are update operations for $f_i$ that acting on the same switch;
```

Fig. 4 is an example to illustrate the operation composition. We assume that UE_2 (shown in Fig. 1(e)) occurs after NS_2 (shown in Fig. 1(c)). For simplicity, Fig. 4 only shows the composition result of operation node O^{F_B}. In NS_2, O^{F_B} contains update operations $\{\bar{o}^{F_B}_{(R_1,add)}, \bar{o}^{F_B}_{(R_5,add)}\}$. In Fig. 4(a), the black line denotes the dependency relationship between

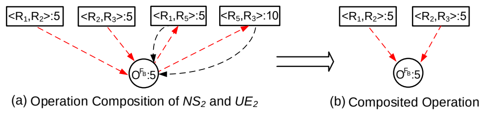


Fig. 4. Illustration of operation composition for O^{F_B}.

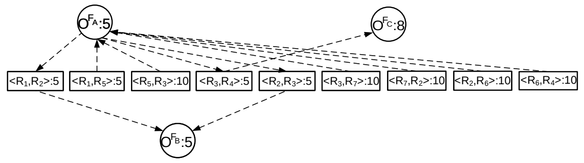


Fig. 5. Composited dependency graph.

link resources and the unexecuted operations in NS_2. Then, UE_2 arises, Coeus adds a set of new operations $\{o^{F_B}_{(R_1,mod)}, o^{F_B}_{(R_5,del)}, o^{F_B}_{(R_2,add)}\}$ to O^{F_B}. The red line in Fig. 4(a) denotes the dependency relationship between link resources and the operations in UE_2. Update operations $\{o^{F_B}_{(R_5,add)}, o^{F_B}_{(R_5,del)}\}$ and $\{o^{F_B}_{(R_1,add)}, o^{F_B}_{(R_1,mod)}\}$ incur two cycles. According to Algorithm 3, we delete $\{o^{F_B}_{(R_5,add)}, o^{F_B}_{(R_5,del)}\}$ and the corresponding edges in the ODG. Besides, we divide $o^{F_B}_{(R_1,mod)}$ into $o^{F_B}_{(R_1,add)}$ and $o^{F_B}_{(R_1,del)}$. Then, we delete $\{o^{F_B}_{(R_1,add)}, o^{F_B}_{(R_1,del)}\}$ from the operation node O^{F_B} and remove directed edges. The composited operation node O^{F_B} is shown in Fig. 4(b) and the composited operation dependency graph is shown in Fig. 5.

Definition 6. Correct Update Operations: *The correct update operations in the update event mean that if all update operations are executed, flows will be routed through their target paths required by the update event, and there are no redundant forwarding rules for each flow in switches.*

In the single update, all operations in the previous update events must be finished before executing the new update event. For example, we assume that two valid update events UE_m, UE_n arrive successively. When UE_n occurs, the update operations of UE_m may be completed, partially executed, or unexecuted. UE_n can only be responded after all of the update operations in UE_m are finished. Since UE_m and UE_n are valid update events, executing UE_m and UE_n orderly makes switches always forward flows to the target paths specified by the update event. Therefore, in the single update, the correctness of update operations can always be guaranteed. Table III represents the correct update operations in the single update, where blanks denote invalid operations. However, the single update incurs lots of unnecessary operations. In Coeus, regardless of the state of UE_m, UE_n can be responded in time by compositing the unexecuted operations in UE_m and the operations in UE_n. Now, we prove that the update operations are still correct after the operation composition.

TABLE III
CORRECT OPERATIONS IN THE SINGLE UPDATE

UE_n \ UE_m	$o^{f_i}_{add}$	$o^{f_i}_{del}$	$o^{f_i}_{mod}$
$o^{f_i}_{add}$		$o^{f_i}_{add}$	
$o^{f_i}_{del}$	$o^{f_i}_{del}$		$o^{f_i}_{del}$
$o^{f_i}_{mod}$	$o^{f_i}_{mod}$		$o^{f_i}_{mod}$

Theorem 1. *The graph-based operation composition produces the correct update operations.*

Proof: Table IV shows the result of operation composition when operations in UE_m have not been executed and

TABLE IV
COMPOSITION OF UE_m AND UE_n

UE_m / UE_n	$o^{f_i}_{add}$	$o^{f_i}_{del}$	$o^{f_i}_{mod}$
$o^{f_i}_{add}$		$\emptyset \parallel \tilde{o}^{f_i}_{mod}$	
$o^{f_i}_{del}$	$\emptyset$		$o^{f_i}_{del}$
$o^{f_i}_{mod}$	$\tilde{o}^{f_i}_{add}$		$\tilde{o}^{f_i}_{mod}$

UE_n occurs. We prove that after the operation composition, executing operations in Table IV and Table III are equivalent. According to Algorithm 3, an *add* operation in UE_n and a *del* operation in UE_m may produce two types of operations after composition. One is the empty set $\emptyset$, which means doing nothing in the switch. The flow still routes along its original path. This composited result corresponds to the condition where a *del* operation in UE_m deletes the original path of flow, then an *add* operation in UE_n adds the new path which is the same as the original path in the single update. Another possible result is a new *mod* operation $\tilde{o}^{f_i}_{mod}$ that will delete the original path and add a new path of f_i, making f_i route along the new path. This composited result corresponds to the condition where a *del* operation in UE_m deletes the original path of flow and an *add* operation in UE_n adds the new path which is different from the original path. Therefore, executing the new operation $\tilde{o}^{f_i}_{mod}$ is equivalent to executing the corresponding update operations serially in the single update.

Similarly, if there is an add operation $o^{f_i}_{add} \in UE_m$ and a del operation $o^{f_i}_{del} \in UE_n$, in the single update, $o^{f_i}_{add}$ and $o^{f_i}_{del}$ are executed successively. The switch will install and then delete the forwarding rule of f_i (*i.e.,* there is no forwarding rule for f_i on the switch in the end). In the continuous update, since UE_m and UE_n have not been executed, we composite these two operations into $\emptyset$ which means that we will not install the forwarding rule for f_i. For an add operation $o^{f_i}_{add} \in UE_m$ and a mod operation $o^{f_i}_{mod} \in UE_n$, Coeus composites these operations into a add operation $\tilde{o}^{f_i}_{add}$. Once $\tilde{o}^{f_i}_{add}$ is executed, f_i will be forwarded to its new path. The routing of f_i is identical to performing an add operation $o^{f_i}_{add} \in UE_m$ and a mod operation $o^{f_i}_{mod} \in UE_n$ successively. For the case where update operations in UE_n and UE_m are mod operations, the single update executes $o^{f_i}_{mod} \in UE_n$ and f_i will finally route through the path specified by $o^{f_i}_{mod} \in UE_n$. According to Algorithm 3, Coeus divides these two mod operations into add operations and del operations. Then the redundant operations are removed, and the residual operations are composed into the new mod operation $\tilde{o}^{f_i}_{mod}$. The final path of f_i specified by $\tilde{o}^{f_i}_{mod}$ is the same as that of $o^{f_i}_{mod} \in UE_n$. ■

Note that if operations in UE_m have been executed when UE_n occurs, operations in UE_n cannot be combined with operations in UE_m. Therefore, Coeus will execute operations of UE_n in switches according to the rule shown in Table III.

D. Operation Node Partition

To speed up the update process, we present an operation node partition algorithm to divide each operation node into a series of independent suboperation nodes that can be updated in parallel. Our algorithm is inspired by the partition technology used in [20], [35], while they either cause excessive dependencies among each partition [20] or cannot be applied to the situation where the network has potential loops [35]. In Coeus, we develop a novel partition algorithm. We prove that our algorithm is optimal for obtaining independent suboperation nodes. Besides, we prove that each suboperation node produced by the algorithm can be updated in a blackhole-free and loop-free manner. Now, we define the suboperation node and the independent suboperation node.

Definition 7. Suboperation Node: *The suboperation node $O^{f_i}_j \subseteq O^{f_i}$ contains at least one mod operation to shift f_i from its original path to target path. The union of suboperation nodes contains all update operations in O^{f_i} and each suboperation node contains different update operations, i.e., $O^{f_i}_j \cup O^{f_i}_{j+1} \cup \cdots = O^{f_i}$, $O^{f_i}_j \cap O^{f_i}_{j+1} \cap \cdots = \emptyset$.*

Definition 8. Independent Suboperation Node: *The independent suboperation node is the suboperation node that can be updated independently without incurring forwarding loops.*

In the following, we describe the process of operation node partition in Algorithm 4. We define $O^{f_i}_j$ as the j-th suboperation node of O^{f_i}, which contains a part of operations to update the forwarding rules of switches along the original path and the target path of f_i. Firstly, we traverse update operations in O^{f_i} along the target path of f_i in reverse order (line 2). When traversing to a *mod* operation $o^{f_i}_{mod}$, we construct a suboperation node $O^{f_i}_j = \varphi^{f_i}_{jo} \cup \varphi^{f_i}_{jt}$, where $\varphi^{f_i}_{jo}$ and $\varphi^{f_i}_{jt}$ denote the sets of update operations acting on the original subpath and the target subpath of f_i, respectively (line 4). Then we add $o^{f_i}_{mod}$ to $O^{f_i}_j$ and a set of *add* operations $\{o^{f_i}_{add}\}$ between $o^{f_i}_{mod}$ and $o^{f_i}_{(mod,nt)}$ to $\varphi^{f_i}_{jt}$, where $o^{f_i}_{(mod,nt)}$ is the next *mod* operation of $o^{f_i}_{mod}$ along the target path (lines 5, 6). Executing $o^{f_i}_{mod}$ and $\varphi^{f_i}_{jt}$ makes f_i route on its target subpath. According to $o^{f_i}_{mod}$ and the operations in $\varphi^{f_i}_{jt}$, we judge whether $O^{f_i}_j$ is an independent suboperation node (lines 7-10). Specifically, a forwarding loop occurs when the following condition are satisfied: (1) there is a potential loop in the flow path (e.g., the flow path shown in Fig. 7(b)), and (2) the operations for adding the target subpath are executed earlier than the operations for deleting the original subpath in a potential loop. To avoid a loop, we assign operations that may incur a loop to two suboperation nodes. The first suboperation node contains operations to delete the original subpath that is involved in a potential loop. The second suboperation node contains operations to add the target subpath that is involved in a potential loop. The second suboperation node cannot be updated until the first suboperation node is completed. Therefore, if the target subpath added by operations in $O^{f_i}_j$ is involved in a potential loop, $O^{f_i}_j$ is a dependent node. Otherwise, $O^{f_i}_j$ is an independent node. Next, we check whether the original subpath deleted by $o^{f_i}_{mod}$ and the target subpath added by $o^{f_i}_{(mod,no)}$ are involved in a loop, where $o^{f_i}_{(mod,no)}$ is the next *mod* operation of $o^{f_i}_{mod}$ along the original path of f_i. If the condition is true, there exits a dependent node (line 11). We iteratively check whether the original subpath deleted by $o^{f_i}_{(mod,no)}$ and the target subpath added by the next *mod* operation of $o^{f_i}_{(mod,no)}$ along the original path are involved in a loop (line 12). When the original subpath and the target subpath of $o^{f_i}_{(mod,no)}$ are involved in two loops, we split $o^{f_i}_{(mod,no)}$ into two operations $o^{f_i}_{(add,sp)}$ and $o^{f_i}_{(del,sp)}$ (line 13). The split operation enables $o^{f_i}_{(add,sp)}$ and $o^{f_i}_{(del,sp)}$ to be assigned to two suboperation nodes, which reduces the length of dependencies. Otherwise, there is a node that depends on the node containing $o^{f_i}_{(mod,no)}$, and the latter depends on the node containing $o^{f_i}_{mod}$. Finally, we add *del* operations $\{o^{f_i}_{del}\}$ between $o^{f_i}_{mod}$ and $o^{f_i}_{(mod,no)}$ to $\varphi^{f_i}_{jo}$ (line 15) and construct the next suboperation node (lines 17, 18).

Algorithm 4 Operation Node Partition

```
Input: The operation node $O^{f_i}$ in dependency graph
Output: The suboperation nodes $\{O^{f_i}_j\}$
 1: $j = 0$;
 2: Traverse operations in $O^{f_i}$ along the target path of $f_i$ in reverse order;
 3: while $o^{f_i}_{mod} \neq \emptyset$ do
 4:     $O^{f_i}_j = \varphi^{f_i}_{jo} \cup \varphi^{f_i}_{jt}$, where $\varphi^{f_i}_{jo} = \varphi^{f_i}_{jt} = \emptyset$;
 5:     $O^{f_i}_j = O^{f_i}_j \cup o^{f_i}_{mod}$, $O^{f_i} = O^{f_i}/o^{f_i}_{mod}$;
 6:     $\varphi^{f_i}_{jt} = \varphi^{f_i}_{jt} \cup \{o^{f_i}_{add}\}$, where $\{o^{f_i}_{add}\}$ is a set of add operations between
        $o^{f_i}_{mod}$ and $o^{f_i}_{(mod,nt)}$;
 7:     if the target subpath added by $o^{f_i}_{mod}$ and $\varphi^{f_i}_{jt}$ is involved in a loop then
 8:         $O^{f_i}_j$ is a dependent suboperation node;
 9:     else
10:         $O^{f_i}_j$ is an independent suboperation node;
11:     if the original subpath deleted by $o^{f_i}_{mod}$ and the target subpath added
        by $o^{f_i}_{(mod,no)}$ are involved in a loop then
12:         while the original subpath deleted by $o^{f_i}_{(mod,no)}$ and the target
            subpath added by the next mod operation of $o^{f_i}_{(mod,no)}$ along the
            original path are involved in a loop do
13:             Split $o^{f_i}_{(mod,no)}$ into $o^{f_i}_{(add,sp)}$ and $o^{f_i}_{(del,sp)}$;
14:             $o^{f_i}_{(mod,no)}$ = the next mod operation along the original path;
15:     $\varphi^{f_i}_{jo} = \varphi^{f_i}_{jo} \cup \{o^{f_i}_{del}\}$, where $\{o^{f_i}_{del}\}$ is a set of del operations between
        $o^{f_i}_{mod}$ and $o^{f_i}_{(mod,no)}$;
16:     $O^{f_i}_j = \varphi^{f_i}_{jo} \cup \varphi^{f_i}_{jt}$;
17:     $j = j + 1$;
18:     $o^{f_i}_{mod} = o^{f_i}_{(mod,nt)}$;
19: Divide $O^{f_i}$ into a set of suboperation nodes $\{O^{f_i}_j\}$;
```

We use the continuous update instance mentioned in Fig. 1 as an example to illustrate the partition process. We assume that UE_2 arises after NS_2. After finishing the operation composition shown in Fig. 5, we partition the operation nodes as follows. As shown in Fig. 1, the update operations along the target path of F_A in O^{F_A} are $\{o^{F_A}_{(R_1,mod)}, o^{F_A}_{(R_5,add)}, o^{F_A}_{(R_3,mod)}, o^{F_A}_{(R_7,add)}, o^{F_A}_{(R_2,mod)}, o^{F_A}_{(R_6,add)}\}$. Firstly, we add $o^{F_A}_{(R_2,mod)}$ to $O^{F_A}_1$ and add $o^{F_A}_{(R_6,add)}$ to $\varphi^{F_A}_{1t}$. Along the original path of F_A, the subpath $\langle R_2, R_3\rangle$ that will be deleted by $o^{F_A}_{(R_2,mod)}$ is involved in a potential loop, *i.e.*, $\{R_2 \to R_3 \to R_7\}$, whereas the original subpath $\langle R_3, R_4\rangle$ that will be deleted by $o^{F_A}_{(R_3,mod)}$ is not involved in potential loop. Therefore, $o^{F_A}_{(R_3,mod)}$ will not be split. According to Algorithm 4, we obtain an independent suboperation node $O^{F_A}_1 = \{o^{F_A}_{(R_2,mod)}, o^{F_A}_{(R_6,add)}\}$ and a dependent suboperation nodes $O^{F_A}_2$. For the suboperation nodes $O^{F_A}_2$, we add $o^{F_A}_{(R_3,mod)}$ to $O^{F_A}_2$ and add $o^{F_A}_{(R_7,add)}$ to $\varphi^{F_A}_{2t}$. We continue to search the next mod operation along the reverse direction of the target path of F_A, *i.e.*, $o^{F_A}_{(R_1,mod)}$ and judge that whether the original subpath that

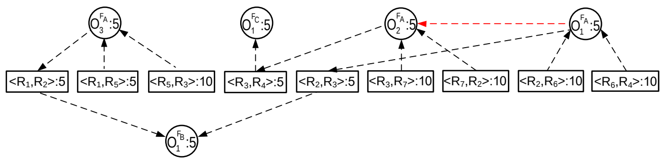
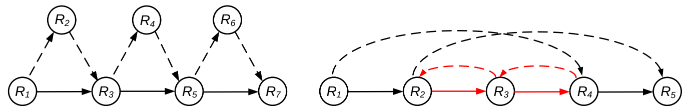


Fig. 6. Dependency graph with suboperation nodes.



(a) The network without potential loops (b) The network with potential loops

Fig. 7. Illustration of independent suboperation nodes.

will be deleted by $o^{F_A}_{(R_1,mod)}$ is involved in a loop. Then, we produce $O^{F_A}_2 = \{o^{F_A}_{(R_3,mod)}, o^{F_A}_{(R_7,add)}\}$. Similarly, the suboperation nodes $O^{F_A}_3 = \{o^{F_A}_{(R_1,mod)}, o^{F_A}_{(R_5,add)}\}$ of F_A, $O^{F_B}_1 = \{o^{F_B}_{(R_1,add)}, o^{F_B}_{(R_2,add)}\}$ of F_B, and $O^{F_C}_1 = \{o^{F_C}_{(R_3,add)}\}$ of F_C are generated. Fig. 6 is the dependency graph with suboperation nodes, where $O^{F_A}_3$ and $O^{F_A}_1$ are independent suboperation nodes of F_A. The red edge between $O^{F_A}_1$ and $O^{F_A}_2$ denotes that $O^{F_A}_2$ cannot be updated before $O^{F_A}_1$.

We can prove the following about Algorithm 4.

Theorem 2. *The number of independent suboperation nodes obtained by Algorithm 4 is optimal.*

Proof: We prove Algorithm 4 is optimal for networks with or without potential loops. If the flow route has no potential loop (*e.g.,* the route shown in Fig. 7(a)), Algorithm 4 adds each *mod* operation to different suboperation nodes. Therefore, the number of *mod* operations is equal to the number of independent suboperation nodes. According to Definition 8, the number of independent suboperation nodes produced by Algorithm 4 is optimal. For the situation where the flow route has potential loops (*e.g.,* the route shown in Fig. 7(b)), we assume that Algorithm 4 is not optimal, *i.e.*, there exists an algorithm that can generate more independent suboperation nodes than Algorithm 4. We assume $O^{f_i}_e$ is an extra independent suboperation node produced by the optimal algorithm. According to the property of the independent suboperation node, the target subpath deleted by $O^{f_i}_e$ cannot be in a loop, while the original subpath added by $O^{f_i}_e$ is not necessary. If the original subpath deleted by $O^{f_i}_e$ is not involved in a loop, $O^{f_i}_e$ has the same properties as the suboperation nodes shown in Fig. 7(a), *i.e.,* both the original subpath and target subpath of $O^{f_i}_e$ are not involved in loops. Therefore, $O^{f_i}_e$ is one of the independent suboperation nodes produced by Algorithm 4, which contradicts the assumption. If the original subpath deleted by $O^{f_i}_e$ is involved in loops (*e.g.,* the subpath $\langle R_2, R_3\rangle$ or $\langle R_3, R_4\rangle$ in Fig. 7(b)), the *mod* operation in $O^{f_i}_e$ has the following properties: (1) the target subpath added by the *mod* operation is not involved in a loop, and (2) the original subpath deleted by the *mod* operation is involved in a loop. However, Algorithm 4 can add each *mod* operation satisfying these properties to an independent operation node. Therefore, $O^{f_i}_e$ is one of the independent suboperation nodes produced by Algorithm 4, which contradicts the assumption. ■

Theorem 3. *The longest dependency chain of suboperation nodes produced by Algorithm 4 is* 2*.*

Proof: We prove the theorem by constructing a contradiction. We assume that Algorithm 4 produces three suboperation nodes O_1, O_2, and O_3, where O_3 depends on O_2, and O_2 depends on O_1. In this case, the dependency chain of these nodes is 3. The above dependency relationships indicate that the original subpath deleted by O_1 and the target subpath added by O_2 are involved in a potential loop. Similarly, the original subpath deleted by O_2 and the target subpath added by O_3 are involved in a loop. Therefore, O_2 contains a *mod* operation to shift the flow from its original subpath to target subpath, where the target subpath and the initial subpath are in two different loops. It contradicts the generation rules of suboperation nodes in Algorithm 4 (lines 11-13), since Algorithm 4 divides such a *mod* operation into two operations and adds the divided operations to two suboperation nodes. ■

After suboperation nodes are produced, we update each of them in the following way. We execute all *add* operations in the suboperation node simultaneously. Then we execute the *mod* operation, making the flow route on its target subpath. The corresponding residual link resources are updated. Finally, we execute all *del* operations in the suboperation node to delete the forwarding rules for the original subpath of the flow.

Theorem 4. *The update of suboperation nodes is blackhole- and loop-free.*

Proof: The update of the suboperation node first needs to execute the operations to forward flows to its target subpath. Then operations that delete the forwarding rules for the original subpath of the flow are executed. The above procedures ensure that packets always have the forwarding rules on switches, the blackhole-free condition is guaranteed. Besides, Algorithm 4 always assigns update operations that may cause a forwarding loop to two suboperation nodes. We restrict the update order of these two suboperation nodes, which ensures the loop-free condition. ■

E. Command Scheduling

After dividing the operation node into several independent suboperation nodes, the parallelism of the ODG has been improved. According to the ODG with suboperation nodes, Algorithm 5 aims to schedule and execute a set of update commands in order to ensure consistency. We describe Algorithm 5 in detail.

In Algorithm 5, we use θ and $\tilde{\theta}$ to denote the set of candidate update nodes and the set of formal update nodes. Initially, we add $O_j^{f_i}$ which only contains *del* operations to $\tilde{\theta}$. Coeus updates such operation nodes directly since executing *del* operations release link resources (lines 3-4). Then, we find the operation nodes which have sufficient resources to update and put them into θ (lines 5-7). The candidate update nodes cannot be updated simultaneously since the residual link resources may be insufficient. To select the operation nodes which can be updated at once, we rank $O_j^{f_i} \in \theta$ in descending order of out-degree and add the congestion-free operation nodes to $\tilde{\theta}$ (lines 8-11). Selecting the operation nodes with high out-degree because updating these nodes at the same time will release more link resources. If the link congestion is unavoidable, *i.e.*, flows occupy insufficient resources of links mutually, deadlocks occur in the ODG. To assign the update order of operation nodes involved in a deadlock, for each operation node, we calculate the throughput loss ratio Φ caused by the update of the operation node. Similar to [35], we select the operation node with minimum Φ and update it by limiting the flow rate to $d^{f_i}(1-\Phi)$ (lines 13-16). Once a set of update commands are sent, Coeus checks whether the new update event arises and decides the next update step.

Algorithm 5 Command Scheduling

```
Input: The operation dependency graph
Output: A set of update commands
 1: $\theta = \tilde{\theta} = \emptyset$;
 2: for each operation node $O_j^{f_i}$ do
 3:     if there is no add and mod operations in $O_j^{f_i}$ then
 4:         $\tilde{\theta} = \tilde{\theta} \cup O_j^{f_i}$;
 5:     else
 6:         if link resources are sufficient for executing operations in $O_j^{f_i}$ then
 7:             $\theta = \theta \cup O_j^{f_i}$;
 8: Rank $O_j^{f_i}$ in $\theta$ in descending order according to their out-degrees;
 9: for each $O_j^{f_i}$ in $\theta$ do
10:     if link resources are sufficient for executing operations in $\tilde{\theta}+O_j^{f_i}$ then
11:         $\tilde{\theta} = \tilde{\theta} \cup O_j^{f_i}$;
12: Update operation nodes in $\tilde{\theta}$;
13: if a deadlock occurs then
14:     for each $O_j^{f_i}$ involved in a deadlock do
15:         Calculate throughput loss ratio $\Phi = \frac{d^{f_i} - min\{r_{u,v}\}}{d^{f_i}}$, where
            $min\{r_{u,v}\}$ is the minimum residual link capacity among the links
            that need to be occupied by the update of $O_j^{f_i}$;
16:     Update $O_j^{f_i}$ with minimum $\Phi$ and limit the rate of $f_i$ to $d^{f_i}(1-\Phi)$;
```

Theorem 5. *Coeus always produces a blackhole-free, loop-free, and congestion-free update sequence.*

Proof: According to Theorem 4, suboperation nodes can be updated in blackhole-free and loop-free manners. Besides, Algorithm 5 always updates operation nodes with sufficient link resources. For the condition where the congestion is unavoidable, *i.e.,* deadlocks occur, we limit the rate of flow to fit the residual link resources. Therefore, the update process will never congest the link. ■

F. Analysis of Time Complexity

Now we analyze the time complexity of Coeus. We assume that an update event UE containing N update operations arrives. UE needs to update M flows, *i.e.,* there are M operation nodes in the dependency graph. We build the dependency relationship for each update operation and composite the operation if the operation is redundant. It costs $O(1)$ to check whether an operation is redundant. Therefore, the cost of dependency relationship construction and operation composition is $O(N)$. Then, we traverse update operations in each operation node to divide an operation node into suboperation nodes. Since the node partition traverses each update operation once, the cost is $O(N)$. Assuming each operation node can be divided into K suboperation nodes. There are MK operation nodes that need to be scheduled. Since the command scheduling updates at least one suboperation node in each round, it needs at most MK rounds to complete the update. In each round, it costs $O(MK)$ to traverse each suboperation node and costs $O(MK\log(MK))$ to sort the suboperation nodes based on their out-degrees. Besides, it costs $O(MK)$ to determine the suboperation nodes in $\tilde{\theta}$ and costs $O(MK)$ to select the suboperation nodes involved in

TABLE V
CORRECT OPERATIONS OF UE_n, WHERE UE_m APPEARED EARLIER THAN UE_n, AND UE_m HAS BEEN EXECUTED

UE_n \ UE_m	$o^{f_i}_{add}$	$o^{f_i}_{del}$	$o^{f_i}_{mod}$
$o^{f_i}_{add}$	$\tilde{o}^{f_i}_{mod}$	$o^{f_i}_{add}$	$\tilde{o}^{f_i}_{mod}$
$o^{f_i}_{del}$	$o^{f_i}_{del}$	$\emptyset$	$o^{f_i}_{del}$
$o^{f_i}_{mod}$	$o^{f_i}_{mod}$	$\tilde{o}^{f_i}_{add}$	$o^{f_i}_{mod}$

TABLE VI
CORRECT OPERATIONS OF UE_n, WHERE UE_m APPEARED EARLIER THAN UE_n, AND UE_m HAS NOT BEEN EXECUTED

UE_n \ UE_m	$o^{f_i}_{add}$	$o^{f_i}_{del}$	$o^{f_i}_{mod}$
$o^{f_i}_{add}$	$o^{f_i}_{add}$	$\emptyset \Vert \tilde{o}^{f_i}_{mod}$	$o^{f_i}_{add}$
$o^{f_i}_{del}$	$\emptyset$	$o^{f_i}_{del}$	$o^{f_i}_{del}$
$o^{f_i}_{mod}$	$\tilde{o}^{f_i}_{add}$	$o^{f_i}_{mod}$	$\tilde{o}^{f_i}_{mod}$

deadlocks to execute. Therefore, the command scheduling costs $O(MK \times (MK + MK\log(MK) + MK + MK)) = O(M^2K^2\log(MK))$. The total time complexity is $O(N) + O(N) + O(M^2K^2\log(MK)) = O(N + M^2K^2\log(MK))$.

G. Discussion

Since the network state of the control plane is not strictly synchronized with the routing state of the data plane, network applications such as traffic engineering may produce invalid update events that contain invalid update operations. A simple way to handle invalid update events is to abandon them and only perform the continuous update with valid update events. However, eliminating invalid update events directly may result in incorrect or missing forwarding rules on switches.

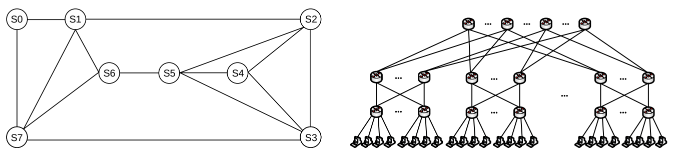
A more reliable method is to transform invalid operations into correct ones. Specifically, we assume that UE_m is a previously arrived update event and UE_n is a newly arriving update event. The update operations in UE_n are not necessarily valid. The transformation of invalid operations is to ensure that flows can be routed along the target path required by the update event UE_n when all update operations (including valid operations and converted operations) in UE_n are executed.

Table V and Table VI show the transformation results of invalid update operations, where the red font represents the correct update operation transformed by the invalid update operation in UE_n, and the black font represents the valid update operation. Table V characterizes the condition where the invalid update event UE_n occurs when UE_m has been executed. In this case, the update operations in UE_m and UE_n cannot be composed. If an *add* operation $o^{f_i}_{add} \in UE_n$ arises when $o^{f_i}_{add} \in UE_m$ has been executed, to ensure that f_i is routed on the path specified by UE_n, we transform $o^{f_i}_{add} \in UE_n$ into a *mod* operation $\tilde{o}^{f_i}_{mod} \in UE_n$. The *mod* operation $\tilde{o}^{f_i}_{mod}$ deletes the forwarding rule of $o^{f_i}_{add} \in UE_m$ and add the forwarding rule of $o^{f_i}_{add} \in UE_n$. Then, we add $\tilde{o}^{f_i}_{mod}$ to O^{f_i} and construct the corresponding relationships for $\tilde{o}^{f_i}_{mod}$ in the ODG. Similarly, other invalid update operations can transform to correct update operations shown in Table V. Table VI characterizes the condition where the invalid update event UE_n occurs when UE_m has not been executed. Algorithm 3 only composites valid update operations. For invalid update operations, we replace unexecuted operations in UE_m with operations in UE_n and construct dependency relationships for operations in UE_n in the ODG to ensure that Coeus can execute operations in UE_n and flows can be routed on the paths specified by UE_n.

VII. EXPERIMENTAL EVALUATION

In this section, we conduct large-scale simulation experiments to verify the performance of Coeus.

Methodology: Our experiments run on a PC with Intel core i5-7200U@2.71GHz quad-core processor and 8G of memory. We evaluate Coeus on two common topologies: (1) The Microsoft's WAN topology (*i.e.*, SWAN) [27] with 8 switches shown in Fig. 8(a). (2) The 8-pods fat-tree [30] with 16 core switches, 64 aggregate switches, 64 edge switches, and 128 hosts shown in Fig. 8(b). We set the link capacity of each topology to be 1-Gbps and generate different numbers of updated flows (*i.e.*, $100, 200, 400, 600$) with random source and destination in the network. We generate continuously arriving update events with different arrival rates λ to modify the routes of flows. For each update event that needs flows to be updated from their initial routes to target routes, we determine the demands of flows in the following way. According to the initial routes of flows, we search for the link that routed through the maximum number of flows and treat this link as the bottleneck. We divide the capacity of the bottleneck equally as the demand of each flow routing through this link. Then we calculate the residual capacity of each link. We iteratively perform the above operations and obtain the demand of each flow on its initial route. We adopt the same method to determine the flow demands on their target routes. The demand of each flow is set to the minimum value of demands calculated from its initial and final routes. According to the test of commodity switches [27], the execution time of insertion, deletion, and modification operations is set to $5ms, 5ms, 10ms$, respectively. The RTT between the controller and data plane switches is set to $50ms$.



(a) Microsoft's WAN topology. (b) Fat-tree topology.

Fig. 8. Network topologies.

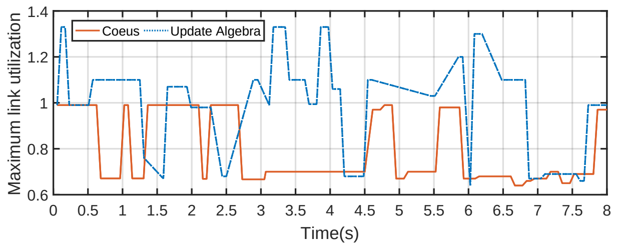


Fig. 9. Maximum link utilization.

We compare the performance of Coeus with "Update Algebra" [28] and Cupid [20]. Specifically, "Update Algebra" models each operation as a set projection. By leveraging the

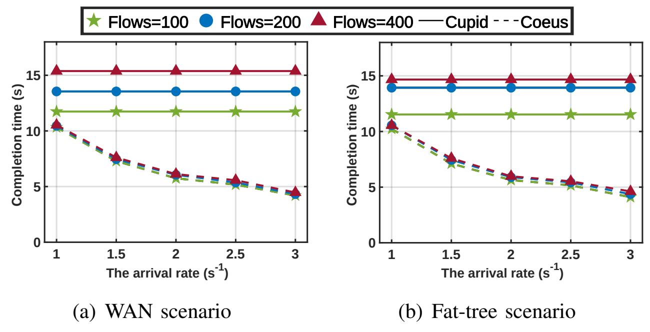


Fig. 10. Update time.

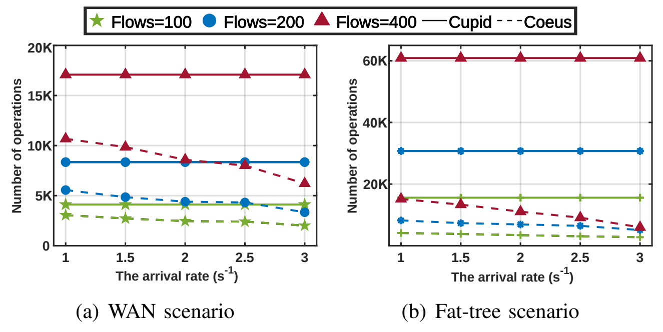


Fig. 11. Number of executed operations.

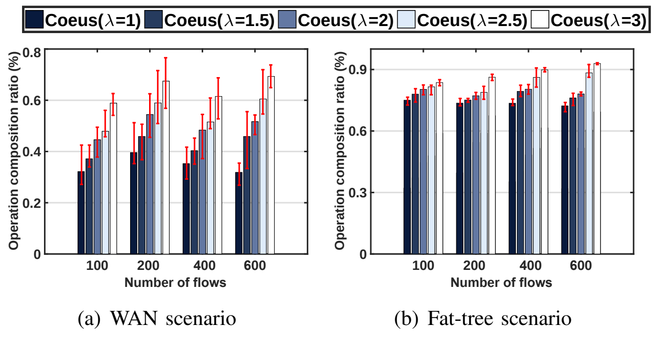


Fig. 12. Operation composition ratio.

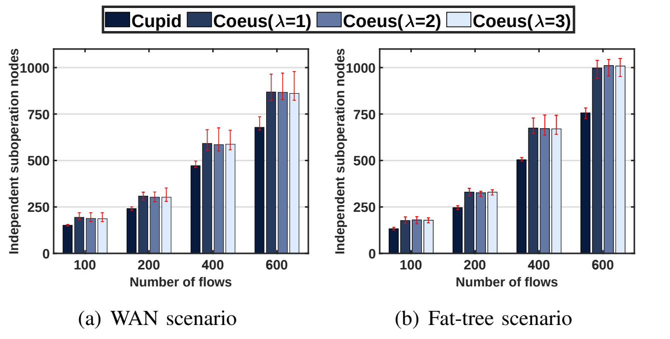


Fig. 13. The number of independent suboperation nodes.

properties of abstract algebra, "Update Algebra" selects the subsets of projections for execution in order. "Update Algebra" can process continuous update events in time. However, it only ensures blackhole-free and loop-free conditions. Cupid is the state-of-the-art approach to achieve the single update. Given the initial and target routes of flows, Cupid divides the global dependencies among flows into several independent segments and identifies the critical nodes which may cause potential link congestion. By updating independent segments in parallel and updating the critical nodes serially, Cupid ensures the blackhole-free, loop-free, and congestion-free properties. However, Cupid can only process one update event at a time. We repeat the experiments 10 times for each data set to generate the results discussed below.

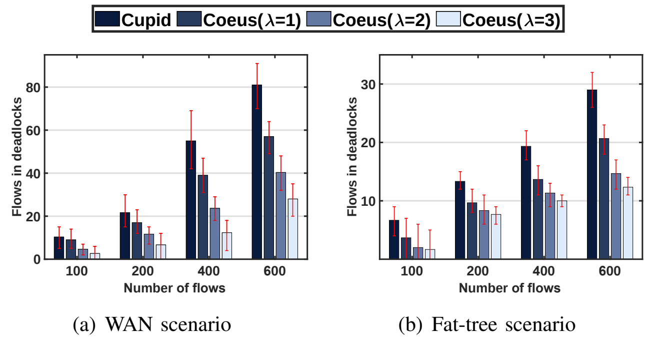
Experiment results: We first investigate the maximum link utilization of Coeus and "Update Algebra" shown in Fig. 9. We do this simulation with 100 updated flows in the SWAN topology. The arrival rate λ of update events is set to $3/s$. Once the maximum link utilization is beyond one, it means that link congestion occurs. Fig. 9 shows that Coeus always guarantees that the maximum link utilization is less than or equal to one, ensuring the congestion-free condition during the update. In contrast, the maximum link utilization of "Update Algebra" sometimes is over 1.3. The link overload results in packet loss and the degradation of network performance.

Next, we generate 10 continuous update events and compare Coeus against Cupid in multiple dimensions. Fig. 10 shows the time to complete all update events. We observe that the update time of Coeus is shorter than that of Cupid. Moreover, the gap between the update completion time of Coeus and Cupid is getting larger with the arrival rate increasing. Specifically, in the SWAN topology with 100 updated flows, compared with Cupid, Coeus improves the update speed by 13.2%, 61.4%, 120.7%, 130.1%, 179.8% when the arrival rate of update events $\lambda = 1/s \sim 3/s$. The reason is that Cupid executes update events serially while the operation composition in Coeus can reduce the number of redundant operations, and the node partition in Coeus can execute update operations in parallel.

Fig. 11 shows the number of executed update operations after finishing 10 continuous update events. The number of executed update operations of Cupid is constant with the arrival rate varying since Cupid needs to execute all update operations of each update event. In contrast, Coeus executes fewer operations. This benefits from the operations composition, which composites unexecuted update operations into the fewer equivalent ones. Furthermore, the number of executed operations in Coeus decreases with the arrival rate increasing. It is because that more unexecuted operations that can be composed. When the arrival rate $\lambda = 3/s$ and the number of updated flows varies from 100 to 400, Coeus reduces 51.2%, 59.9%, 64.2% of executed operations compared with Cupid in the SWAN. We also observe that executed update operations in the fat-tree are much more than that of in the SWAN since flows in the fat-tree are routed through more links, requiring more update operations. In the fat-tree with 100, 200, 400 flows, Coeus reduces 82.2%, 83.4%, 90.1% of executed operations compared with Cupid when the arrival rate $\lambda = 3$. This demonstrates that Coeus can execute fewer update operations while maintaining consistency.

Fig. 12 reflects the average operation composition ratio. With the arrival rate increasing, the composition ratio increases. Specifically, in the SWAN, Coeus composites at least 30% redundant operations of each update event, while in the fat-tree, at least 70% operations can be composed by Coeus. The composition ratio in the fat-tree is higher than that of in the SWAN because the hierarchical structure of the fat-tree increases the probability of operation composition.

When 10 continuous update events are completed, we count the average number of independent suboperation nodes produced in each update event in Fig. 13. Compared with Cupid, Coeus generates more independent suboperation nodes both in the SWAN and the fat-tree since we have proved that the longest dependency chain of suboperation nodes produced by Coeus is 2, while the suboperation nodes produced by Cupid may have long dependency chains.



(a) WAN scenario (b) Fat-tree scenario

Fig. 14. Number of flows in deadlocks.

Fig. 14 shows the number of flows in deadlocks in 10 continuous update events. We observe that Cupid produces more flows in deadlocks in both two topologies. The reason is that suboperation nodes produced by Cupid have long dependency chains. The dependent suboperation nodes can only be updated along the dependency chain. In contrast, Coeus produces more independent suboperation nodes that can be updated in parallel. Coeus can release more link resources compared with Cupid, so there are fewer flows in deadlocks.

VIII. Conclusion

In this paper, we studied the continuous update problem in SDNs. We proposed Coeus to respond to continuous update events in time while guaranteeing the blackhole-free, loop-free, and congestion-free properties simultaneously during the update procedure. We developed a set of efficient algorithms to handle update events and speed up the update process. Extensive simulations demonstrate that Coeus can reduce the makespan and redundant update operations significantly.

References

[1] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," in *ACM SIGCOMM*, 2013, pp. 3–14.

[2] J. W. Jiang, T. Lan, S. Ha, M. Chen, and M. Chiang, "Joint vm placement and routing for data center traffic engineering," in *IEEE INFOCOM*, 2012, pp. 2876–2880.

[3] J. Zheng, Q. Zheng, X. Gao, and G. Chen, "Dynamic load balancing in hybrid switching data center networks with converters," in *IEEE ICPP*, 2019, pp. 1–10.

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

[5] J. Zheng, H. Xu, X. Zhu, G. Chen, and Y. Geng, "Sentinel: Failure recovery in centralized traffic engineering," *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 1859–1872, 2019.

[6] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *ACM SIGCOMM*, 2012, pp. 323–334.

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

[8] J. McClurg, H. Hojjat, P. Černỳ, and N. Foster, "Efficient synthesis of network updates," in *Acm Sigplan Notices*, vol. 50, no. 6, 2015, pp. 196–207.

[9] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *ACM SIGCOMM*, 2010, pp. 267–280.

[10] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *ACM SIGCOMM*, 2009, pp. 51–62.

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

[12] J. Zheng, X. Hong, X. Zhu, G. Chen, and Y. Geng, "We've got you covered: Failure recovery with backup tunnels in traffic engineering," in *ICNP*, 2016, pp. 1–10.

[13] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, and D. Yang, "Experience-driven networking: A deep reinforcement learning based approach," in *IEEE INFOCOM*, 2018, pp. 1871–1879.

[14] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *ACM CoNEXT*, 2011, p. 8.

[15] K.-T. Foerster, S. Schmid, and S. Vissicchio, "Survey of consistent software-defined network updates," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1435–1461, 2018.

[16] S. Brandt, K.-T. Förster, and R. Wattenhofer, "On consistent migration of flows in sdns," in *IEEE INFOCOM*, 2016, pp. 1–9.

[17] S. Akhoondian Amiri, S. Dudycz, S. Schmid, and S. Wiederrecht, "Congestion-free rerouting of flows on dags," in *Springer ICALP*, 2018.

[18] J. Zheng, Q. Ma, C. Tian, B. Li, H. Dai, H. Xu, G. Chen, and Q. Ni, "Hermes: Utility-aware network update in software-defined wans," in *IEEE ICNP*, 2018, pp. 231–240.

[19] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *ACM HotNets*, 2013, pp. 1–7.

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

[21] A. Basta, A. Blenk, S. Dudycz, A. Ludwig, and S. Schmid, "Efficient loop-free rerouting of multiple sdn flows," *IEEE/ACM Transactions on Networking*, vol. 26, no. 2, pp. 948–961, 2018.

[22] K.-T. Förster, R. Mahajan, and R. Wattenhofer, "Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes," in *IEEE IFIP Networking*, 2016, pp. 1–9.

[23] H. Xu, Z. Yu, X.-Y. Li, C. Qian, L. Huang, and T. Jung, "Real-time update with joint optimization of route selection and update scheduling for sdns," in *IEEE ICNP*, 2016, pp. 1–10.

[24] H. Xu, Z. Yu, X.-Y. Li, L. Huang, C. Qian, T. Jung, H. Xu, Z. Yu, X.-Y. Li, L. Huang *et al.*, "Joint route selection and update scheduling for low-latency update in sdns," *IEEE/ACM Transactions on Networking*, vol. 25, no. 5, pp. 3073–3087, 2017.

[25] J. Zheng, H. Xu, G. Chen, and H. Dai, "Minimizing transient congestion during network update in data centers," in *ICNP*, 2015, pp. 1–10.

[26] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese *et al.*, "Conga: Distributed congestion-aware load balancing for datacenters," in *ACM SIGCOMM*, 2014, pp. 503–514.

[27] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," in *ACM SIGCOMM*, 2014, pp. 539–550.

[28] G. Li, Y. R. Yang, F. Le, Y.-s. Lim, and J. Wang, "Update algebra: Toward continuous, non-blocking composition of network updates in sdn," in *IEEE INFOCOM*, 2019, pp. 1081–1089.

[29] X. Wen, C. Diao, X. Zhao, Y. Chen, L. E. Li, B. Yang, and K. Bu, "Compiling minimum incremental update for modular sdn languages," in *ACM HotSDN*, 2014, pp. 193–198.

[30] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM*, 2008, pp. 63–74.

[31] A. Ludwig, J. Marcinkowski, and S. Schmid, "Scheduling loop-free network updates: It's good to relax!" in *ACM PODC*, 2015, pp. 13–22.

[32] K. T. Foerster, A. Ludwig, J. Marcinkowski, and S. Schmid, "Loop-free route updates for software-defined networks," *IEEE/ACM Transactions on Networking*, vol. PP, no. 99, pp. 328–341, 2018.

[33] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," in *ACM SIGCOMM*, 2013, pp. 15–26.

[34] R. Gandhi, O. Rottenstreich, and X. Jin, "Catalyst: Unlocking the power of choice to speed up network updates," in *ACM CoNEXT*, 2017, pp. 276–282.

[35] K.-R. Wu, J.-M. Liang, S.-C. Lee, and Y.-C. Tseng, "Efficient and consistent flow update for software defined networks," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, pp. 411–421, 2018.

[36] T. Mizrahi, E. Saat, and Y. Moses, "Timed consistent network updates," in *ACM SOSR*, 2015, pp. 21:1–21:14.

[37] T. Mizrahi, O. Rottenstreich, and Y. Moses, “Timeflip: Scheduling network updates with timestamp-based tcam ranges,” in *IEEE INFOCOM*, 2015, pp. 2551–2559.

[38] T. Mizrahi and Y. Moses, “Software defined networks: It’s about time,” in *IEEE INFOCOM*, 2016, pp. 1–9.

[39] J. Zheng, G. Chen, S. Schmid, H. Dai, and J. Wu, “Chronus: Consistent data plane updates in timed sdns,” in *IEEE ICDCS*, 2017, pp. 319–327.

[40] J. Zheng, B. Li, C. Tian, K.-T. Foerster, S. Schmid, G. Chen, and J. Wu, “Scheduling congestion-free updates of multiple flows with chronicle in timed sdns,” in *IEEE ICDCS*, 2018, pp. 12–21.

[41] J. Zheng, B. Li, C. Tian, K.-T. Foerster, S. Schmid, G. Chen, J. Wu, and R. Li, “Congestion-free rerouting of multiple flows in timed sdns,” *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 5, pp. 968–981, 2019.

[42] T. D. Nguyen, M. Chiesa, and M. Canini, “Decentralized consistent updates in sdn,” in *ACM SOSR*, 2017, pp. 21–33.

[43] J. Zheng, H. Xu, G. Chen, H. Dai, and J. Wu, “Congestion-minimizing network update in data centers,” *IEEE Transactions on Services Computing*, 2016.

[44] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. E. Li, and M. Thottan, “Measuring control plane latency in sdn-enabled switches,” in *ACM SOSR*, 2015, pp. 25:1–25:6.

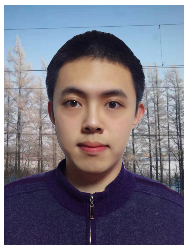
[45] B. Heller, C. Scott, N. McKeown, S. Shenker, A. Wundsam, H. Zeng, S. Whitlock, V. Jeyakumar, N. Handigol, J. McCauley *et al.*, “Leveraging sdn layering to systematically troubleshoot networks,” in *ACM HotSDN*, 2013, pp. 37–42.

[46] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, “Ofrewind: Enabling record and replay troubleshooting for networks,” in *USENIX ATC*, 2011, pp. 327–340.

Xin He received the B.S. degree from the Inner Mongolia University and M.E. degree from the Beijing Institute of Technology, and the Ph.D. degree from Nanjing University. He is currently an assistant professor with the School of Computer Science at Nanjing University of Posts and Telecommunications. His research interests are in the areas of SDN, data center network and edge computing. He is a student member of IEEE.

Jiaqi Zheng is currently a Research Assistant Professor from Department of Computer Science and Technology, Nanjing University, China. His research area is computer networking, particularly data center networks, SDN/NFV, machine learning system and online optimization. He was a Research Assistant at the City University of Hong Kong in 2015 and collaborated with Huawei Noah’s Ark Lab. He visited CIS center at Temple University in 2016. He received the Best Paper Award from IEEE ICNP 2015, Doctorial Dissertation Award from ACM SIGCOMM China 2018, the First Prize of Jiangsu Science and Technology Award in 2018, Doctorial Dissertation Award from Jiangsu Province and Nanjing University in 2019. He is a member of ACM and IEEE.

Haipeng Dai received the B.S. degree in the Department of Electronic Engineering from Shanghai Jiao Tong University, Shanghai, China, in 2010, and the Ph.D. degree in the Department of Computer Science and Technology in Nanjing University, Nanjing, China, in 2014. His research interests are mainly in the areas of wireless charging, mobile computing, and data mining. He is a research assistant professor in the Department of Computer Science and Technology in Nanjing University. His research papers have been published in many prestigious conferences and journals such as ACM MobiSys, ACM MobiHoc, ACM VLDB, ACM SIGMETRICS, ACM UbiComp, IEEE INFOCOM, IEEE ICDCS, IEEE ICNP, IEEE SECON, IEEE IPSN, IEEE JSAC, IEEE/ACM TON, IEEE TMC, IEEE TPDS, and IEEE TOSN. He is an IEEE and ACM member. He serves/ed as Poster Chair of the IEEE ICNP’14, Track Chair of the ICCCN’19, TPC member of the IEEE INFOCOM’20, IEEE IWQoS’19, IEEE ICNP’14, IEEE ICC’14-18, IEEE ICCCN’15-18 and the IEEE Globecom’14-18. He received Best Paper Award from IEEE ICNP’15, Best Paper Award Runner-up from IEEE SECON’18, and Best Paper Award Candidate from IEEE INFOCOM’17.



Chong Zhang is currently an undergraduate student of the software institute of Nanjing University. He has majored in software engineering since he entered Nanjing University in 2017. He is also a student member of IEEE. His research interests include computer communication, SDN and data center transport.

Geng Li is an Associate Research Scientist in the Department of Computer Science at Yale University. He received the Ph.D. degree in wireless communications from Peking University in June 2016, advised by Professor Yuping Zhao, the B.S. degree in electronics engineering, the B.A. degree in economics (double major) from Peking University in 2011. He works in the areas of computer networks and wireless communications.

Wanchun Dou received the Ph.D. degree in mechanical and electronic engineering from Nanjing University of Science and Technology, China, in 2001. From Apr. 2001 to Dec. 2002, he did his postdoctoral research in the Department of Computer Science and Technology, Nanjing University, China. Now, he is a full professor of the State Key Laboratory for Novel Software Technology, Nanjing University, China. From Apr. 2005 to Jun. 2005 and from Nov. 2008 to Feb. 2009, he respectively visited the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, as a visiting scholar. Up to now, he has published more than 60 research papers in international journals and international conferences. His research interests include cloud computing, big data, and service computing.

Wajid Rafique is currently pursuing his Ph.D. degree in computer science at Nanjing University, China. He received the BS (computer science) degree from Virtual University of Pakistan and the MS (software engineering) degree from National University of Sciences and Technology, Pakistan. His research works have been appeared in several prestigious international journals and the top tier conferences. His research interests include big data services, machine learning, mobile cloud computing.

Qiang Ni received the B.Sc., M.Sc., and Ph.D.degrees in engineering from the Huazhong University of Science and Technology, Wuhan, China. He is currently a Professor and the Head of Communication Systems Research Group, InfoLab21, School of Computing and Communications, Lancaster University, Lancaster, U.K. His research interests include future generation communications and networking systems, including green communications, cloud systems, cognitive radio network systems, heterogeneous networks, 5G and SDN, IoT, and big data analytics.

Guihai Chen received B.S. degree in computer software from Nanjing University in 1984, M.E. degree in computer applications from Southeast University in 1987, and Ph.D. degree in computer science from the University of Hong Kong in 1997. He is a professor and deputy chair of the Department of Computer Science, Nanjing University, China. He had been invited as a visiting professor by many foreign universities including Kyushu Institute of Technology, Japan in 1998, University of Queensland, Australia in 2000, and Wayne State University, USA during Sept. 2001 to Aug. 2003. 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.