Coeus: Consistent and Continuous Network Update in Software-Defined Networks

Xin He[†], Jiaqi Zheng[†], Haipeng Dai[†], Chong Zhang[†], Wajid Rafique[†], Geng Li[‡], Wanchun Dou[†], Qiang Ni[♯],
[†]State Key Laboratory for Novel Software Technology, Nanjing University, China
[‡]Department of Computer Science, Yale University, USA
[♯]School of Computing and Communications, Lancaster University, UK

***Abstract*—Network update enables Software-Defined Networks (SDNs) to optimize the data plane performance via southbound APIs. The single update between the initial and the final network states fail to handle high-frequency changes or the burst event during the update procedure in time, leading to prolonged update time and inefficiency. On the contrary, the continuous update can respond to the network condition changes at all times. However, existing work, especially "Update Algebra" can only guarantee blackhole- and loop-free. The congestion-free property cannot be respected during the update procedure. In this paper, we propose Coeus, a continuous network update system while maintaining blackhole-, loop- and congestion-free simultaneously. Firstly, we establish an operation-based continuous update model. Based on this model, we dynamically reconstruct an operation dependency graph to capture unexecuted update operations and the link utilization variations. Subsequently, we develop an operation composition algorithm to eliminate redundant update commands and an operation node partition algorithm to speed up the update procedure. We prove that the partition algorithm is optimal and can guarantee the consistency. Finally, extensive evaluations show that Coeus can improve the makespan by at least 179% compared with state-of-the-art approaches when the arrival rate of update events equals to three times per second.**

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], [13], [14], [34] and failures recovery [7], [29], [33]). Due to high-frequency network condition variations, the controller requires handling continuous update events rapidly to optimize the performance of the data plane [5], [27]. Therefore, the network needs to have the ability to perform update events as soon as possible and guarantee consistency [2], [6], [31] (*i.e.*, blackhole-, loop- and congestion-free) during updates.

Given the initial and the final network states in the data plane, the single update [4], [9], [18], [23], [25], [26], [32] advocates determining a consistent update sequence to shorten the update time. However, they fail to handle high-frequency changes or the burst event during the update procedure. Each update event can only be executed after the completion of the previous update events, leading to prolonged update time and poor performance. Instead of the single update, existing work, especially "Update Algebra" [11] explores the continuous update solution to respond to network condition variations, where the update events arrive in an online manner. "Update Algebra" models operations in continuous update events as a set of projections and selects a feasible subset of projections to execute.

Although "Update Algebra" [11] has the ability to handle continuous update events in an online manner, it can only guarantee blackhole- and loop-free properties. Essentially, "Update Algebra" mainly considers the order of update operations in the flow-based granularity, but different flows compete for the limited link bandwidth resource. The transient congestion may occur during updates, which incurs packets loss and degrades network performance.

In this paper, we initiate the study of consistent continuous network update problem, which can handle the frequent update events in an online manner and guarantee blackhole-, loop- and congestion-free properties simultaneously. To the best of our knowledge, little is known today about how to schedule flows in this problem. Performing the consistent continuous update remains algorithmically challenging. We face three main challenges: (i) Performing continuous updates requires the controller to identify which update commands have been executed, which ones are being executed, and which ones will be executed. Besides, the time-varying network resources and update commands should be jointly considered to produce a consistent update sequence. (ii) The number of possible congestion-free update orders is exponential many even for a single update event [15]. For an update event containing n flows which need to be rerouted, the number of possible update orders is $O(n!)$. Involving multiple update events makes the problem essentially harder. (iii) We need to speed up the continuous update procedure, *i.e.*, increasing the degree of parallelism. We wish to find as many independent parts as possible to execute them simultaneously in a consistent way.

To address the challenges mentioned above, we make the following contributions. Firstly, we develop Coeus, a continuous network update system to maintain consistency. We give an overview of Coeus (Sec. III). Besides, we extend the continuous model proposed in [11], which only captures the data plane update. We establish the operation-based continuous update model, which can capture both the update event in the control plane and forwarding actions in the data plane (Sec. IV). Secondly, we propose a set of algorithms to achieve the consistent continuous update (Sec. V). Specifically, based on the continuous update model, we dynamically reconstruct

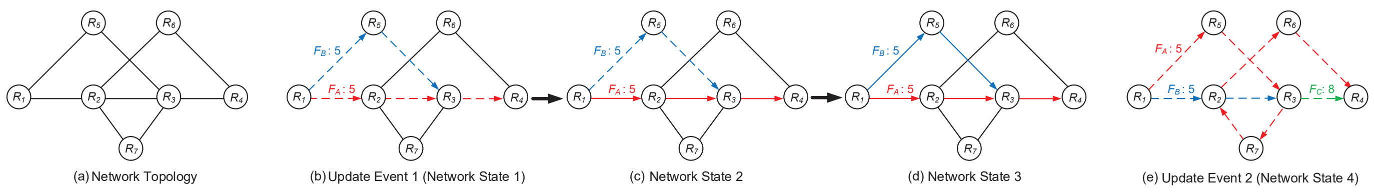


Fig. 1. A continuous update example. Fig. 1(a) shows the network topology. Fig. 1(b)∼Fig. 1(e) denote the update events and network states.

an operation dependency graph to capture utilization variations of each link and dependencies of unexecuted update operations in continuous update events. However, an update operation from the newly arrived event may cancel out an existing unexecuted update operation. We develop the operation composition algorithm in the dependency graph to eliminate redundant operations. By leveraging the dynamic operation dependency graph and operation composition, we solve the first two challenges. To address the third challenge, we design the operation node partition algorithm to split the operation node into a series of independent suboperation nodes. The update of independent suboperation nodes improves the degree of update parallelism since parts of flow path can be updated directly, and the flow does not need to wait until all link resources along its target path are sufficient. We prove that our node partition algorithm is optimal, and the update of suboperation nodes ensures the consistency. Finally, we evaluate Coeus by using experiments in large-scale simulations (Sec. VI). We use two common topologies (*i.e.*, SWAN and fat-tree) to verify the effectiveness of Coeus by changing the arrival rate of update events constantly. The simulations show that Coeus improves the makespan by at least 179% compared with state-of-the-art approaches when the arrival rate of update events equals to three times per second.

II. A Motivating Example

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

Fig. 1(a) shows a network topology containing of seven switches $\{R_1, ..., R_7\}$, where the capacity of link $\langle R_1, R_5\rangle$ equals to 5 units and others equal to 10 units. Two update events UE_1 and UE_2 arrive sequentially. The update event 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. The update event UE_2 will install forwarding rules of flow F_C, where the demand of F_C is 8 unit. Each update event incurs different network states, and each network state captures different network routing information. We use the directed edge to denote the routing of each flow. The dashed line characterizes that the forwarding rules of the flow have not been installed while the solid line characterizes that the forwarding rules of the flow have already been installed. Fig. 1(b) denotes that when the update event UE_1 appears, the controller will assign the path of flows F_A and F_B. However, the corresponding forwarding commands have not been installed in switches. Fig. 1(c) shows that in the network state NS_2, F_A has been updated to its target path while F_B has not been updated. The network state NS_3 shown in Fig. 1(d) indicates that F_B has been updated, and the update event UE_1 is finished since all of the flows in UE_1 are routed through their target paths. Fig. 1(e) shows that when UE_2 occurs after UE_1, the controller will assign the route of flow 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	Acyclic 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
$\widehat{UE}$	Set of unexecuted update operations in UE.
NS	Network state
NS_{init}	Relative initial state
O_I	Set of executed update operations
O_{NI}	Set of unexecuted update operations
O^{f_i}	Set of unexecuted update operations of f_i
o^{f_i}	Update operation of f_i
$R_{u,v}$	Resource nodes in the operation dependency graph
$r_{u,v}$	Residual resource of the resource node

In the single update scheme, the update event must wait for the completion of previous update events, *e.g.*, if UE_2 occurs after NS_1, UE_2 cannot be responded until UE_1 is finished. This serial update scheme prolongs the makespan significantly. We aim to develop a system to respond random and continuous update events. However, dealing with the continuous update in an online manner and guaranteeing blackhole-, loop- and congestion-free simultaneously are full of challenges.

For example, under the condition where UE_1 has installed the forwarding rule of F_A on the switch R_2 and has not installed the forwarding rule of F_A on the switch R_3, UE_2 occurs. Then, the update event UE_1 encourages R_3 to install the following forwarding rule for F_A (*i.e.*, forwarding F_A to switch R_4) while the update event UE_2 encourages R_3 to install another forwarding rule for F_A (*i.e.*, forwarding F_A to switch R_7). Therefore, handling continuous update events simply 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.*, maintaining blackhole- and loop-free). For example, when UE_2 occurs after NS_2, the switch R_5 should has the forwarding rule of F_A before F_A routes through the link $\langle R_1, R_5\rangle$. Besides, the forwarding rules of F_A should be installed on switch R_3 and R_7 after the original forwarding rule on switch

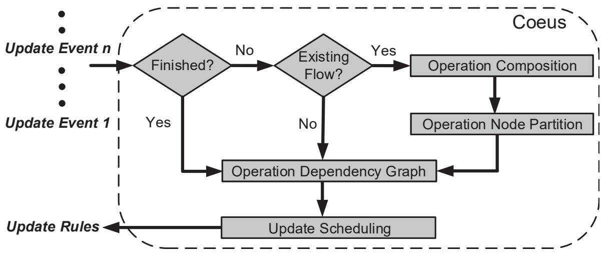


Fig. 2. Overview of Coeus.

R_2 is removed. Otherwise, the transient loop $\langle R_2, R_3\rangle \to \langle R_3, R_7\rangle \to \langle R_7, R_2\rangle$ 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 will occur due to the limited link resource of $\langle R_1, R_5\rangle$. Similarly, F_C should be updated after F_A. To speed up the continuous update procedure, the concurrency of update should be taken into consideration. We will introduce the corresponding method in detail in Sec. V-B.

III. Coeus Overview

To solve the problems mentioned above, we propose a system, Coeus, to achieve the consistent and continuous update. In this section, we will introduce an overview of Coeus.

The entire workflow of Coeus is shown in Fig. 2. In SDNs, the update events come in an online manner. To handle continuous updates, the centralized controller firstly judges whether the previous update events are finished. Once the condition is true, we build the operation dependency graph which has a set of resource nodes and a set of operation nodes to capture the link utilization variations and the dependencies of update operations. Otherwise, we make further judgment about whether updated flows which involved in the new update event have already existed in the data plane. If flows do not have forwarding paths at this moment (*e.g.*, F_C in Fig. 1(e)), we insert new operation nodes of flows in the operation dependency graph and construct corresponding relationships. Otherwise, update operations are added into existing operation nodes, and we dynamically reconstruct the dependency relationship in the dependency graph. However, adding update operations into operation nodes continuously may lead to redundant operations in operation nodes. By compositing update operations, Coeus replaces redundant operations into equivalent ones. Subsequently, Coeus designs the operation node partition algorithm to increase the parallelism of updates and speed up the continuous update procedure. The node partition algorithm divides the operation node into several suboperation nodes which can be updated independently in a blackhole- and loop-free manner. Finally, by leveraging the dynamic dependency graph, our update scheduling scheme sends a set of update commands to the data plane.

IV. Continuous Updates Model

After describing Coeus, we will introduce the continuous update model in this section. The general network can be modeled as a graph $G = (V, E)$, where V is the set of switches and $E \subseteq V^2$ is the set of 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 flow f. Each flow $f_i \in F$ is an unsplittable flow with demand d^{f_i}.

In the control plane, continuous update events $\{UE\}$ arrive randomly. Each update event $UE = \{o^{f_i} | \forall f_i \subseteq F\}$ contains a set of update operations $\{o^{f_i}\}$ to assign the route of each flow f_i. We define $\widehat{UE} \in UE$ to characterize the set of operations that have not been executed in the data plane. The network states $\{NS\}$ present the current route information captured by the control plane, where $NS = \{O_I, O_{NI}\}$. O_I denotes the set of operations that have been executed while O_{NI} denotes the set of operations that have not been executed in the data plane. When the n-th update event arises, O_I and O_{NI} can be expressed as follows.

$$O_I = \{(UE_1 - \widehat{UE_1}) \cup (UE_2 - \widehat{UE_2}) ... \cup (UE_n - \widehat{UE_n})\} \quad (1)$$

$$O_{NI} = \{\widehat{UE_1} \cup \widehat{UE_2} ... \cup \widehat{UE_n}\} \quad (2)$$

In this paper, we focus on scheduling unexecuted update operations orderly. We introduce O^{f_i} to denote the set of unexecuted operations of f_i, *i.e.*, $O^{f_i} = O^{f_i}_{NI}$. Different from the single update with the deterministic initial and final state, continuous update events with unknown update operations make it hard to determine the final state of the network. We define the relative initial state as follows.

Definition IV.1. Relative Initial State: *The relative initial state represents that all operations of the previous update events $\{UE\}$ have been executed in the data plane, i.e., all of the previous update events $\{UE\}$ are completed.*

Now we discuss the mapping relationship between update operations in the control plane and forwarding actions in the data plane. The controller sends update operations to individual switches. Once an update operation o^{f_i} is executed on the specified switch, it will execute the corresponding action if f_i matches the flow entry in the switch flow table. We classify update operations into three types $\{add, mod, del\}$. At the specified switch u, once the *add* operation $o^{f_i}_{add}$: $add\{fwd\_v\}$ is executed, the forwarding rule of f_i is added, and f_i will be forwarded to switch v. Once the *del* operation $o^{f_i}_{del}$: $del\{fwd\_v\}$ is executed, the forwarding rules of f_i is deleted from the switch. Similarly, the *mod* operation $o^{f_i}_{mod}$: $mod\{fwd\_v \to fwd\_w\}$ denotes that modifying the forwarding rules of f_i enables f_i to be forwarded to switch w instead of v. Indeed, $o^{f_i}_{mod}$ can be treated as a delete operation $o^{f_i}_{del}$ that removes the original forwarding rule, and an add operation $o^{f_i}_{add}$ that adds the new forwarding rule. In the continuous update, not all of the update operations are valid, we define the validity of update operations as follows.

Definition IV.2. Valid Update Operations: *The valid update operations of flow depend on the previous operations of the same flow acting on the same switches, valid update operations enable each switch to have at most one forwarding rule for each flow, and the repeated forwarding rule is forbidden.*

TABLE II
VALID UPDATE OPERATIONS OF UE

Update Event	Update Operation		
UE_p	$o^{f_i}_{add}$	$o^{f_i}_{del}$	$o^{f_i}_{mod}$
UE	$o^{f_i}_{del} \parallel o^{f_i}_{mod}$	$o^{f_i}_{add}$	$o^{f_i}_{del} \parallel o^{f_i}_{mod}$

Specifically, the valid update operations must follow the rule shown in Table II, where UE_p is the previous update events, and UE is a new update event. For example, on the specified switch, if UE_p involves the *add* operation $o^{f_i}_{add}$, the operation in UE should only be $o^{f_i}_{del}$ or $o^{f_i}_{mod}$. Otherwise, there is another *add* operation for f_i acting on the same switch, which is forbidden in the continuous update. If UE_p involves the *del* operation $o^{f_i}_{del}$, the operation in UE should only be $o^{f_i}_{add}$. Since once the *del* operation of f_i is executed, there is no forwarding rule in the switch for f_i that needs to be deleted or modified. Similarly, if UE_p contains the *mod* operation $o^{f_i}_{mod}$, the operation of UE acting for the same flow on the same switch should only be $o^{f_i}_{del}$ or $o^{f_i}_{mod}$.

V. COEUS SCHEDULING

In this section, we will introduce modules of Coeus and present efficient scheduling algorithms to ensure the consistency policy in the continuous update. Before that, we sketch the continuous update process in Algorithm 1.

Our update process starts from a relative initial state NS_{init} defined in Definition IV.1. Once an update event appears after the relative initial state, the initial operation dependency graph $G_D = (V_D, E_D)$ is built (lines 1-5). If the flow has existed in NS_{init}, we apply Algorithm 4 to divide the operation node of the flow into independent suboperation nodes to speed up the update process (lines 6, 7). When the operation dependency graph $G_D \neq \emptyset$, *i.e.*, the update events have not been finished, we check whether a new update event NS arrives (lines 9). Once a new update event occurs, we split updated flows in UE into existing flows and emerging flows. For existing flows, we apply Algorithm 3 and Algorithm 4 to composite the update operations and generate independent suboperation nodes in the operation dependency graph (lines 11-15). For emerging flows, we construct new operation nodes and the corresponding relationships in the operation dependency graph (lines 16-19). According to the operation dependency graph, we produce a set of update commands to update the data plane until $G_D = \emptyset$ (line 20).

A. The Dependency Graph

To capture the variation of link resources and the order of update operations, we construct the operation dependency graph and adjust the dependency graph dynamically to handle continuous update events. We define the operation dependency graph as follows.

Definition V.1. Operation Dependency Graph (ODG): *The operation dependency graph $G_D = (V_D, E_D)$ is a directed graph that captures dependent relationships between update operations of each updated flow and link resources. There are two types of nodes in the ODG: the operation node $O^{f_i} \in V_D$ and the resource node $R_{u,v} \in V_D$. And it contains two types of edges: $\{R_{u,v} \dashleftarrow O^{f_i}\} \in E_D$ and $\{O^{f_i} \dashleftarrow R_{u,v}\} \in E_D$.*

Specifically, the operation node $O^{f_i} \in V_D$ with a set of unexecuted update operations $\{o^{f_i}\}$ is labeled with the flow demand d^{f_i}. The resource node $R_{u,v} \in V_D$ is labeled with the residual link resource $r_{u,v}$. Initially, the link residual resource $r_{u,v}$ equals to the link capacity $c_{u,v}$. The edge $\{O^{f_i} \dashleftarrow R_{u,v}\}$ represents that once the update operation $o^{f_i} \in O^{f_i}$ is executed, the link resource will be occupied by flow f_i. Inversely, the edge $\{R_{u,v} \dashleftarrow O^{f_i}\}$ denotes that the update operation $o^{f_i} \in O^{f_i}$ will release the link resource. In the continuous update process, the operation dependency graph changes dynamically with update operations executed and new update events arriving.

Algorithm 1: Continuous Update Process

```
Input: The update events {$UE$} with updated flows
Output: A set of update commands
 1: $G_D = (V_D, E_D) = \emptyset$;
 2: for each updated flow $f_i$ do
 3:     $V_D \leftarrow V_D \cup O^{f_i}$ with its demand $d^{f_i}$;
 4:     for each operation $o^{f_i} \in O^{f_i}$ do
 5:         Apply Algorithm 2 to obtain the initial ODG;
 6:     if the flow $f_i$ is the existing flow then
 7:         Apply Algorithm 4 to divide operation node $O^{f_i}$ into
            suboperation node $O^{f_i}_j$;
 8: for $G_D \neq \emptyset$ do
 9:     if the new update event $UE$ arrives then
10:         for each flow $f_i \in NS$ do
11:             if the flow $f_i$ is the existing flow then
12:                 if $O^{f_i}$ has been divided into suboperation nodes
                    then
13:                     Treat suboperation nodes as a virtual node;
14:                 Apply Algorithm 3 to composite operations;
15:                 Apply Algorithm 4 to divide $f_i$ into independent
                    suboperation nodes;
16:             else
17:                 $V_D \leftarrow V_D \cup O^{f_i}$ with its demand $d^{f_i}$;
18:                 for each operation $o^{f_i} \in O^{f_i}$ do
19:                     Apply Algorithm 2 to build the dependency
                        relationship;
20:     Apply Algorithm 5 to update the dependency graph;
```

Now we discuss the construction of dependency relationship in Algorithm 2. As Algorithm 1 mentioned, we add the operation node O^{f_i} into the ODG (lines 2, 3). Then, we judge the type of each operation o^{f_i} for f_i and build the corresponding dependency relationship with the resource node $R_{u,v}$ in Algorithm 2 (lines 1-13). For example, if the operation o^{f_i} is an *add* operation, the forwarding rule will be added into the switch u to forward f_i to the link $\langle u, v \rangle$, and the corresponding resource of $\langle u, v \rangle$ will be consumed. We add the resource node $R_{u,v}$ and the edge $\{R_{u,v} \dashleftarrow O^{f_i}\}$ if $R_{u,v}$ does not exist in the ODG (lines 2-4). Similarly, we build the corresponding relationship among operation nodes and resource nodes if the operation is del or mod (lines 5-13). Specifically, we treat a mod operation as a del operation and an add operation, and construct two directed edges $\{R_{u,v} \dashleftarrow O^{f_i}\}$ and $\{O^{f_i} \dashleftarrow R_{p,q}\}$ in the ODG (lines 9-13).

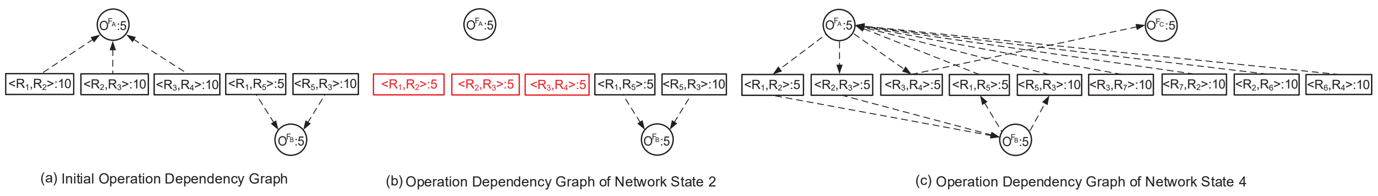


Fig. 3. The operation dependency graph of UE_1, NS_2, and UE_2 in Fig. 1.

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 $R_{u,v} \notin V_D$ then
3:         $V_D \leftarrow V_D \cup R_{u,v}$ with its current capacity $c_{u,v}$;
4:     $E_D \leftarrow E_D \cup \{R_{u,v} \dashleftarrow O^{f_i}\}$;
5: if $o^{f_i}$ is a del operation then
6:     if corresponding link node $R_{p,q} \notin V_D$ then
7:         $V_D \leftarrow V_D \cup R_{p,q}$ with its current capacity $c_{u,v}$;
8:     $E_D \leftarrow E_D \cup \{O^{f_i} \dashleftarrow R_{p,q}\}$;
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 $R_{p,q} \notin V_D$ then
12:        $V_D \leftarrow V_D \cup R_{u,v}$ with its current capacity $c_{u,v}$ or
           $V_D \leftarrow V_D \cup R_{p,q}$ with its current capacity $c_{p,q}$;
13:    $E_D \leftarrow E_D \cup \{R_{u,v} \dashleftarrow O^{f_i}\}$ and
       $E_D \leftarrow E_D \cup \{O^{f_i} \dashleftarrow R_{p,q}\}$;
```

By leveraging the ODG, the state of update operations can be identified. For the unexecuted operations, the ODG maintains the relationship between the operation node and the resource node. When update operations are executed, Coeus removes update operations from operation nodes O^{f_i} and corresponding edges between O^{f_i} and $R_{u,v}$ in the ODG. Then, Coeus updates the link resource of $R_{u,v}$. 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 NS_1 shown in Fig. 1(b), where the operation node O^{F_A} contains a set of uninstalled operations $\{o^{F_A}_{(R_1,add)}, o^{F_A}_{(R_2,add)}, o^{F_A}_{(R_3,add)}\}$ of F_A. In NS_2 shown in Fig. 1(c), the flow F_A has been routed on its target path, the corresponding operation node $O^{F_A} = \emptyset$, which is shown in Fig. 3(b). Also, corresponding directed edges are removed, and link resources are consumed. Similarly, Fig. 3(c) represents the ODG of UE_2, which means the resource request and release relationship among F_A, F_B, and F_C.

B. Operation Composition

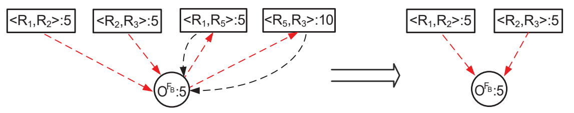
Based on the ODG, we present the operation composition to reduce the number of redundant operations in the continuous update. The basic idea is that once the update event occurs, we add new update operations into existing operation nodes and construct the new dependency relationship. By checking a cycle between operation nodes and resource nodes, Coeus judges whether unnecessary operations exist and replaces redundant operations with fewer equivalent ones.

Algorithm 3 gives a detailed operation composition process. We default all of the update operations are valid. We add each valid operation o^{f_i} into the corresponding operation node O^{f_i} and apply Algorithm 2 to update the dependency relationship (lines 1-3). Once the new update operation o^{f_i} incurs a cycle between O^{f_i} and $R_{u,v}$, it is denoted that update operations o^{f_i} and $\bar{o}^{f_i}$ will request and release the same link resource successively, where $\bar{o}^{f_i}$ is an update operation in the previous update events. We treat these operations as redundant operations. Therefore, we remove these two directed edges and corresponding update operations from the ODG (lines 4-21). Particularly, if the *mod* operation o^{f_i} forms a cycle, we split o^{f_i} as $o^{f_i}_{add}$ and $o^{f_i}_{del}$ and eliminate the redundant operations that are related to the removed edges (lines 14-20). 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 merge the residual operations into a new *mod* operation $\tilde{o}^{f_i}$ (line 21). If o^{f_i} does not incur a cycle while there exists another update operation $\hat{o}^{f_i}$ acting on the same switch, we merge these two operations into a new *mod* operation $\tilde{o}^{f_i}$ (line 23).

Algorithm 3: Operation Composition

```
Input: The update event with a set of new operations
Output: The composited dependency graph
1: for each $o^{f_i} \in UE$ do
2:     $O^{f_i} \leftarrow o^{f_i} \cup O^{f_i}$;
3:     Apply Algorithm 2 to update the dependency relationship;
4:     if there is a cycle between $O^{f_i}$ and $R_{u,v}$ then
5:         Remove edges $\{O^{f_i} \dashleftarrow R_{u,v}\}$ and $\{R_{u,v} \dashleftarrow O^{f_i}\}$;
6:         if $o^{f_i}$ is an add operation then
7:             $O^{f_i} = O^{f_i}/\{o^{f_i}, \bar{o}^{f_i}_{del}\}$, where $\bar{o}^{f_i}_{del}$ is the operation
               which is related to the edge $\{R_{u,v} \dashleftarrow O^{f_i}\}$;
8:         if $o^{f_i}$ is a del operation then
9:             if $\bar{o}^{f_i}$ is an add operation then
10:                $O^{f_i} = O^{f_i}/\{o^{f_i}, \bar{o}^{f_i}_{add}\}$, where $\bar{o}^{f_i}_{add}$ is the operation
                   that is related to the edge $\{O^{f_i} \dashleftarrow R_{u,v}\}$;
11:            else
12:                Divide $\bar{o}^{f_i}$ into $\bar{o}^{f_i}_{add}$ and $\bar{o}^{f_i}_{del}$;
13:                $O^{f_i} = O^{f_i}/\{o^{f_i}, \bar{o}^{f_i}_{add}\}$;
14:        if $o^{f_i}$ is a mod operation then
15:            Divide $o^{f_i}$ into $o^{f_i}_{add}$ and $o^{f_i}_{del}$;
16:            if $\bar{o}^{f_i}$ is an add or a del operation then
17:                $O^{f_i} = O^{f_i}/\{o^{f_i}_{del}, \bar{o}^{f_i}_{add}\}$ or $O^{f_i} = O^{f_i}/\{o^{f_i}_{add}, \bar{o}^{f_i}_{del}\}$;
18:            else
19:                Divide $\bar{o}^{f_i}$ into $\bar{o}^{f_i}_{add}$ and $\bar{o}^{f_i}_{del}$;
20:                Remove the divided operations of $o^{f_i}$ and $\bar{o}^{f_i}$
                   which related to the removed edges;
21:                Merge the residual operations of $o^{f_i}$ and $\bar{o}^{f_i}$
                   as a new mod operation $\tilde{o}^{f_i}$;
22:    else
23:        Merge $o^{f_i}$ and $\hat{o}^{f_i}$ as a new operation $\tilde{o}^{f_i}$, where $\hat{o}^{f_i}$ is
           an operation for $f_i$ that acting on the same switch;
```

Fig. 4 is an example to illustrate the operation composition. We assume that the update event UE_2 (shown in Fig. 1(e)) oc-



(a) Operation Composition of *NS*$_2$ and *UE*$_2$ (b) Composited Operation

Fig. 4. Illustration of update composition

TABLE III
CORRECT OPERATIONS OF SINGLE UPDATE

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

TABLE IV
COMPOSITION OF UE_m AND UE_n

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

curs after the network state NS_2 (shown in Fig. 1(c)). Fig. 4(a) shows the dependency relationship among the operation node O^{F_B} and corresponding link resources under the situation mentioned above. After operation composition, the simplified dependency relationship is shown in Fig. 4(b).

Definition V.2. Correct Update Operations: *The correct update operations mean that if all update operations are executed, flows are routed on their target paths, 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. The update event UE_n can be responded when all of the flows in UE_m have been routed on their target paths. Since UE_m, UE_n are valid update events, executing UE_m, UE_n serially makes switches have no redundant rules. Therefore, the correct update operations can always be guaranteed in the single update. Table III represents the correct update operations in the single update, where blanks denote invalid operations. However, the single update causes lots of unnecessary operations. In Coeus, continuous update events are handled by compositing the unexecuted operations in UE_m with the operations in UE_n. Now, we prove after operation composition, the update operations are still correct.

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

Proof: Table IV shows the result of operation composition under the assumption that operations of UE_m have not been executed. We prove that the final route 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, which means doing nothing in the specified 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 route of flow, then an *add* operation in UE_n adds the new route which is the same as the original route in the single update. Another possible composited result is producing a *mod* operation $\tilde{o}^{f_i}_{mod}$. The new operation $\tilde{o}^{f_i}_{mod}$ will delete the original route and add a new route for f_i. The flow f_i will route along the new path. This composited result corresponds to the condition where a *del* operation in UE_m deletes the original route of flow and an *add* operation in UE_n adds the new route which is different from the original route in the single update. Therefore, executing the new operation produced by compositing $o^{f_i}_{add}$ in UE_n and $o^{f_i}_{del}$ in UE_m is equivalent to executing corresponding update operations serially in the single update. Similarly, once operations $\tilde{o}^{f_i}_{del}$, $\tilde{o}^{f_i}_{add}$, $\tilde{o}^{f_i}_{mod}$ produced by operation composition are executed, we obtain the same routes compared with the single update. We omit the proof in detail due to space constraints. ■

C. Operation Node Partition

To improve the parallelism of update, we divide the operation node into multiple suboperation nodes. We aim to generate independent suboperation nodes as many as possible while maintaining the consistency. Our algorithm is inspired by the partition technology used in [23], [24], while they either cause excessive dependencies among each partition [23] or cannot be applied to the situation where the network has potential loops [24]. In Coeus, we develop a novel partition algorithm. We prove that our algorithm is optimal to obtain independent suboperation nodes and each suboperation node can be updated in a blackhole- and loop-free manner. Now, we define suboperation node and independent suboperation node.

Definition V.3. Suboperation Node: *The suboperation node $O^{f_i}_j$ owns a part of update operations of the operation node O^{f_i}. Specifically, the suboperation node contains at least one mod operation to delete the original forwarding rule and add the new forwarding rule. Besides, each suboperation node has different update operations compared with other suboperations, i.e.,* $O^{f_i}_j \cup O^{f_i}_{j+1} \cup \cdots = O^{f_i}$, $O^{f_i}_j \cap O^{f_i}_{j+1} \cap \cdots = \emptyset$.

In the following, we describe the process of operation node partition in Algorithm 4. $O^{f_i}_j$ is the j-th suboperation node of O^{f_i}. When the current operation is $o^{f_i}_{mod}$, we define next *mod* operations along the target path and the original path as $o^{f_i}_{(mod,nt)}$ and $o^{f_i}_{(mod,no)}$ respectively. Firstly, we traverse update operations of O^{f_i} along the target path of f_i in reverse order (line 3). If the operation node O^{f_i} has a *mod* operation $o^{f_i}_{mod}$, we initialize a suboperation node $O^{f_i}_j$ (lines 4, 5). Then we add $o^{f_i}_{mod}$ into $\varphi^{f_i}_o$, $\varphi^{f_i}_n$ and remove $o^{f_i}_{mod}$ from O^{f_i} (lines 6, 7), where $\varphi^{f_i}_o$ and $\varphi^{f_i}_n$ are sets of update operations acting on the original path and the target path. Since installing *add* operations makes f_i routes through its target path, we add a set of *add* operations $\{o^{f_i}_{add}\}$ between $o^{f_i}_{mod}$ and $o^{f_i}_{(mod,nt)}$ into $\varphi^{f_i}_n$ (line 8). If the original path deleted by $o^{f_i}_{mod}$ is involved in a loop, the next suboperation node will depend on this suboperation node, *i.e.*, the next suboperation node cannot be updated directly until this suboperation node is completed. Otherwise, a forwarding loop will occur. Next, we check whether the original path deleted by $o^{f_i}_{(mod,no)}$ is

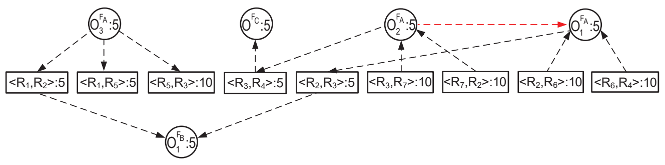


Fig. 5. Dependency graph with suboperation nodes.

involved in a loop. If this condition holds, we split $o^{f_i}_{(mod,no)}$ into $o^{f_i}_{(add,sp)}$ and $o^{f_i}_{(del,sp)}$ and search the next *mod* operation $o^{f_i}_{(mod,no)}$ along the original route of f_i until $o^{f_i}_{(mod,no)}$ does not incur a loop (lines 9-12). Finally, we add a set of *del* operation $\{o^{f_i}_{del}\}$ between $o^{f_i}_{mod}$ and $o^{f_i}_{(mod,no)}$ into $\varphi^{f_i}_{o}$ (line 13) and construct the next suboperation node (lines 15, 16). We eliminate potential loops by this way since we add update operations which may incur loops into two suboperation nodes.

Algorithm 4: Operation Node Partition

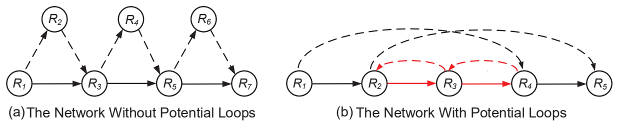
```
Input: The current dependency graph
Output: The dependency graph with suboperation nodes
 1: for each operation node $O^{f_i}$ in dependency graph do
 2:     $j = 0$;
 3:     Traverse operations along the target path of $f_i$ in reverse
        order;
 4:     while $o^{f_i}_{mod} \neq \emptyset$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

By executing Algorithm 3 and Algorithm 4, the dependency relationship among suboperation nodes is shown in Fig. 5. $O_1^{F_A}$, $O_3^{F_A}$, $O_1^{F_B}$, $O_2^{F_B}$, and $O_1^{F_B}$ are suboperation nodes which can be executed independently, while $O_2^{F_A}$ is a dependent node which can be executed until $O_1^{F_A}$ is completed. Now, we define the independent suboperation node formally.

Definition V.4. Independent Suboperation Node: *The independent suboperation node is the suboperation node that can execute update operations independently.*

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

Proof: If there is no potential loop shown in Fig.6(a), each *mod* operation will not incur loops. Algorithm 4 adds



R1
R2
R3
R4
R5
R6
R7
(a)The Network Without Potential Loops
R1
R2
R3
R4
R5
(b) The Network With Potential Loops

Fig. 6. Illustration of independent suboperation nodes.

each of *mod* operation into different suboperation nodes, *i.e.*, the number of *mod* operations equals to the number of independent suboperation nodes. According to Definition V.4, Coeus produces the most number of independent suboperation nodes. If forwarding loops occur shown in Fig. 6(b), we assume that the optimal algorithm can generate an extra independent suboperation node $O^{f_i}_{e}$. If the original path which $O^{f_i}_{e}$ will be deleted is not in a potential loop, *e.g.*, the link $\langle R_1, R_2\rangle$ or $\langle R_4, R_5\rangle$, this situation is similar to the loop-free network shown in Fig. 6(a). $O^{f_i}_{e}$ must be one of the independent suboperation nodes produced by Algorithm 4, which contradicts with our assumption. Otherwise, the original path which will be deleted by $O^{f_i}_{e}$ is involved in loops, *e.g.*, $\langle R_2, R_3\rangle$ or $\langle R_3, R_4\rangle$. According to Definitions V.3, $O^{f_i}_{e}$ must has a *mod* operation. And a *mod* operation in $O^{f_i}_{e}$ acting on the target link must be involved in a loop, *e.g.*, $o_{(R_3,mod)}$. Otherwise, $O^{f_i}_{e}$ is one of the independent suboperation nodes that we produced. $O^{f_i}_{e}$ cannot be an independent node under this situation because to guarantee loop-free condition, the forwarding rule that added on the target path by a *mod* operation in $O^{f_i}_{e}$ must be executed after deleting the corresponding forwarding rule of the original path. ■

After suboperation nodes are determined, we update each of them in the following way. Firstly, we install all *add* operations of the subsection node simultaneously and remove the corresponding edges and operations in the ODG. Then we install the *mod* operation, which makes the flow route through its target path. The residual link resources along the original path and target path are updated. Finally, we install the *del* operations to delete the forwarding rules acting on the original path of the flow.

Theorem V.3. *The update of suboperation nodes is blackhole- and loop-free.*

Proof: Firstly, the update of each suboperation node requires to install the forwarding rules of the target path. Then the *mod* operation and *del* operations are executed, which ensures that packets always have the forwarding rule. Besides, each suboperation node contains the operations that will delete current forwarding rules and add target forwarding rules. Packets among suboperation nodes will not be dropped and the update is blackhole-free. In Algorithm 4, the suboperation node only contains one *mod* operation which is not going to form a loop. It is because that we always assign update operations incurring a potential loop to two suboperation nodes. We restrict the update order of these two suboperation nodes to ensure the loop-free condition. ■

D. Update Scheduling

After dividing the operation node into several independent suboperation nodes, the parallelism of the ODG has been improved. Then according to the ODG, a set of update commands are sent from the controller orderly. We describe the update scheduling in Algorithm 5.

```
Algorithm 5: Operation Dependency Graph Update
Input: The current dependency graph
Output: A set of update commands
 1: $\theta = \tilde{\theta} = \emptyset$
 2: for each independent operation node $O_j^{f_i}$ do
 3:     if there are no add and mod operations then
 4:         $\tilde{\theta} = \tilde{\theta} \cup O_j^{f_i}$;
 5:     else
 6:         if all link resources are sufficient for $O_j^{f_i}$ then
 7:             $\theta = \theta \cup O_j^{f_i}$;
 8: Rank $O_j^{f_i}$ in $\theta$ in descending order according to their
    out-degree;
 9: for each $O_j^{f_i}$ in $\theta$ do
10:     if all link resources are sufficient for $\tilde{\theta}+O_j^{f_i}$ then
11:         $\tilde{\theta} = \tilde{\theta} \cup O_j^{f_i}$;
12: Update each operation node in $\tilde{\theta}$;
13: if a deadlock occurs then
14:     for each $O_j^{f_i}$ in a deadlock do
15:         Calculate throughput loss ratio $\Phi = \frac{d^{f_i} - _{min} r_{u,u}}{d^{f_i}}$;
16:     Choose $O_j^{f_i}$ with minimal $\Phi$ and limit its rate;
17:     Update $O_j^{f_i}$ with minimal $\Phi$;
```

We use θ and $\tilde{\theta}$ to present the set of candidate update nodes and formal update nodes. Initially, we put the operation node $O_j^{f_i}$ which only contains *del* operations into $\tilde{\theta}$. Coeus can update such a $O_j^{f_i}$ directly since $O_j^{f_i}$ just releases 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 the same time, we rank $O_j^{f_i} \in \theta$ in descending order of out-degree and add the congestion-free operations into $\tilde{\theta}$ (lines 8-11). Selecting the operation nodes with high out-degree and updating them simultaneously will release more link resources (line 8). Once deadlock occurs in the ODG, *i.e.*, flows occupy insufficient resources of links mutually, the throughput loss ratio Φ is calculated for each operation node in deadlock. Then we select the operation node with minimal Φ and update it directly by limiting the flow rate to $d^{f_i}(1-\Phi)$ (lines 13-17). Once a set of update commands are sent, Coeus checks whether the new update event arises and decides the next update step.

Theorem V.4. *Coeus always produces a blackhole-free, loop-free, and congestion-free update sequence.*

Proof: According to TheoremV.3, suboperation nodes can be updated in a blackhole-free and loop-free manner. Besides, Algorithm 5 always updates operation nodes with sufficient link resources, which makes the network congestion-free. If deadlocks occur, we limit the rate of flow to fit the residual link resource. Therefore, updating the flow involved in the deadlock will never congest the link. ∎

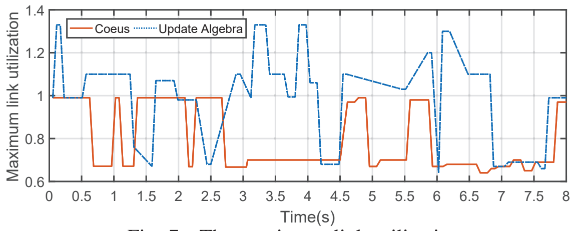


Fig. 7. The maximum link utilization.

VI. Experimental Evaluation

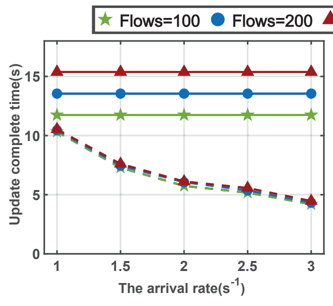
In this section, we design large-scale simulation experiments to verify the performance of Coeus.

Methodology: We evaluate Coeus with two common topologies: Microsoft's WAN topology [15] with 8 switches (*i.e.*, SWAN) and 8-pods fat-tree [3] with 80 switches and 128 hosts. We set the link capacity of each topology to be 1-Gbps and then generate different numbers of updated flows (*i.e.*, 100, 200, 400, 600) with random source and destination in the network. For each updated flow, we generate continuous update events with different arrival rate λ. Besides, according to the test of commodity switches [15], we set the time of insertion, deletion, and modification operations to be $5ms, 5ms, 10ms$ respectively. The RTT between the controller and switches is set to be $50ms$. We compare the performance of Coeus with "Update Algebra" [11] which only ensures blackhole- and loop-free conditions and Cupid [23] which processes the single update with congestion-free condition guaranteed. The program executes 10 times for each data set.

Experiment results: At first, we investigate the maximum link utilization of Coeus and "Update Algebra" in Fig. 7. We do this simulation with 100 flows in SWAN topology and the arrival rate of update events $\lambda = 3/s$. Once the maximum link utilization is beyond one, the link congestion will happen. Fig. 7 shows that Coeus always guarantees that the maximum link utilization is less than or equal to one, which ensures congestion-free condition. In contrast, sometimes the maximum link utilization of "Update Algebra" is over 1.3, which incurs packets loss and degrades network performance.

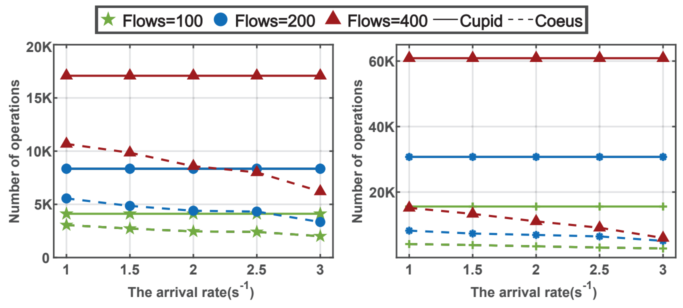
Next, we generate 10 continuous update events and compare Coeus against Cupid in multiple dimensions. Fig. 8 shows the completion time of all update events. We observe that the update time of Coeus is shorter than that of Cupid. Moreover, the gap between the update time in Coeus and Cupid is getting larger with the arrival rate increasing. Specifically, in SWAN with 100 flows, Coeus shortens the makespan by 13.2%, 61.4%, 120.7%, 130.1%, 179.8% when λ =1 $\sim$ 3 respectively. The reason is that Cupid executes update events serially while Coeus enables the parallel execution. We also observe that the update time of Coeus in SWAN and fat-tree are highly similar. One reason is that lots of flows with sufficient resources can be updated simultaneously. Another reason is that Coeus only determines the update order of flows which may incur potential link congestion, while the number of potential congested flows in SWAN is close to that in fat-tree.

Fig. 9 shows the number of executed update operations after finishing 10 continuous update events. The number of update



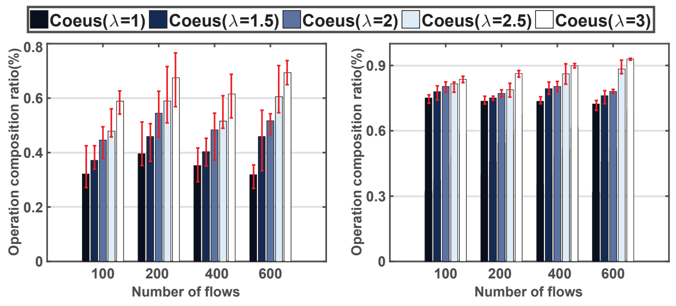
(a) WAN scenario (b) Fat-tree scenario

Fig. 8. Update complete time.



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

Fig. 9. Number of operations.



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

Fig. 10. Operation composition ratio.

operations of Cupid is constant with arrival rate varying since Cupid executes all update operations for each update event. In contrast, Coeus executes fewer update operations. This benefits from the operations composition which eliminates redundant update operations. Furthermore, with the arrival ratio increasing, Coeus owns more unexecuted operations which can be composited with operations in new update events. This further reduces the number of update operations. Specifically, when the arrival rate $\lambda = 3$, Coeus reduces 51.2%, 59.9%, 64.2% update operations compared with Cupid in SWAN with 100, 200, 400 updated flows. We also observe that operations in fat-tree are much more than operations in SWAN since flows in fat-tree are routed through more links. In fat-tree topology with 100, 200, 400 updated flows, when the arrival rate $\lambda = 3$, Coeus reduces 82.2%, 83.4%, 90.1% operations respectively compared with Cupid since many operations are redundant in fat-tree.

Fig. 10 reflects the operation composition ratio of each update event. In SWAN, Coeus composites at least 30% redundant operations of each update event. While in fat-tree, the composition ratio is pretty high since fat-tree involves more update operations. Specifically, at least 70% operations can be composited by Coeus in each update event. This demonstrates that Coeus can execute fewer update operations while maintaining consistency in continuous updates.

VII. Related Work

With the advance of SDNs, the update problem has been widely studied. Reitblatt *et al.* [22] introduced the notion of consistent update in SDNs and proposed the two-phase update protocols to maintain per-packet coherence. To ensure the connectivity consistency, Ludwig *et al* [8], [17] achieved the fast blackhole- and loop-free update by using node-based protocols. To guarantee the congestion-free and fast update, zUpdate [16] and SWAN [12] utilized the slack capacity of each link to produce the static congestion-free update sequence. On these bases, Xin *et al* [15] and Gandhi *et al.* [10] performed update scheduling dynamically by utilizing the global resource dependency graph. Wang *et al.* [23] and Wu *et al.* [24] divided the global dependency relationship into the local dependency relationship by dividing flows into segments and updated each segment in a congestion-free manner. By taking advantage of time synchronization protocols [19]–[21], Zheng *et al.* [28], [30] 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. Nevertheless, all of the update solutions mentioned above mainly focus on how to improve the update order of the single update. Such an update manner leads to the serial execution of continuous update events, which slows down the makespan significantly. The continuous update method is firstly proposed in [11]. The authors built the theoretical framework based on abstract algebra and generated the blackhole- and loop-free update order. However, the update order in [11] may cause transient congestion which leads to packets loss and network performance degradation. To the best of our knowledge, Coeus is the first work to handle the continuous update with blackhole-, loop- and congestion-free properties guaranteed simultaneously.

VIII. Conclusion

We studied the consistency and continuous update problem in SDNs. We proposed Coeus to obtain a consistent update sequence during the continuous update procedure. The results of evaluation show that Coeus can reduce the makespan and redundant update operations significantly.

Acknowledgment

This work was supported in part by the National Key R&D Program of China under Grant No. 2017YFB1001801, the National Natural Science Foundation of China under Grant No. 61802172, 61672276, 61872178, 61832005, 61672353, the Natural Science Foundation of Jiangsu Province under Grant No. BK20181251, and Royal Society project IEC170324. The corresponding authors are Jiaqi Zheng and Wanchun Dou.

REFERENCES

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

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

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM*, pages 63–74, 2008.

[4] A. Basta, A. Blenk, S. Dudycz, A. Ludwig, and S. Schmid. Efficient loop-free rerouting of multiple sdn flows. *IEEE/ACM Transactions on Networking*, 26(2):948–961, 2018.

[5] T. Benson, A. Anand, A. Akella, and M. Zhang. Microte: Fine grained traffic engineering for data centers. In *ACM CoNEXT*, page 8, 2011.

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

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

[8] K. T. Foerster, A. Ludwig, J. Marcinkowski, and S. Schmid. Loop-free route updates for software-defined networks. *IEEE/ACM Transactions on Networking*, PP(99):328–341, 2018.

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

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

[11] F. L. Y.-s. L. Geng Li, Y. Richard Yang and J. Wang. Update algebra: Toward continuous, non-blocking composition of network updates in sdn. In *IEEE INFOCOM*, pages 1081–1089, 2019.

[12] 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*, pages 15–26, 2013.

[13] 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*, pages 3–14, 2013.

[14] 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*, pages 2876–2880, 2012.

[15] 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*, pages 539–550, 2014.

[16] 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*, pages 411–422, 2013.

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

[18] R. Mahajan and R. Wattenhofer. On consistent updates in software defined networks. In *ACM HotNets*, page 20, 2013.

[19] T. Mizrahi and Y. Moses. Software defined networks: It's about time. In *IEEE INFOCOM*, pages 1–9, 2016.

[20] T. Mizrahi, O. Rottenstreich, and Y. Moses. Timeflip: Scheduling network updates with timestamp-based tcam ranges. In *IEEE INFOCOM*, pages 2551–2559, 2015.

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

[22] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *ACM SIGCOMM*, pages 323–334, 2012.

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

[24] 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*, 36(3):411–421, 2018.

[25] 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*, 25(5):3073–3087, 2017.

[26] 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*, pages 1–10, 2016.

[27] 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*, pages 1871–1879, 2018.

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

[29] 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*, pages 1–10, 2016.

[30] 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*, pages 12–21, 2018.

[31] 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*, pages 231–240, 2018.

[32] J. Zheng, H. Xu, G. Chen, and H. Dai. Minimizing transient congestion during network update in data centers. In *IEEE ICNP*, pages 1–10, 2015.

[33] J. Zheng, H. Xu, X. Zhu, G. Chen, and Y. Geng. Sentinel: Failure recovery in centralized traffic engineering. *IEEE/ACM Transactions on Networking*, 27(5):1859–1872, 2019.

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