Minimizing Update Makespan in SDNs without TCAM Overhead

Mahdi Dolati, Ahmad Khonsari, *Member, IEEE* and Majid Ghaderi, *Member, IEEE*

***Abstract*—Efficient and consistent update of the network routing rules is a challenging task that significantly affects the performance, correctness, and security of Software-Defined Networks (SDN). In this work, we consider the problem of minimizing the makespan of updating the routing rules in SDNs, while guaranteeing three crucial consistency requirements: (1) WayPoint Enforcement, (2) Loop Freedom, and (3) Conflict Freedom. This problem is known to be NP-hard, and thus we focus on designing approximate algorithms that run in polynomial time without incurring TCAM storage overhead. To compute consistent rule-update schedules, we propose two algorithms, called TimeX and RMS. TimeX employs the solution of a linear program (LP) to address the makespan minimization goal systematically. RMS is an LP-independent heuristic that provides higher scalability. We demonstrate and utilize a property of rule-updates, called reversibility, to reduce the makespan in RMS. Extensive simulations show that our algorithms reduce the makespan by 2% to 18% and attain a 4.9x speedup compared to previous studies. Moreover, Mininet experiments reveal that the proposed algorithms can mitigate the transient congestion caused by conflicting flows.**

***Index Terms*—SDN, Consistent Rule-Update, WayPoint-Enforcement, Loop-Freedom, Conflict-Freedom**

I. Introduction

A. Background and Motivation

SOFTWARE-Defined Networking decouples the network control logic from the underlying switches that forward traffic. Then, a logically centralized *controller* implements the control plane by installing and *updating* forwarding rules on network switches to instruct them on how to forward traffic in the network. The SDN controller can increase resource utilization and network efficiency by reacting to dynamic changes in the network by quickly updating the forwarding rules [1]–[4]. Furthermore, achieving goals such as failure recovery in a timely manner requires a fast update of the forwarding rules [4]–[6]. Thus, efficient mechanisms for quickly updating the forwarding rules are crucial for the operation of future networks [7].

Updating forwarding rules in SDNs, however, is a challenging task. The main challenge is that the delay of sending new rules to switches and the required time to install the rules in the switch memories are non-deterministic. Therefore, even if the controller sends out the new rules simultaneously, some switches may update their forwarding behavior considerably sooner than others [8]. Studies show that the gap between sending the update messages and when the data plane is updated can be several milliseconds [9], which is a significant delay in high-capacity networks. This asynchronous behavior, which causes partial execution of rule-update messages, may lead to network state inconsistencies [10]. This inconsistency is observable in terms of sending packets in transient loops [11], overloading link capacities [3], or bypassing important waypoints such as a flow classifier [12]. Network properties that should endure throughout the rule-update process are called the network *consistency properties*.

Some works store the new forwarding rules in the switch TCAM before removing the old forwarding rules and use tagging to implement an update procedure that guarantees the consistency properties [11]. This extra TCAM usage can restrict network operations, such as fine-grained flow management [13] that extensively rely on TCAM. Also, when a large-scale rule update becomes necessary due to a security breach or a device outage [10], the TCAM overhead can be prohibitive. Performing the rule update in a sequence of smaller steps can alleviate the TCAM consumption problem. This approach, however, results in longer latencies until the network converges to its new state and can not address situations when some switches have already exhausted their TCAM capacity. Furthermore, these approaches typically store the tags in the VLAN field that may conflict with other applications that use it [14].

Later studies showed that it is possible to preserve the consistency properties by coordinating rule updates across the switches to address the TCAM usage issue [15]. This method partitions the rule updates into multiple *time slots*, where the order of updates in a single time slot does not violate the consistency properties. The number of required time slots for delivering all the rule updates is called the update makespan. Update makespan determines the time a network spends in a sub-optimal configuration [8], which considerably affects the network performance. Since minimizing the makespan is a known NP-hard problem [16], in this paper, we focus on designing fast algorithms for this problem.

B. Our Work

In this work, we consider consistent multi-flow rule-update with minimum makespan in SDNs. We assume that the SDN is under a single administrative domain and controlled by a virtually centralized controller. We design our approaches

Mahdi Dolati is with the Department of Electrical and Computer Engineering, University of Tehran, Iran. E-mail: mahdidolati@ut.ac.ir

Ahmad Khonsari is with the Department of Electrical and Computer Engineering, University of Tehran, and Institute for Research in Fundamental Sciences (IPM), Iran. E-mail: a_khonsari@ut.ac.ir

Majid Ghaderi is with the Department of Computer Science, University of Calgary, Canada; Email: mghaderi@ucalgary.ca

based on an iterative scheduling model to save TCAM memory. We consider three consistency properties: (1) WayPoint Enforcement (WPE) that mandates all packets of each flow in the network visit a particular node, such as a firewall. (2) Loop Freedom (LF) that prevents transient routing loops. (3) Conflict Freedom (CF) that ensures conflicting flows (*e.g.*, correlated bursty flows) do not use the same link at the same time (see [17]). To the best of our knowledge, no schedule-based polynomial-time algorithm exists for minimizing the makespan while considering these three consistency properties. For example, the authors in [16], [18] restrict themselves to the single-flow setting, works in [19], [20] ignore the update makespan, and works in [3], [21] neglect routing restrictions that enforce traversing through specific nodes. We propose two algorithms, named TimeX and RMS, to address this research gap. Our contributions can be summarized as follows:

- We formulate the rule-update makespan minimization problem in an SDN while preserving the Loop-Freedom, Conflict-Freedom, and WayPoint Enforcement properties.
- To advance the theoretical study of the makespan minimization problem, we present the design of an integer program relaxation-based algorithm, called TimeX, and investigate its performance in practice.
- We propose a more scalable algorithm, called RMS, that does not rely on linear program solutions. We prove a property called reversibility, which allows obtaining two potentially different solutions for each problem instance and then merging the solutions to reduce the update makespan.
- Finally, we thoroughly evaluate the performance of the proposed algorithms with simulation and Mininet experiments.

C. Paper Organization

We review related works in Section II. System model and problem definition are presented in Sections III and IV, respectively. In Section V, we design an LP-based solution called TimeX. In Section VI, we avoid using the LP solution to create a scalable algorithm called RMS. Section VII presents the evaluation results. Section VIII concludes the paper.

II. Related Work

We categorize previous studies into TCAM-based, schedule-based, and multi-flow methods to highlight our work's distinctions. Each category discusses several seminal and recent studies. For a comprehensive list of papers on the consistent update, please refer to [22]. Table I presents a concise comparison of our proposed methods with the existing ones.

TCAM-based Methods. Authors in [11] considered PerPacket Consistency (PPC) property and proposed a method named the 2-phase commit based on the packet and rule tagging to guarantee PPC. Although their solution minimizes the update makespan by imposing only two rounds of message passing between the controller and switches, it doubles the TCAM memory usage [23]. Authors in [24] considered the Per-Flow Consistency (PFC) model, which is more imposing than PPC, and proposed concrete mechanisms for PPC and PFC. Authors in [25] considered the Per-Bidirectional-Flow Consistency (PBFC) property that guarantees PFC for both

TABLE I: Relevant Related Works.

Literature	no-TCAM	Makespan	Multi-flow	Consistency	Complexity
[11]	✗	✓	✗	PPC	Poly
[26]	✗	✓	✗	Path-based	Exp
[15]	✓	✓	✗	LF	Poly
[18]	✓	✓	✗	WPE+LF	Exp
[16]	✓	✓	✗	Multi-WPE+LF	Exp
This Paper					
TimeX	✓	✓	✓	WPE+LF+CF	Poly
RMS	✓	Heuristically	✓	WPE+LF+CF	Poly

forward and backward directions. Authors in [23] proposed an incremental update mechanism to reduce the TCAM overhead when forwarding rules are not exact matches. Their algorithm uses a pre-determined set of predicates to update a subset of flows with the 2-phase commit. They proposed an integer program to select predicates that minimize the rule-update makespan subject to a constraint on the TCAM overhead. These works rely on using extra TCAM entries, which reduces the switching capacity and increases energy consumption.

Schedule-based Methods. To avoid redundant TCAM entries, authors in [15] considered scheduling the rule updates in multiple *time slots*. They discussed a general *framework* for SDN updates. However, they do not provide mechanisms for ensuring policy-related properties like WPE. Authors in [27] proved that approximating the *minimum* makespan with a ratio better than $\frac{4}{3}$ is NP-hard, even for the simple consistency property of loop-freedom. [18] and [16] present exact Mixed-Integer Program (MIP) formulations for loop-freedom and waypoint enforcement consistency properties. Authors in [26] combined the scheduling and packet tagging methods. However, their approach relies on solving integer linear programs that are computationally expensive. Authors in [20], instead of minimizing the makespan, considered maximizing the number of updated switches in each time slot in topologies with bounded tree-width. However, this approach can increase the update makespan by a factor of n, compared to the optimal makespan (where n is the maximum path length of the affected traffic flows in the network [28]). The work presented in [29] assumes that update rules are bounded and pre-computed and chooses the configuration that minimizes the makespan with a *max-cover*-based algorithm. The effectiveness of this approach is limited to the availability of these pre-computed configurations. These works ignore flow inter-dependencies and focus on the single flow update problem.

Multi-flow Methods. The majority of existing multi-flow methods address congestion freedom and packet delivery during the network update [3], [21], [27], [30]–[33]. However, these works neglect higher-level policies that are typically crucial for network operators. For example, some flows can only use specific paths for security or quality of the service reasons. Authors in [34] defined an abstraction at the event level to aggregate related traffic flows. Then, they minimized the bandwidth consumed for the migration of flows that block a specific update event. The challenges of handling wireless interference while updating a software-defined wireless network are considered in [35], in which the authors propose a greedy-heuristic scheme to minimize the update makespan. Authors in [9] used a priority-based sorting mechanism that exploits the dependencies among the forwarding rules to reduce the flow update delay. Most of these methods do not consider high-level network policies or do not consider the makespan minimization objective.

A preliminary version of this work appeared in [36], where we presented a heuristic algorithm to minimize the update makespan in the single-flow setting.

III. System Model

In this section, we model the consistent update of paths in an SDN based on message scheduling. We specify network and traffic models in subsections III-A and III-B, respectively. Then, we discuss the set of properties used to model the network consistency in subsection III-C. Finally, we explain the controller's approach for preserving the consistency properties in subsection III-D. Table II lists notations of this section.

A. Network Model

We consider an SDN with a centralized controller and a set of OpenFlow-enabled switches. The controller maintains a connection to each switch for installing forwarding rules in their memories, where each forwarding rule is composed of a match on the packet header fields and an action. Switches forward the traffic by applying the action corresponding to the best-matching forwarding rule. Meanwhile, the controller monitors the network and updates the rules to keep the performance at the desired level. The controller can update a forwarding rule and change its action by sending a specific message defined in the OpenFlow protocol (*i.e.*, OFPFC_MODIFY) to the corresponding switch. We assume that rule updates do not disrupt the connections between the controller and switches. This assumption is valid when those connections use dedicated paths (*i.e.*, out-of-band control). Moreover, when those connections use the same infrastructure as data flows (*i.e.*, in-band control), we assume that control connections use isolated forwarding rules that do not overlap with the forwarding rules of traffic flows. For example, Open vSwitch implements the in-band control with rules not visible through the OpenFlow protocol [37].

B. Traffic Model

We define a flow as a sequence of packets that use the same forwarding rules along their paths. For splittable flows, the controller treats each split as a distinct flow. We assume that the controller selects a subset of flows to update their routing path and stores them in the set $\mathcal{F}$. Each flow $f \in \mathcal{F}$ carries traffic from its source s_f to its destination d_f. We use $\mathcal{L}_f^1$ and $\mathcal{L}_f^2$ to show the set of links in the paths of flow f before and after the update, respectively. We refer to the routing paths before and after the update as *initial* and *final* paths, respectively. Since a switch that is only in the initial path does not undergo a routing update, we can remove it from the model and connect its neighbors directly via a link. Furthermore, a switch that is only in the final path does not initially forward packets of the corresponding flow, so we can update its routing state before other switches and eliminate it similar to switches that are only in the initial path. Consider

TABLE II: Input Notations.

Sym.	Description	Sym.	Description
$\mathcal{F}$	Set of flows to be updated	s_f	Source of flow f
$\mathcal{W}_f$	Switches in the path of f	d_f	Destination of flow f
$\mathcal{L}_f^1$	Links in the initial path of f	Ω_f	WayPoint of flow f
$\mathcal{L}_f^2$	Links in the final path of f	$\ell.h$	Head of the link ℓ
$\mathcal{C}$	Partition of flows to compute makespan	$\ell.t$	Tail of the link ℓ
$\overline{\mathcal{L}}_f$	Links in $\mathcal{L}_f$, not connected to Ω_f	$\mathcal{L}_f$	$\mathcal{L}_f^1 \cup \mathcal{L}_f^2$
$P_f(w,t)$	Path of flow f from switch w in t	$P_f(t)$	$P_f(s_f,t)$
$M(i)$	Makespan for flows in $\mathcal{C}_i$	K	Size of $\mathcal{C}$
$X(Q,\ell)$	Conflict indicator function		

switches w_8 and w_9 in Fig. 1 that are only in the initial and final paths, respectively, and their removal in Fig. 2. Therefore, we assume that the initial and final paths use the same switches but in a different order and use $\mathcal{W}_f$ to show the set of common switches between both paths. We use $\ell.h$ and $\ell.t$ to show the head and tail of each link $\ell \in \mathcal{L}_f^1$ (or $\ell \in \mathcal{L}_f^2$), which means packets of f over ℓ go from switch $\ell.h$ to switch $\ell.t$. We allow each flow to specify a switch that its packets should traverse before reaching the destination. Henceforth, we call this distinguished switch a WayPoint, and for each flow f, represent it by Ω_f. The WayPoint can model the existence of mission-critical functions in the network (*e.g.*, a NAT). If a flow does not have a WayPoint, we let $\Omega_f = d_f$, which eliminates the effect of the WayPoint, as all packets always traverse their destination switch. For convenience, we define $\mathcal{L}_f$ and $\overline{\mathcal{L}}_f$ to show the links in $\mathcal{L}_f^1 \cup \mathcal{L}_f^2$ and links not connected to Ω_f, respectively. We consider conflict among flows, where some flows should not share specific links with other designated flows even for a short period. For example, bursty flows that can momentarily fill a switch queue and create transient congestion are conflicting. SDN applications can use techniques such as the one in [17] to detect these flows. To represent the conflicts, we define the function $X(Q,\ell)$, which is equal to 1 if flows in $Q \subseteq \mathcal{F}$ conflict with each other over link ℓ and is equal to 0 otherwise.

C. Consistency Model

Previous studies showed that even when the controller dispatches all rule-update messages together, network switches do not change their routing behavior simultaneously [38]. This inherent uncertainty about the exact moment of change can lead to a transient inconsistent routing state that is not equal to the initial or the final routing states. We consider three *consistency properties* to guarantee during update periods regardless of existing uncertainty. To demonstrate these properties, we investigate different possible outcomes for the shown flow $f \in \mathcal{F}$ in Fig. 2 when the controller sends all rule-update messages without further planning. We draw the initial and final paths of the flow with solid and dashed black lines, respectively. Switch w_2 is the WayPoint of f. Also, we consider a conflicting flow $f' \in \mathcal{F}$ and represent it by a bold red line. For simplicity, we omit the complete list of links and switches that f' traverses. We only assume that the initial path of f' uses the link between switches w_2 and w_6. The considered consistency properties are listed as follows:

Loop-Freedom (LF). This property indicates that packets should not enter a loop at line-rate speed. Consider Fig. 2 and

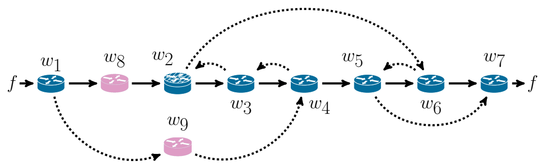


Fig. 1: Example of a scenario with switches only in the initial or final path of the flow (see w_8 and w_9).

assume that switch w_4 updates its rule before other switches and starts using the dashed outgoing link instead of the solid outgoing link. Consequently, w_4 forwards the incoming packets from w_3 back to w_3 and creates a routing loop, where packets will drop quickly since their TTL expire.

WayPoint-Enforcement (WPE). This property indicates that all the packets in a flow should traverse the specified WayPoint before reaching the destination. Assume that switch w_1 in Fig. 2 updates its rule before other switches and starts forwarding the packets to switch w_4. Forwarded packets will follow the initial path from switch w_4 and eventually reach the destination switch w_7 without traversing the WayPoint.

Conflict-Freedom (CF). We use this property to avoid transient congestion during the time that the routing of flows is updated. Consider Fig. 2 and assume that switch w_2 updates the forwarding path of flow f while f' still uses the link between switches w_2 and w_6. Consequently, two conflicting flows share the same link and create transient congestion in the network.

D. Update Model

The controller employs a discrete-time scheduling approach by dividing time into time slots to guarantee consistency properties. The controller works in a stop-and-wait fashion by sending rule-update messages only at the beginning of time slots. If the time slot duration is sufficiently long, all sent rule updates come into effect by the end of the time slot. We only consider fixed-length time slots, although it is possible to consider adaptive time slots using the approach presented in [9]. We use the experimental results reported in [8] to set the length of a time slot as the maximum propagation delay on any control-plane path plus the maximum processing delay of rule-updates on any network switch.

Time slotting permits the controller to control the order in which rule updates come into effect. To be specific, when the controller sends two rule-update messages in two consecutive time slots, the time-slotted system ensures that the first message applies before the second message. Assume that in time slot t, a packet of flow f arrives at switch w that has not received a rule-update message from the controller so far. Thus, the switch will forward the incoming packet via link $\ell \in \mathcal{L}_f^1$ such that $\ell.h = w$. Alternatively, switch w that has received a rule-update message in the first t time slots forwards the packet via link $\ell' \in \mathcal{L}_f^2$ such that $\ell'.h = w$. We define two notations to represent the forwarding behavior of switches in time slot t for packets of flow f. We use $P_f(w,t)$ to show the set of switches that a packet of flow f traverses if forwarded from switch w in time slot t. When the starting switch is the source switch s_f, we omit the first argument and

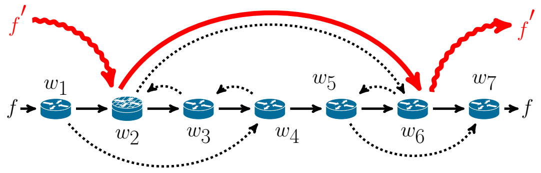


Fig. 2: Solid and dashed lines represent the initial and final paths of flow f, respectively. The thicker line partially shows the initial path of flow f' that has a conflict with flow f.

use the notation $P_f(t)$ (*i.e.*, $P_f(t) \equiv P_f(s_f, t)$). In time slot t, the *current path* of flow f is an ordered sequence of switches that a packet of f traverses to reach the destination switch d_f. $P_f(t)$ specifies the switches along the current path of flow f in time slot t. We call a switch *reachable* in time slot t if it is a member of $P_f(t)$.

To send out all the rule-update messages, the controller should partition them into a sequence of disjoint subsets and send them out in consecutive time slots. The order in which rule-update messages come into effect in the same time slot should not violate the consistency properties. We call this sequence of subsets a *rule-update schedule*. Let "*makespan*" denote the number of time slots required to apply a rule-update schedule. Note that conflicting flows affect the makespan of each other. To capture this interdependence when computing the makespan, we partition the flows into disjoint subsets in such a way that flows from different subsets do not have a conflict. To obtain these subsets, we pick a random flow and assign it to the first subset. Then, we add all flows that have a conflict with the selected flow to the first subset and repeat this process until no further flow can be added to the first subset. Further subsets can be constructed similarly. We denote the partition of flows by $C = \{C_1, \ldots, \ldots C_K\}$, where we assumed that the partition has K subsets. Then, we define $M(i)$ to show the makespan of the rule-update schedule for the flows in each $C_i \in C$. In the next subsection, we develop an optimization formulation to compute a rule-update schedule that minimizes the sum of $M(i)$'s. Since subsets in C are disjoint, the sum of $M(i)$'s is minimized when all $M(i)$'s are minimized individually and vice versa.

Discussion. Consider Fig. 3 for an example of the importance of explicitly modeling the WayPoint. Assume a solution approach in which we split the problem into two sub-problems, where the WayPoint is the destination and source of the first and second subproblems, respectively. But, updating all the switches preceding the WayPoint along the final path (w_1, w_8, and w_2) creates a loop. And, updating the WayPoint and its successors along the final path (w_4, w_6, w_5, w_3, and w_7) violates WPE.

IV. Formal Problem Definition

In this section, we formally define the problem of minimizing the **M**akespan of **C**onsistent **M**ulti-flow **U**pdate schedule and refer to it as **MCMU** in the rest. We extend the formulation of [18] to the multi-flow setting and add the CF property to it.

First, we describe the conditions to ensure that all rule-update messages associated with flows are sent to the corresponding switches, while LF, CF, and WPE properties are

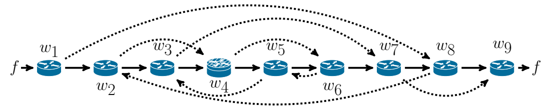


Fig. 3: Example in which updating all the switches that come after the WayPoint or before it in the final path violates one of the consistency properties.

preserved. Then, we formally specify the objective. Important notations are listed in Table III.

Delivery. To compute the rule-update schedule, we define the set of binary decision variables $x_f^w(t) \in \{0,1\}$ to indicate whether or not the controller sends a rule-update message for flow $f \in \mathcal{F}$ to switch $w \in \mathcal{W}_f$ in time slot t. Let T_i be the maximum possible value of t for any flow in the subset $C_i \in C$. Also, for all $f \in C_i$ define T_f to be equal to T_i. Since destination switches do not need a rule-update message in our model (*e.g.*, w_7 in Fig. 2 that is the destination switch for flow f) and we can send a rule-update message for at least one flow to one switch (otherwise we are blocked), T_i is equal to:

$$T_i = \textstyle\sum_{f \in C_i} \left(|\mathcal{W}_f| - 1\right). \qquad i \in \{1,\ldots,K\} \tag{1}$$

We ensure that the rule-update messages of flow f are delivered to the switches in $\mathcal{W}_f$ with the following constraint:

$$\textstyle\sum_{t \in \{1,\ldots,T_f\}} x_f^w(t) = 1. \qquad w \in \mathcal{W}_f - \{d_f\}, f \in \mathcal{F} \tag{2}$$

Loop-Freedom. To prevent loops in the path of flow f, we characterize the switches and links that receive a packet from s_f and enforce a strictly ascending order on the switches that forward packets of f in each time slot t. To this end, we first define binary variable $y_\ell^f(t)$, which is equal to 1 if switch $\ell.h$ forwards the traffic of flow f using the link ℓ in time slot t. Link ℓ in the initial path of flow f is in use while the controller has not sent an update message to switch $\ell.h$. However, link ℓ in the final path of flow f becomes available if the controller sends the rule-update message to switch $\ell.h$,

$$y_\ell^f(t) = 1 - \textstyle\sum_{t' \in \{1,\ldots,t\}} x_f^{\ell.h}(t'), \qquad t \in \{1,\ldots,T_f\}, \ell \in \mathcal{L}_f^1 \tag{3}$$

$$y_\ell^f(t) = \textstyle\sum_{t' \in \{1,\ldots,t\}} x_f^{\ell.h}(t'). \qquad t \in \{1,\ldots,T_f\}, \ell \in \mathcal{L}_f^2 \tag{4}$$

Next, we define binary variables $a_w^f(t)$ to represent whether switch $w \in \mathcal{W}_f$ receives packets from the source of f in time slot t or not, which happens if there is another switch in time slot t that forwards the traffic of f on a link that its tail is w. Since switches in time slot t are updated at different moments, we should consider all the active links of the previous time slot (*i.e.*, $t-1$) along with the links that will become active in the current time slot,

$$a_{\ell.t}^f(t) \geq a_{\ell.h}^f(t) \times y_\ell^f(t), \qquad t \in \{1,\ldots,T_f\}, \ell \in \mathcal{L}_f \tag{5}$$

$$a_{\ell.t}^f(t) \geq a_{\ell.h}^f(t) \times y_\ell^f(t-1), \qquad t \in \{1,\ldots,T_f\}, \ell \in \mathcal{L}_f \tag{6}$$

Note that the source always receives packets and thus $a_{s_f}^f(t) = 1$ for all t. Similarly, we define binary variables $z_\ell^f(t)$ to represent whether any packet of flow f is in transmission on link ℓ in time slot t or not. Similar to the definition of $a_w^f(t)$, we can write,

TABLE III: Formulation Notations.

Sym.	Description
$x_f^w(t)$	Update of switch w for flow f in time slot t
$y_\ell^f(t)$	Availability of link ℓ to flow f in time slot t
$a_w^f(t)$	Possible reachability of switch w from s_f in time slot t
$z_\ell^f(t)$	Existence of f's packets on link ℓ in time slot t
$o_w^f(t)$	Order of switch w in f's path in time slot t
$\overline{a}_w^f(t)$	Reachability of switch w from s_f without meeting the WayPoint

$$z_\ell^f(t) \geq a_{\ell.h}^f(t) \times y_\ell^f(t), \qquad t \in \{1,\ldots,T_f\}, \ell \in \mathcal{L}_f \tag{7}$$

$$z_\ell^f(t) \geq a_{\ell.h}^f(t) \times y_\ell^f(t-1). \qquad t \in \{1,\ldots,T_f\}, \ell \in \mathcal{L}_f \tag{8}$$

Finally, we impose a strictly ascending ordering on the switches that carry the packets of flow f in time slot t by defining integer decision variables $o_w^f(t) \in \mathbb{N}$ as follows:

$$o_{\ell.t}^f(t) \geq o_{\ell.h}^f(t) \times z_\ell^f(t) + 1. \qquad t \in \{1,\ldots,T_f\}, \ell \in \mathcal{L}_f \tag{9}$$

WayPoint-Enforcement. We define binary decision variables $\overline{a}_w^f(t)$ to represent whether packets of f in time slot t can reach switch w without going through Ω_f or not. A packet in the source has not traversed through the WayPoint yet if $\Omega_f \neq s_f$, thus we set $\overline{a}_{s_f}^f(t) = 1$. Otherwise, $\overline{a}_{s_f}^f(t) = 0$. Then, we can characterize $\overline{a}_w^f(t)$'s as,

$$\overline{a}_{\ell.t}^f(t) \geq \overline{a}_{\ell.h}^f(t) \times y_\ell^f(t), \qquad t \in \{1,\ldots,T_f\}, \ell \in \overline{\mathcal{L}}_f \tag{10}$$

$$\overline{a}_{\ell.t}^f(t) \geq \overline{a}_{\ell.h}^f(t) \times y_\ell^f(t-1). \qquad t \in \{1,\ldots,T_f\}, \ell \in \overline{\mathcal{L}}_f \tag{11}$$

Finally, we ensure that no packet can reach the destination without first going through Ω_f by enforcing $\overline{a}_{d_f}^f(t) = 0$.

Conflict-Freedom. For every subset of flows $Q \subseteq \mathcal{F}$ that have a conflict with each other on a specific link ℓ (*i.e.*, $X(Q,\ell) = 1$), we apply the following constraint to ensure that they do not share the link in the transient periods of updating the routes,

$$\textstyle\prod_{f \in Q} z_\ell^f(t) = 0. \qquad Q \subseteq \mathcal{F}, \ell \text{ s.t. } X(Q,\ell) = 1 \tag{12}$$

Makespan. To compute the makespan associated with the flows in the subset $C_i \in C$, we determine the latest time slot in which a rule-update message for some flow $f \in C_i$ is sent out to a switch in $\mathcal{W}_f$. To this end, we compute the largest value of any time slot t where $x_f^w(t)$ is equal to 1 for some flow $f \in C_i$ and switch $w \in \mathcal{W}_f$. We first obtain the largest time slot for each switch by summing the values of $t \times x_f^w(t)$ over all the time slots. Then, we use the inequality operator to capture the maximum time slot across all switches by using the following equation:

$$\textstyle\sum_{t \in \{1,\ldots,T_i\}} \left(t \times x_f^w(t)\right) \leq M(i). \quad w \in \mathcal{W}_f, f \in C_i, C_i \in C \tag{13}$$

Recall that the objective is to compute a rule-update schedule that minimizes the sum of $M(i)$'s. Therefore, we obtain the following objective,

$$\text{Min. } \textstyle\sum_{C_i \in C} M(i). \tag{14}$$

Theorem 1. *MCMU is NP-hard.*

Proof. We can demonstrate a reduction from [18], a proven NP-hard problem [16], by assuming that set $\mathcal{F}$ contains only one flow. Thus, if we could solve MCMU in polynomial time, the problem in [18] is also solved in polynomial time, which contradicts its NP-hardness. Thus, MCMU is NP-hard. □

V. TimeX: LP-based Algorithm

Since MCMU is NP-hard, exact methods such as branch-and-bound and cutting-plane can take a prohibitively long time to compute the optimal solution. So, we design TimeX, an approximation algorithm outlined in Algorithm 1, by using the solution of the Linear Program (LP) obtained from *relaxing* the integrality constraints in MCMU as guidance to reduce the makespan. Observe that constraints with multiplication operators (*e.g.*, (5) to (12)) are linearizable because $a^f_{\ell.h}(t)$, $y^f_\ell(t)$, and $z^f_\ell(t)$ are binary variables (see [39]). Without loss of generality, we assume that the number of CF constraints (12) is polynomial in the size of the problem input (*i.e.*, number of flows, time slots, and switches). Thus, the number of decision variables and constraints is polynomial in the input size. Therefore, the obtained LP is solvable in polynomial time with methods such as interior point [40]. We use the notation $\widetilde{x}^w_f(t) \in [0,1]$ to denote the optimal (possibly fractional) value of the decision variable $x^w_f(t)$ in the LP. The usage of the values of $\widetilde{x}^w_f(t)$ by TimeX is discussed in subsection V-A.

TimeX does not use the value of other decision variables besides $\widetilde{x}^w_f(t)$, and instead, uses efficient subroutines to guarantee the required consistency properties. To ensure LF and WPE, we use three easy-to-check conditions discussed in subsection V-C. Regarding CF, observe that constraint (12) is not affected by the relaxation process. So, in the solution of the relaxed problem, not all conflicting flows traverse a link in the same time slot. We only send a rule-update message to a switch when the solution of the relaxed problem indicates a non-zero probability for that update (*i.e.*, $\widetilde{x}^w_f(t) > 0$). Therefore, the output of TimeX respects constraint (12). We use the book-keeping mechanism of Section VI that records the existence of flows on links when constrain (12) is affected by the linearization process.

A. Randomized Acceptance of Solutions

TimeX uses the input data to construct the MCMU and then relaxes it, respectively in lines 1 to 2. Then, it uses an off-the-shelf LP solver to compute the optimal fractional solution and the values of $\widetilde{x}^w_f(t)$'s in line 3. Then, TimeX enumerates the flows in $\mathcal{F}$ and for each one constructs a series of schedules based on the values of $\widetilde{x}^w_f(t)$ and probabilistically accepts one of them. TimeX initializes a *map* data structure, denoted by $\mathcal{H}$, to store the computed rule-update schedules for flows in $\mathcal{F}$. We use $\mathcal{H}[f]$ to refer to the rule-update schedule of flow $f \in \mathcal{F}$ and $\mathcal{H}[f][t]$ to refer to the switches that receive a rule-update message for flow $f \in \mathcal{F}$ in time slot $t \in \{1, \ldots, T_f\}$.

For each flow in $\mathcal{F}$, TimeX defines a normalization coefficient named p_f to scale the acceptance probabilities based

Algorithm 1: TimeX: LP-based MCMU solver

```
   Input  : $C, \mathcal{F}, \{\mathcal{W}_f, \mathcal{L}^1_f, \mathcal{L}^2_f, s_f, d_f, \Omega_f | f \in \mathcal{F}\}, \Gamma$
   Output: Rule-update schedule $\mathcal{H}$ for all flows in $\mathcal{F}$
 1 MCMU ← Construct the problem based on input values
 2 $\widetilde{\text{MCMU}}$ ← Relax(MCMU)
 3 $\{\widetilde{x}^w_f(t)\}$ ← Solve($\widetilde{\text{MCMU}}$)            // Using existing LP solvers
 4 $\mathcal{H}$ ← Map()                               // A map to store schedules
 5 for $f \in \mathcal{F}$ do
 6     $p_f \leftarrow 0$
 7     $\mathcal{H}[f] \leftarrow$
         SchedulerFinder($\mathcal{W}_f, \mathcal{L}^1_f, \mathcal{L}^2_f, \Omega_f, s_f, d_f, \{\widetilde{x}^w_f(t)\}, \Gamma$)
 8     if $\mathcal{H}[f]$ is empty then
 9         goto line 5                                        // Failure
10     $\pi_f \leftarrow \min_{t \in \{1,\ldots,T_f\}} \min_{w \in \mathcal{H}[f][t]} \{\widetilde{x}^w_f(t)\}$
11     if rand() $< \pi_f/(1 - p_f)$ then
12         for $t \in \{1, \ldots, T_f\}$ do
13             for $w \in \mathcal{H}[f][t]$ do
14                 $\widetilde{x}^w_f(t) \leftarrow \widetilde{x}^w_f(t) - \pi_f$
15         $p_f \leftarrow p_f + \pi_f$
16         goto line 7                                     // Try again
17 return $\mathcal{H}$
```

on the probability of previously rejected schedules in line 6. For example, assume that there are three schedules, where the acceptance probability of each one is $1/3$. If we reject the first schedule, we will have two schedules with equal probabilities. Thus, we should scale their acceptance probabilities to $\frac{1/3}{1-1/3} = 1/2$. If we reject the second schedule, only one schedule remains. So, we should scale its acceptance probability to $\frac{1/3}{1-(1/3+1/3)} = 1$. Thus, p_f shows the sum of the probability of rejected schedules and the acceptance probability of a new schedule is scaled by dividing it by $1 - p_f$.

TimeX employs a subroutine called SchedulerFinder to obtain a feasible rule-update schedule in line 7. SchedulerFinder is a limited-backtrack search mechanism that is explained in subsection V-B. If SchedulerFinder can not find a schedule for the flow, it is necessary to use a fall-back algorithm (*e.g.*, tagging-based solutions) to perform the update. However, when a schedule is found, an acceptance probability π_f is assigned to it in line 10. π_f is set to the minimum value of variables $\widetilde{x}^w_f(t)$, where switch w receives its rule-update message in time slot t, according to $\mathcal{H}[f]$. Then, TimeX uses a uniform random number generator to accept or reject the computed schedule in line 11. Upon acceptance, the algorithm proceeds to compute a schedule for the next flow in the next iteration of the for loop in line 5. When $\mathcal{H}[f]$ is rejected, we subtract π_f from the corresponding fractional values $\widetilde{x}^w_f(t)$ and repeat the process (see line 16). This subtraction helps to find a different schedule in the next iteration. Then, we add π_f to p_f in line 15. Our complexity analysis in Appendix B demonstrates that this process terminates in polynomial time. We analyze the performance of TimeX in Appendix C.

B. Computing a Complete Schedule

SchedulerFinder, outlined in Algorithm 2, is a limited-backtrack search mechanism [41] that calls a subroutine called SwUpdate in successive time slots until all switches receive their corresponding update messages. Throughout the search, the algorithm is allowed to undo Γ steps and take $\Gamma + 1$ new steps. Specifically, SchedulerFinder starts by initializing five variables: (1) τ that represents the time slot, (2) $\mathcal{S}$ is a map that shows which switches receive a rule-update message

Algorithm 2: SchedulerFinder: LP-based schedule finder

```
   Input  : $\mathcal{W}_f, \mathcal{L}_f^1, \mathcal{L}_f^2, \Omega_f, s_f, d_f, \{\tilde{x}_f^w(t)\}, \Gamma$
   Output: Rule-update schedule $\mathcal{S}$ for flow $f$
1  $\tau \leftarrow 1$, $\mathcal{S} \leftarrow$ Map(), $\overline{\mathcal{S}} \leftarrow$ Map(), $\gamma \leftarrow \Gamma$, $\mathcal{V} \leftarrow$ set()
2  while $|\mathcal{V}| < |\mathcal{W}_f| - 1$ do
3      $\pi \leftarrow$ SwUpdate($\mathcal{L}_f^1, \mathcal{L}_f^2, \Omega_f, s_f, d_f, \{\tilde{x}_f^w(t)\}, \tau, \mathcal{V}, \overline{\mathcal{S}}[\tau]$)
4      if $\pi \neq \{\}$ then
5          $\mathcal{S}[\tau] \leftarrow \pi$                    // Target switches in time slot $\tau$
6          $\mathcal{V}$.add($\pi$)
7          $\tau \leftarrow \tau + 1$
8          $\gamma \leftarrow \min\{\Gamma, \gamma + 1\}$
9      else if $\gamma = \Gamma$ then
10         $\gamma \leftarrow \max\{-1, \Gamma - \tau - 1\}$          // Perform a backtrack
11         $\tau \leftarrow \max\{0, \tau - \Gamma\}$
12         $\overline{\mathcal{S}}[\tau] \leftarrow \mathcal{S}[\tau]$
13         $\forall t \geq \tau$ : delete $\mathcal{S}[\tau]$
14         $\mathcal{V} \leftarrow \bigcup_{t \in \{1,\ldots,\tau-1\}} \mathcal{S}[t]$
15     else if $|\mathcal{W}_f| - 1 - |\mathcal{V}| \leq T_f - \tau$ then
16         $\tau \leftarrow \tau + 1$
17         $\gamma \leftarrow \min\{\Gamma, \gamma + 1\}$
18     else
19         return {}
20 return $\mathcal{S}$
```

in a given time slot, (3) $\overline{\mathcal{S}}$ is an auxiliary map that stores information about the unsuccessful search decisions to avoid them subsequently, (4) γ is an auxiliary variable that is used to implement the limited-backtrack mechanism, and (5) $\mathcal{V}$ is a set that stores updated switches. Then, in line 3, the SwUpdate subroutine is called to compute a set of switches π that can receive a rule-update message in time slot τ. SwUpdate considers the fractional solution and obtains a set with the higher values of $\tilde{x}_f^w(\tau)$ while assuming that the switches in $\mathcal{V}$ are updated in previous time slots (*i.e.*, before τ) and avoiding solutions in $\overline{\mathcal{S}}$. If SwUpdate computes a non-empty set of switches to be updated, they are added to $\mathcal{V}$ and stored in the τ-th entry of $\mathcal{S}$. Then, the time slot is incremented by one (see lines 4-7). If SwUpdate can not update any switch due to the consistency properties, it attempts to perform a backtrack, which is viable when the value of γ is equal to Γ. Then, the value of γ is set to -1, and for each non-backtracking step in the future, γ is incremented by one but never greater than Γ (see line 8) to ensure that the algorithm takes $\Gamma + 1$ new steps before doing another backtrack. If backtrack is not possible but the number of remaining time slots is bigger than or equal to the number of switches that need a rule-update message, it is still possible to update all switches. So, SwUpdate continues without updating any switch (see lines 15 to 17). Finally, if it is not possible to proceed, the subroutine returns an empty set to indicate a failure.

C. Determining Target Switches in a Specific Time Slot

In this subsection, we design an algorithm named SwUpdate that finds a subset of switches that can receive a rule-update message for a specific flow in a given time slot. To this end, we assume that a feasible value assignment for the decision variables $o_w^f(t)$, which represent an ordering on the set of switches (see equation (9)), is available. In appendix A, we present a procedure, called OrderCompute, that computes these values efficiently. Then, we employ the following conditions on the values of $o_w^f(t)$'s to determine which switches can receive a rule-update message for flow f in time slot t without violating WPE or LF properties.

Algorithm 3: SwUpdate: Target switches in a time slot

```
   Input  : $\mathcal{L}_f^1, \mathcal{L}_f^2, \Omega_f, s_f, d_f, \{\tilde{x}_f^w(t)\}, t, \mathcal{V}, \overline{\mathcal{S}}$
   Output: Set of target switches for flow $f$ in time slot $t$
1  Compute $P_f(t)$ based on $\mathcal{V}$
2  $\{o_w^f(t)\} \leftarrow$ OrderCompute($\mathcal{L}_f^1, \mathcal{L}_f^2, P_f(t), \mathcal{V}$) // See Appendix A
3  best_sol $\leftarrow$ {}
4  $x^\star_{\text{best\_sol}} \leftarrow -\infty$
5  for $w \in P_f(t)$ do
6      if $w \notin \mathcal{V}$ and $x^\star_{best\_sol} < \tilde{x}_f^w(t)$ and $0 < \tilde{x}_f^w(t)$ then
7          $Q \leftarrow$ priority_queue(($w$, [$w$]), $\overline{\mathcal{S}}$)
8          while $Q$ is not empty do
9              $n, p \leftarrow$ Q.dequeue()
10             if $n = d_f$ then
11                 Remove updated switches from $p$
12                 best_sol $\leftarrow p$
13                 $x^\star_{\text{best\_sol}} \leftarrow \min_{w' \in p}\{\tilde{x}_f^{w'}(t)\}$
14                 break
15             $M \leftarrow \{\}$
16             if $n \in \mathcal{V}$ then
17                 $M$.append($\ell.t : \ell \in \mathcal{L}_f^2$ and $\ell.h = n$)
18             else
19                 if $1 - \tilde{x}_f^n(t) \geq \tilde{x}_f^w(t)$ then
20                     $M$.append($\ell.t : \ell \in \mathcal{L}_f^1$ and $\ell.h = n$)
21                 if $\tilde{x}_f^n(t) \geq \tilde{x}_f^w(t)$ then
22                     $M$.append($\ell.t : \ell \in \mathcal{L}_f^2$ and $\ell.h = n$)
23             for $m \in M$ do
24                 if Eq. (15) or (16) holds and Eq. (12) holds then
25                     $p' \leftarrow p + [m]$              // Append $m$ to $p$
26                     Q.enqueue(($m, p'$))
27 sol2 $\leftarrow \{w | \tilde{x}_f^w(t) \geq x^\star_{\text{best\_sol}}$ and $w \notin P_f(f)\}$
28 if $|sol2| > |best\_sol|$ then
29     Return sol2
30 else
31     Return best_sol
```

$$o_{\Omega_f}^f(t) \leq o_w^f(t) < o_{w'}^f(t), \quad \ell \in \mathcal{L}_f^2, \ell.h = w, \ell.t = w' \tag{15}$$

$$o_w^f(t) < o_{w'}^f(t) \leq o_{\Omega_f}^f(t), \quad \ell \in \mathcal{L}_f^2, \ell.h = w, \ell.t = w' \tag{16}$$

$$w \notin P_f(t)\,. \tag{17}$$

Condition (15) states that a packet in w has already passed through the WayPoint Ω_f and it will be forwarded to w' which has a strictly higher $o_{w'}^f(t)$ and thus the packet would not enter a routing loop. Condition (16) states that a packet in w will pass through the WayPoint after leaving w' and similar to (15) this condition ensures the LF property. Condition (17) states that when no packet can reach w from the source of f, we can change its routing behavior without violating LF or WPE properties. Based on conditions (15)-(17), we compute a path of switches from source to destination that has at least one new updated switch, respects the consistency properties, and maximizes the minimum value of $\tilde{x}_f^w(\tau)$'s.

SwUpdate is outlined in Alg. 3. The inputs to this subroutine are: a specific time slot t, the fractional solution $\{\tilde{x}_f^w(t)\}$, set of all updated switches in the previous time slots $\mathcal{V}$ (is empty is in the first time slot), and set of switches that should not be updated together $\overline{\mathcal{S}}$. Then, SwUpdate computes a set of switches that can receive their corresponding rule-update messages in time slot t, where their minimum value of the fractional value $\tilde{x}_f^w(t)$ is maximized. Maximization with respect to the values of the fractional solution guides the algorithm to select a subset of switches that provide a higher probability of minimizing the overall makespan. We search for the best set of switches in lines 5 to 26 and save the best set of switches found so far

TABLE IV: Example values for flow f in Fig. 2.

	$\widetilde{x}_f^w(t)$						$o_w^f(t)$					
	t=1	t=2	t=3	t=4	t=5	t=6	t=1	t=2	t=3	t=4	t=5	t=6
w_1	0.00	0.50	0.50	0.00	0.00	0.00	1	1	1	1	1	1
w_2	0.50	0.00	0.50	0.00	0.00	0.00	2	2	2	4	4	4
w_3	0.00	0.50	0.50	0.00	0.00	0.00	3	3	2	3	3	3
w_4	0.00	0.75	0.00	0.25	0.00	0.00	4	3	2	2	2	2
w_5	0.50	0.00	0.25	0.00	0.25	0.00	5	3	3	5	6	6
w_6	0.00	0.88	0.00	0.00	0.00	0.12	6	3	3	5	5	5
w_7	-	-	-	-	-	-	7	4	4	6	6	7

and its corresponding minimum $\widetilde{x}_f^w(t)$ in variables best_sol and $x^{\star}_{\text{best\_sol}}$, respectively. In multiple rounds, we select a switch $w \in P_f(t)$ and assume that it is one of the target switches that receive a rule-update message in time slot t. Note that we only consider switch w when its update probability is greater than the best solution so far (see line 7). We use a priority queue Q to find the best path from switch w to the destination switch d_f. This priority queue favors the paths with the highest number of updated switches. Therefore, among the subsets with the same value of $\widetilde{x}_f^w(t)$, the one which updates more switches is selected. Also, the priority queue gets $\overline{\boldsymbol{S}}$ as an input, and when the current search path contains all the switches in $\overline{\boldsymbol{S}}$, it will delete the path to prevent exploring previous search branches. In lines 15 to 22, the next switches that can be explored are computed based on the status of the current switch (whether it has been updated in previous time slots or not) and the value of their corresponding fractional variables. In lines 23 to 26, we use the order variables $o_w^f(t)$ and conditions (15)-(16) to explore only the search branches that preserve the network consistency properties (LF and WPE). Finally, upon finding a path to the destination switch, the algorithm updates the best solution and breaks the while loop to restart its search from the next switch in the current path $P_f(t)$. We also select a subset of unreachable switches with fractional variables $\widetilde{x}_f^w(t) \geq x^{\star}_{\text{best\_sol}}$ in line 27. If this set (*i.e.*, sol2) becomes equal to $\overline{\boldsymbol{S}}$, we remove the switch with the smallest value of $\widetilde{x}_f^w(t)$ from sol2. Finally, SwUpdate returns the bigger set of either unreachable switches or those switches that constitute a new path from s_f to d_f in lines 28 to 31.

Example 1. Consider the values of $\widetilde{x}_f^w(t)$ in Table IV for the flow f in Fig. 2. Based on the values of order variables, obtained from the OrderCompute algorithm, and conditions (15)-(17), only w_2 and w_5 can receive a rule-update message in the first time slot. Since the value of $\widetilde{x}_f^w(t)$ for w_5 is not bigger than w_2's value, SwUpdate selects w_2 after executing codes in lines 5-26. For the second time slot, no switch in the current path can receive a rule-update message, and thus `best_sol` will be empty. However, SwUpdate selects the unreachable switches w_3 and w_4 (stored in `sol2` in line 27), as their $\widetilde{x}_f^w(t)$ values are greater than 0. In the third time slot, switch w_1 is selected to receive its rule-update message, as its $\widetilde{x}_f^w(t)$ value is bigger than w_5's value. Based on the values of $\widetilde{x}_f^w(t)$ no switch can receive a rule-update message in the fourth time slot. In the fifth and sixth time slots, only the values of $\widetilde{x}_f^w(t)$ for w_5 and w_6 are greater than zero. Therefore, these switches receive their rule-update messages accordingly and the rule-update process terminates.

Algorithm 4: RMS: LP-independent MCMU solver

```
Input  : $\mathcal{F}, \{\mathcal{W}_f, \mathcal{L}_f^1, \mathcal{L}_f^2, s_f, d_f, \Omega_f \,|\, f \in \mathcal{F}\}$
Output: Rule-update schedule $\boldsymbol{\mathcal{H}}$ for all flows in $\mathcal{F}$
1  $\{z_\ell^f(t) \mid \ell \in \mathcal{L}_f, t \in \{1, \ldots, T_f\}, f \in \mathcal{F}\} \leftarrow \mathbf{0}$
2  $\boldsymbol{\mathcal{H}} \leftarrow$ Map()                     // A map to store schedules
3  for $f \in \mathcal{F}$ do
4      $\alpha \leftarrow$ FindSchedule($\mathcal{W}_f, \mathcal{L}_f^1, \mathcal{L}_f^2, \Omega_f, \{z_\ell^{f'}(t) | f' \in \mathcal{F}\}$)
5      $\beta \leftarrow$ FindSchedule($\mathcal{W}_f, \mathcal{L}_f^2, \mathcal{L}_f^1, \Omega_f, \{z_\ell^{f'}(t) | f' \in \mathcal{F}\}$)
6      $\boldsymbol{\mathcal{H}}[f] \leftarrow$ Merge($\alpha, \beta$)          // See Algorithm 6
7      for $\ell \in \mathcal{L}_1^f \cup \mathcal{L}_f^2$ and $t \in \{1, \ldots, T_f\}$ do
8          if $f$ uses $\ell$ at $t$ or $t-1$ then
9              $z_\ell^f(t) \leftarrow 1$
10 return $\boldsymbol{\mathcal{H}}$
```

VI. RMS: LP-INDEPENDENT ALGORITHM

The TimeX algorithm runs in polynomial time (see Appendix B) but relies on the generic linear program solvers, which eventually limits its scalability. The complexity of solving LPs is $O(n^{3.5}L)$, where n is the number of variables (in MCMU $n = O(T_i|\mathcal{W}_f||\mathcal{F}|)$) and L is the number of required bits to represent the problem. So, time of solving the LP can dominate the runtime of TimeX. So, we design a more scalable heuristic algorithm, which we call **R**educed-**M**akespan **S**cheduler (**RMS**). RMS applies OrderCompute subroutine that we employed in Section V to compute the values of decision variables $o_w^f(t)$ and uses equations (15) to (17) to send rule-update messages to a large number of switches in successive time slots. Algorithm 4 outlines RMS, where it starts by computing a rule-update schedule for each flow by calling a search-based mechanism called FindSchedule in line 4. Then, RMS swaps the initial and final paths (*i.e.*, $\mathcal{L}_f^1$ and $\mathcal{L}_f^2$) and calls FindSchedule for the second time in line 5 to obtain a possibly different rule-update schedule. The possibility of obtaining the second schedule is based on an important property called reversibility that we will demonstrate later in this section. Then, RMS uses the Merge algorithm, defined in subsection VI-C, to combine obtained schedules and improve the makespan of the rule-update schedule. RMS uses a set of variables $z_\ell^f(t)$ to handle inter-flow dependencies that are specified by function $X(Q, \ell)$. After computing the rule-update schedule for each flow f, RMS sets the variable $z_\ell^f(t)$ to 1 to indicate that link ℓ is used by flow f during the time slot t. See lines 7 to 9. RMS passes these variables to FindScheudle subroutine to ensure that CF property is respected.

A. FindSchedule Algorithm

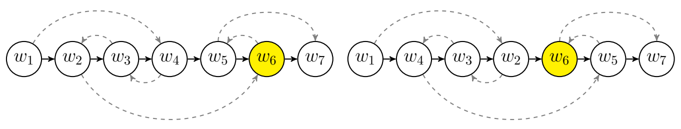
We first explain a heuristic to obtain a large subset of switches that can receive a rule-update message while respecting consistency properties. Then, we design a search mechanism to send out all rule-update messages in an iterative manner during successive time slots.

Determining Target Switches. We define $\mathcal{I}_f(t)$ to show the set of those switches that satisfy one of the equations (15) to (17) in time slot t. In [36], we proved that it is possible to send a rule-update message to at least half of the switches in $\mathcal{I}_f(t)$ while preserving the LF and WPE properties. The proof is based on partitioning the set $\mathcal{I}_f(t)$ into reachable switches $\mathcal{U}_f(t) = \mathcal{I}_f(t) \cap P_f(t)$ and unreachable switches $\overline{\mathcal{U}}_f(t) = \mathcal{I}_f(t) - \mathcal{U}_f(t)$.

Algorithm 5: FindSchedule: Heuristic schedule finder

```
   Input  : $\mathcal{W}_f, \mathcal{L}_f^1, \mathcal{L}_f^2, \Omega_f, \{z_\ell^{f'}(t)\}$
   Output: A rule-update schedule for flow $f$
1  $\theta \leftarrow \{x_f^w(0) = 0 | w \in \mathcal{W}_f - \{d_f\}\}$
2  Q ← PriorityQueue()
3  Q.push(($\theta$, []))
4  while $Q$ is not empty and $\xi_1 > 0$ do
5      $\theta, \Theta \leftarrow$ Q.pop()
6      Compute $P_f(t)$ based on $\theta$
7      $\{o_w^f(t)\} \leftarrow$ OrderCompute($G_f, P_f(t), \Theta$)
8      $\mathcal{I}_f(t) \leftarrow$ Compute based on $\{o_w^f(t)\}$ and conditions (15)-(17)
9      $\mathcal{U}_f(t) \leftarrow \mathcal{I}_f(t) \cap P_f(t)$
10     $\overline{\mathcal{U}}_f(t) = \mathcal{I}_f(t) - \mathcal{U}_f(t)$
11     if $\mathcal{U}_f(t) \cup \overline{\mathcal{U}}_f(t) = \{\}$ then
12         $\xi_1 \leftarrow \xi_1 - 1$
13     for $k \in \{1, \ldots, \xi_2\}$ do
14         $\mathcal{V} \leftarrow k$-th set in $\mathbb{P}(\mathcal{U}_f(t)) \cup \mathbb{P}(\overline{\mathcal{U}}_f(t))$
15         for $w \in \mathcal{W}_f - \{d_f\}$ do
16             $\ell \leftarrow$ outgoing link of $w$ based on $\mathcal{L}_f^2$
17             $\mathcal{Z} \leftarrow \{f' | z_\ell^{f'}(t+1) = 1\} \cup \{f\}$
18             if $w \in \mathcal{V}$ and $X(\mathcal{Z}, \ell) = 0$ then
19                 $\theta'[x_f^w(t+1)] \leftarrow 1$
20             else
21                 $\theta'[x_f^w(t+1)] \leftarrow \theta[x_f^w(t)]$
22         if $x_f^w(t+1) = 1, \forall x_f^w(t+1) \in \theta'$ then
23             return $\Theta$.append([$\theta$, $\theta'$])
24         else
25             $\Theta' \leftarrow \Theta$.append($\theta$)
26             Q.push(($\theta'$, $\Theta'$))
27 return No Solution
```

Sending All Messages. We design the FindSchedule procedure, presented in Algorithm 5, that starts from the network configuration where none of the switches in the path of a specific flow have received their corresponding rule-update message and searches for the network configuration where every switch is updated. We use a map data structure from variables $x_f^w(t)$ to $\{0, 1\}$ to represent a network configuration. The first configuration, called θ, is defined in line 1, where all $x_f^w(t)$'s are set to 0. We use the bracket notation to set or get the value of a specific variable (*e.g.*, $\theta[x_f^w(t)]$). FindSchedule uses a priority queue to store the network configurations. Then, it uses the stored configurations to compute the next configurations, until it finds the final configuration, where every switch has received its corresponding rule-update message. The priority queue prioritizes the network configurations with the higher number of updated switches, which significantly increases the probability of finding the target configuration. FindSchedule uses OrderCompute in Algorithm 7 to efficiently compute the values of variables $o_w^f(t)$ and then specifies the sets $\mathcal{U}_f(t)$ and $\overline{\mathcal{U}}_f(t)$. Based on [36], we know that either of these two sets can be updated without violating the consistency properties. Therefore, FindSchedule considers the subsets of these two sets to search the network configurations. To this end, we used $\mathbb{P}(.)$ to denote the power set function that determines all subsets of a given set in line 14. We determine the set of flows that use each link based on the value of $z_\ell^f(t)$ variables in lines 16 and 17. Then, we use the condition in line 18 to ensure that conflicting flows do not share that link. Parameters ξ_1 and ξ_2 control the runtime by, respectively, limiting the number of backtracks and the branching factor. Thus, the time complexity of FindSchedule is $O(\xi_1 \xi_2 |\mathcal{W}_f|^2 \log |\mathcal{W}_f|)$, as the maximum length of a search branch and the time complexity of OrderCompute subroutine are both $O(|\mathcal{W}_f|)$. The time complexity of insertion into a priority queue is $O(\log |\mathcal{W}_f|)$.



(a) Original Problem. (b) Transformed Problem.

Fig. 4: FindSchedule algorithm finds a better rule-update schedule when we swap the initial path and final paths.

B. Reversibility Property

We demonstrate the existence of a property in the MCMU, called *reversibility*, which can be used to obtain rule-update schedules with shorter makespans. Reversibility allows the algorithm to transform the input problem by swapping the initial and final paths, and obtain another schedule (we have to apply it in reverse order) with a potentially shorter makespan. We demonstrate the reversibility with an example. For the formal proof refer to [36].

Example 2. Consider the problem in Fig. 4(a). If we update the maximum number of switches in the first time slot (*i.e.*, $\{w_1, w_2\}$), we have to send a rule-update message to only one switch in the next 4 time slots. Specifically, in the second time slot, only switch w_3 can be updated because updating switch w_4 or w_6 creates the loops ($w_4 \rightarrow w_3 \rightarrow \ldots$) and ($w_6 \rightarrow w_5 \rightarrow \ldots$), respectively, and updating switch w_5 violates WPE. In the third time slot, we can only update switch w_4. Finally, switches w_5 and w_6 are updated in the fourth and fifth time slots, respectively. Therefore, the makespan of the resulting rule-update schedule is 5 time slots. However, if we transform the problem by swapping the initial path and final path (see Fig. 4(b)), we can find a rule-update schedule with 3 time slots-long makespan. We can update $\{w_1, w_6\}$ in the first time slot, $\{w_3, w_4, w_5\}$ in the second time slot, and $\{w_2\}$, in the third time slot. If we apply the second schedule in reverse order, the original problem in Fig. 4(a) is solved in 3 time slots.

C. Merge Algorithm

We propose Merge, outlined in Algorithm 6, to derive a third schedule from two given schedules with a possibly shorter makespan. Merge gets two schedules α and β, and creates a graph, called G_m, by adding a node for every distinct network configuration that is reachable in schedules α and β. G_m has two sets of nodes $n_\alpha(t)$ and $n_\beta(t)$, which are, respectively, the set of switches that have received their intended rule-update message in time slots $\tau \leq t$ if the controller follows schedule α and β (see lines 5 and 8). Also, two dummy nodes n_s and n_d represent the initial and final network configurations, respectively. Each edge in G_m represents a transition between two network configurations, where a subset of switches receives their rule-update messages while consistency properties are preserved. Clearly, for each pair $n_\alpha(t-1), n_\alpha(t)$ or $n_\beta(t-1), n_\beta(t)$ the transition preserves consistency properties because they are intermediate steps of the schedules α and β, respectively, which we know preserve the consistency properties at every moment (see lines 10 and

Algorithm 6: Merge: Merge two rule-update schedules

```
   Input  : $\alpha$, $\beta$: Two rule-update schedules
   Output: A rule-update schedule from merging $\alpha$ and $\beta$
 1 $G_m(\mathcal{N}_m = \{\}, \mathcal{E}_m = \{\})$
 2 $\rho_\alpha, \rho_\beta \leftarrow |\alpha|, |\beta|$
 3 for $t \in \{1, \dots, \rho_\alpha\}$ do
 4     $n_\alpha(t) = \{w | \forall \tau \le t : x_f^w(\tau) = 1 \text{ in } \alpha\}$
 5     $\mathcal{N}_m$.add_node($n_\alpha(t)$)
 6 for $t \in \{1, \dots, \rho_\beta\}$ do
 7     $n_\beta(t) = \{w | \forall \tau \le t : x_f^w(\tau) = 1 \text{ in } \beta\}$
 8     $\mathcal{N}_m$.add_node($n_\beta(t)$)
 9 for $t \in \{2, \dots, \rho_\alpha\}$ do
10     $\mathcal{E}_m$.add_edge($n_\alpha(t)$, $n_\alpha(t-1)$)
11 for $t \in \{2, \dots, \rho_\beta\}$ do
12     $\mathcal{E}_m$.add_edge($n_\beta(t)$, $n_\beta(t-1)$)
13 $\mathcal{N}_m$.add_nodes($\{n_s, n_d\}$)
14 $\mathcal{E}_m$.add_edge($n_s$, $n_\alpha(1)$), $\mathcal{E}_m$.add_edge($n_s$, $n_\beta(1)$)
15 $\mathcal{E}_m$.add_edge($n_\alpha(\rho_\alpha)$, $n_d$), $\mathcal{E}_m$.add_edge($n_\beta(\rho_\beta)$, $n_d$)
16 for $n_i(t), n_j(t') \in \mathcal{N}_m$ and $i, j \in \{\alpha, \beta\}$ do
17     if $n_j(t') \subseteq \mathcal{U}_f^i(t) \cup n_i(t)$ or $n_j(t') \subseteq \overline{\mathcal{U}}_f^i(t) \cup n_i(t)$ then
18         if Outgoing link of $w \in n_j(t') - n_i(t)$ is not occupied by
             conflicting flows in time slot $t+1$ then
19             $\mathcal{E}_m$.add_edge($n_i(t)$, $n_j(t')$)
20 return Sets along the shortest path from $n_s$ to $n_d$
```

12). Moreover, for each pair of nodes $n_\alpha(t), n_\beta(t')$ we check whether or not it is possible to send rule-update message to all the switches that are in $n_\beta(t')$ but not in $n_\alpha(t)$ without violating the consistency properties, and add a new edge in between $n_\alpha(t)$ and $n_\beta(t')$. These new edges provide the opportunity to create a schedule from fragments of either α and β with fewer time slots. Finally, we obtain a valid schedule by following the configurations on the shortest path from n_s to n_d.

Example 3. Consider Fig. 5 that presents the graph G_m for the schedules in Fig. 4. The square nodes in the first and second rows represent schedules α and β, respectively. The shortest schedule with three time slots can be obtained from two different paths.

Lemma 1. *The time complexity of* Merge *is* $O(|\mathcal{W}_f|^3)$.

Proof. G_m has at most $2|\mathcal{W}_f|+2$ nodes and $|\mathcal{W}_f|^2+2|\mathcal{W}_f|+2$ edges. We use order variables of Appendix A to specify the edges in G_m, which requires $O(|\mathcal{W}_f|)$ per edge. So, we can construct the graph and use the Breadth-First Search algorithm to find the shortest path from n_s to n_d in $O(|\mathcal{W}_f|^3)$. □

VII. Performance Evaluation

In this section, we present the evaluation of the proposed algorithms: TimeX and RMS.

Evaluation Methodology. The first set of results in subsection VII-A shows the behavior of algorithms by using simulations. These results demonstrate the runtime, success rate, and makespan minimization ability of our algorithms by focusing on the *single flow* setting. The second set of results, presented in subsection VII-B, use realistic network conditions implemented in Mininet [42] to show the ability to prevent transient congestion in a *multi-flow* setting.

Comparison Methods. We evaluate TimeX and RMS by comparing them with MIP [18] and GRD [20]. The former solves the integer formulation directly. The latter computes the maximum number of switches in each time slot (an NP-hard problem) as a heuristic for minimizing the makespan. Although this greedy approach exhibits good performance on

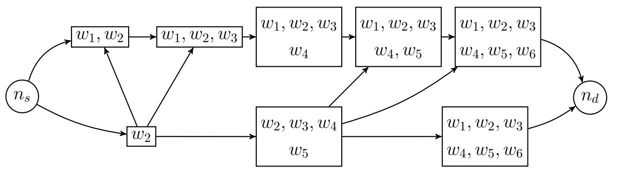

n_s
w_1, w_2
w_1, w_2, w_3
w_1, w_2, w_3 w_4
w_1, w_2, w_3 w_4, w_5
w_1, w_2, w_3 w_4, w_5, w_6
w_2
w_2, w_3, w_4 w_5
w_1, w_2, w_3 w_4, w_5, w_6
n_d

Fig. 5: Merging two schedules for the problem in Fig. 4.

average, its worst-case performance is not bounded. We also present the results of running the FindSchedule procedure (see Alg. 5) on the original and transformed problem instances (see Section VI) and denote them by ORG and SWP, respectively. We implemented all algorithms in Python and used Gurobi 9.0 [43] to solve the optimization problems. All algorithms terminate after 1 second if they don't find a solution. We conducted the experiments on a machine with Intel(R) Core(TM) i7-8550U at 1.80-1.99 GHz CPU and 16.0 GB RAM.

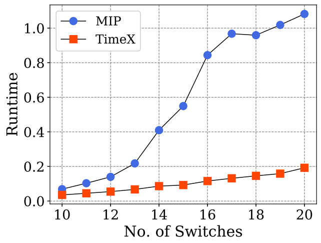
Dataset. In simulations of subsection VII-A, we used 3265 instances of the WayPoint-enforced single-flow rule-update problem from a public dataset that is publicly available at [44] with 10 to 20 switches in each instance.

Time slot. In emulations of subsection VII-B, we assume that the round-trip delay between switches and controller is 50 milliseconds (similar to [8]) and per-rule update time is between 3 and 18 milliseconds (based on [8]). Thus we let each time slot be $80 > 18 + 50$ milliseconds. Consequently, the controller waits for an extra 80 milliseconds before sending new rule-update messages to the switches. Although this value is larger than the typical delays in local-area networks, we adopted it to account for the uncertainties of the emulation environment and the software-based sleep mechanism.

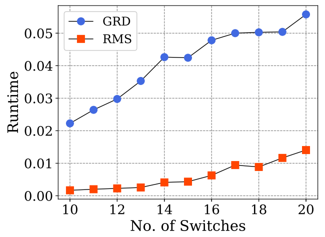
Topology. Each problem instance in the simulation has at least 10 switches and 18 links and has at most 20 switches and 38 links (see Fig. 2 for an example instance). In Mininet experiments, we implemented the G-Scale topology [38] that has 12 switches and 18 links, where the bandwidth of each link is selected from the interval $[1, 15]$ Mbps.

A. Simulation Results

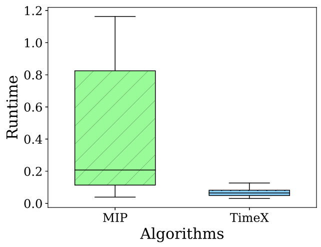
Runtime. Figure 6(a) shows the average runtime of TimeX and MIP as the number of switches increases from 10 to 20. On average, TimeX consistently is better than MIP. Although their difference is small at the beginning, MIP reaches the 1 second time limit for updating more than 17 switches. However, TimeX exhibits great scalability, where its average runtime reaches 0.2 seconds for the largest problem instances. The average and maximum speed-up achieved by TimeX in Fig. 6(a) are 4.9x and 7.3x, respectively. To investigate the runtime in more detail, we also presented the range of runtime values obtained by TimeX and MIP for 10 to 14 and 15 to 20 switches in Figs. 6(c) and 6(d), respectively. Figure. 6(c) shows that even for a considerable number of instances with 10 to 14 switches MIP encounters the 1 second time limit. Figure. 6(d) shows that for the majority of problem instances with 15 to 20 switches MIP incurs at least 1 second of delay. However, TimeX spends less than 0.2 seconds to find a solution for the majority of problem instances (consider the upper-whiskers of box plots). To analyze the average runtime of RMS, we compared it with GRD in Fig. 6(b) for the increasing number



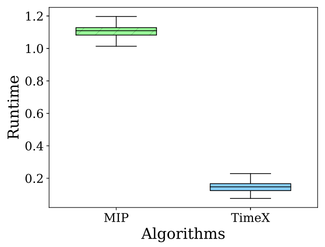
(a) Effect of number of switches on the runtime of MIP and TimeX.



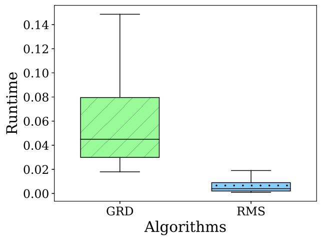
(b) Effect of number of switches on the runtime of GRD and RMS.



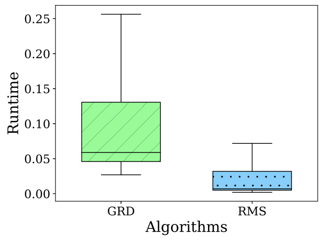
(c) Runtime range with 10 to 14 switches for MIP and TimeX.



(d) Runtime range with 15 to 20 switches for MIP and TimeX.



(e) Runtime range with 10 to 14 switches for GRD and RMS.



(f) Runtime range with 15 to 20 switches for GRD and RMS.

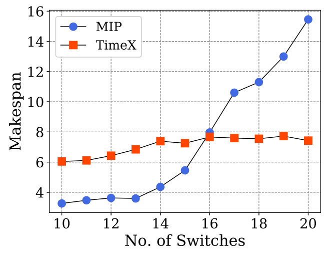
Fig. 6: Runtime evaluation, in seconds, of TimeX by comparing it to MIP. Runtime evaluation of RMS by comparing it to GRD.

of switches. We observe that RMS shows better scalability, as its runtime increases by less than 0.015 seconds, compared to the 0.033 seconds increase in GRD. Based on Fig. 6(b), RMS achieves 9.1x and 13.8x speed-up on average and in maximum, respectively, compared to GRD. Figs. 6(e) and 6(f) show that the runtime of RMS remains less than 0.08 seconds. However, GRD's runtime can be as high as 0.25 seconds. By comparing Figs. 6(e) and 6(f) with Figs. 6(c) and 6(d), we can verify that the range of runtime for TimeX and GRD is less than 0.3 seconds for the majority of problem instances.

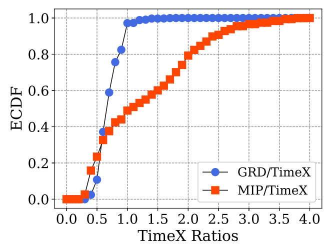
Makespan. Fig. 7(a) shows the average makespan of TimeX and MIP for the increasing number of switches from 10 to 20. When the problem size is smaller, MIP performs better by about 2 time slots. As the problem instances become more complex, the performance of MIP degrades significantly, due to the runtime limit. For problem instances with 16 switches, TimeX starts to outperform MIP. For the largest problem instances, TimeX improves the makespan by more than 8 time slots. Based on Fig. 7(a), TimeX improves the makespan by 2% and 108% on average and in maximum, respectively. Next, we study the presented theoretical bound in Appendix C. If we evaluate the bound for the different number of switches, we find out that TimeX should not increase the makespan by more than 6.1 times of the optimal value with the probability of at least 1/2. To investigate this, we plotted the empirical cumulative density function (ECDF) for the ratio of makespan between TimeX and MIP and GRD in Fig. 7(b). We observe that for 2.6% of problems the ratio between MIP and TimeX is 0.3 (*i.e.*, a factor of 3.3), and for 2.4% of problems the ratio between GRD and TimeX is 0.4 (*i.e.*, a factor of 2.5). These results show that the worst-case performance of TimeX in practice is considerably better than the theorized bound. Note that the ECDF in Fig. 7 only presents the problem instances that both GRD and TimeX have solved. Thus, it does not show the effect of GRD failures that happen up to 15% more compared to TimeX (see the *success rate* experiment later in this section for further detail). Our measurements show that each failure by GRD on average wastes about 120 milliseconds and the distance between the makespans of GRD and TimeX in the majority of cases is less than or equal to two time slots. If we use a fallback solution whose performance is similar to GRD (*e.g.*, TCAM-based work in [26]) to handle failures, in an environment where the duration of a time slot is less than or equal to 40 milliseconds, the practical performance of TimeX is better by up to 15% compared to the result of Fig. 7. The difference becomes more visible in time-stringent systems that have shorter time slots, where the wasted time due to failures translates to more wasted time slots. Figure 8(a) investigates the makespan of RMS for the increasing number of switches by comparing it with GRD. RMS consistently outperforms GRD and achieves an 18% and 36% reduction on average and in maximum. Figure 8(b) shows that MIP and GRD achieve better results than RMS in about 10% of instances. The performance of MIP and GRD is similar to RMS in about 35% and 49% of instances, respectively, which results in the jump at ratio 1. Thus, RMS provides a better solution compared to MIP and GRD in 55% and 41% of instances, respectively.

Effect of the Merge algorithm. To show the effect of the Merge mechanism that RMS employs, we compared its makespan with ORG and SWP and plotted the average result for the increasing number of switches in Fig. 9(a). On average, we observe an improvement of 4% to 10% compared to SWP and ORG, respectively. We compared the average runtime of RMS with ORG and SWP in Fig. 9(b) to study the Merge algorithm's runtime overhead. We observe that the maximum overhead is about 0.01 seconds, while the average runtime overhead is approximately 0.004 and 0.002 seconds compared to ORG and SWP, respectively. Also, we recorded the range of makespan values for RMS, ORG, and SWP for 15 to 20 switches to present a deeper comparison. We observed that the third quantiles increase by less than 0.02 seconds, displaying the efficiency of the Merge algorithm.

Success rate. Figure 10(a) shows that RMS obtains the highest success rate. At first, for smaller problem instances with fewer switches, MIP's performance is similar to RMS. However, as the size increases, the performance of MIP degrades significantly. Specifically, for larger problem instances, MIP has the worst result compared to other algorithms. In this figure, we set $\Gamma = 0$, which means that TimeX does not backtrack. Nevertheless, TimeX solves 5 to 10 percent more problems compared to GRD. However, their performance converges for instances with 16 switches or more. We observe that the success rate of MIP monotonically decreases due to its lack of scalability. However, the success rates of other algorithms increase after

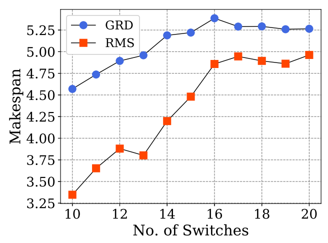


(a) Effect of number of switches on the makespan of MIP and TimeX.

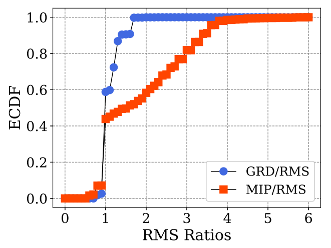


(b) ECDF of makespan ratio between MIP, GRD and TimeX.

Fig. 7: Makespan evaluation of TimeX by comparing it to MIP.

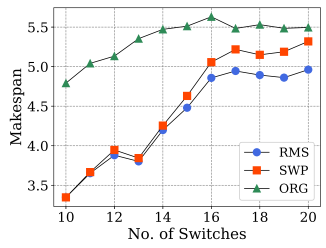


(a) Effect of number of switches on the makespan of GRD and RMS.

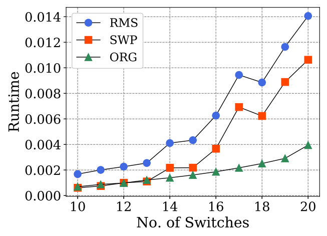


(b) ECDF of makespan ratio between MIP, GRD and RMS.

Fig. 8: Makespan evaluation of RMS by comparing it to GRD.

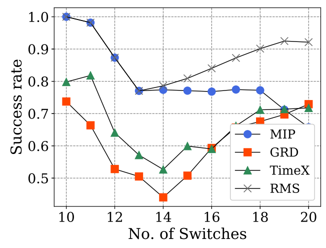


(a) Makespan improvement by RMS via the Merge algorithm.

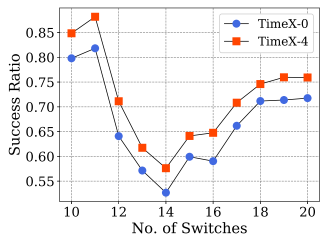


(b) Runtime overhead of the Merge algorithm.

Fig. 9: Effects of the Merge algorithm.



(a) Effect of the number of switches on the success rate of different algorithms.



(b) Effect of the parameter Γ on success rate of the TimeX algorithm.

Fig. 10: Evaluation of algorithms' success rate.

their first decline. The behavior resembles finding a path between two nodes in a graph. Finding a path is not time-consuming when the graph has few edges. On the other hand, having many edges means that there probably are several short paths between nodes that simplify the path-finding process. In intermediate cases, long paths from the source that do not lead to the destination can significantly waste the available time of the search process.

Effect of parameter Γ on TimeX. This parameter allows TimeX to perform backtracking and find a solution with a higher probability. Figure 10(b) shows that $\Gamma = 4$, compared to $\Gamma = 0$, improves the success rate by 7% on average and 10% at maximum. Moreover, our experiments with other intermediate values show that the success rate consistently improves as Γ increases. In our experiments, we observed that the effect of Γ on the runtime is negligible, which allows us to increase it without significant overhead.

TCAM. To show the effect of RMS and TimeX on TCAM usage, we compare them with the method in [23]. To simulate a data center with a 3-tier 4-pod fat-tree network, where the maximum hop between two servers is 5, we restrict this test to employ the problem instances with five switches in [44]. When RMS or TimeX fail, they use the 2-phase commit algorithm as the fallback strategy, which maximally penalizes them by doubling the TCAM entries along the path. RMS and TimeX reduced the number of extra TCAM entries used to update the path of all considered problem instances by about 42% compared to the algorithm in [23].

B. Mininet Results

Setup. The network controller is implemented in Ryu [45]. We test the algorithms with 5 to 20 flows that have random sources, destinations, and waypoints. The runtime of MIP, TimeX, and GRD is similar to simulation results and lies in the range of 0.04 to 0.11 seconds. RMS outperforms others by at least an order of magnitude. Each flow has a fluctuating behavior, where its minimum and maximum transmission rates are uniformly selected from the intervals 0.2-0.5 and 1-3 Mbps, respectively. We ensure that the initial routing of the flows in the network respects the link capacities. Then, flows increase their transmission rate by a factor of 2 over a span of 5 minutes. Whenever the utilization of a link goes above 80 percent, the controller computes a set of new paths for all the affected flows and re-configures the network.

Makespan. Figure 11 shows the average makespan of different algorithms throughout the operational lifetime of the system. We can see that the overhead of RMS and TimeX for ensuring that conflicting flows do not use the same link in the same time slot is marginal. Since the network size and number of flows are moderate, the MIP manages to obtain the shortest makespan. However, it does not consider flow conflicts and leads to transient congestion. GRD is similar to MIP and does not account for flow conflicts. TimeX uses the solution of the relaxed linear program and achieves slightly shorter makespans. However, the performance of TimeX and RMS is similar. We observe that the probability of a flow conflict during the transient period of an update rises as the number of flows increases. Consequently, TimeX and RMS have to schedule the rule-update messages across more time slots to avoid routing conflicting flows over the same links. Specifically, the update makespan is increased up to 10 percent. However, when the number of flows is low, the overhead is as low as 4 percent.

Utilization. Figure 12 shows the distribution of link utilization throughout the system's lifetime. We can see that Both MIP and GRD allow link utilization for some links in the network to go considerably above the desired 80 percent in several time slots. Note that our routing algorithm does not allow conflict-

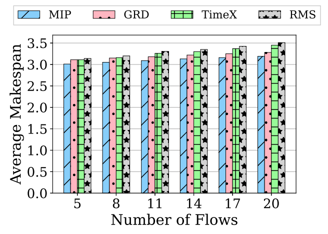


Fig. 11: Average Makespan

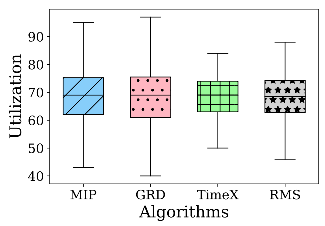


Fig. 12: Link Utilization.

ing flows to use the same link. Particularly, this undesirable situation happens because the MIP and GRD algorithms do not account for conflicting flows during the transient update interval. However, TimeX and RMS ensure that conflicting flows do not use the same link even during the transient periods of routing updates (note the upper-whiskers, which are limited to 85% utilization). We see that the maximum link utilization is about 80 percent. Since the network configurations before and after updates are identical under all algorithms, their median of utilization is similar. The only difference stems from the way that algorithms handle the network update procedure.

VIII. Conclusion

We designed TimeX and RMS algorithms to minimize the update makespan in SDNs while guaranteeing LF, WPE, and CF properties. TimeX is a randomized algorithm that exploits the solution of a linear program to minimize the makespan methodically. We also presented the performance analysis of TimeX under special considerations. To increase the scalability, we designed a second algorithm called RMS that does not rely on solving linear programs. Then, we demonstrated a property of rule-update schedules, called reversibility, that RMS employs to achieve shorter makespans. Considering multi-domain systems, service chaining, and opportunistic usage of empty TCAM entries to increase the success rate are interesting future directions for our work.

References

[1] N. Foster *et al.*, “Using deep programmability to put network owners in control,” *SIGCOMM Comput. Commun. Rev.*, vol. 50, no. 4, p. 82–88, 2020.
[2] Z. Xu *et al.*, “Experience-driven networking: A deep reinforcement learning based approach,” in *Proc. IEEE INFOCOM*, 2018, pp. 1871–1879.
[3] C.-Y. Hong *et al.*, “Achieving high utilization with software-driven WAN,” *Comput. Commun. Rev.*, vol. 43, no. 4, pp. 15–26, 2013.
[4] X. Fan *et al.*, “Real-time update of joint SFC and routing in software defined networks,” *IEEE/ACM Trans. Netw.*, pp. 1–14, 2021.
[5] A. Shukla *et al.*, “Toward consistent SDNs: A case for network state fuzzing,” *IEEE Trans. Netw. Service Manag.*, vol. 17, no. 2, pp. 668–681, 2020.
[6] G. Zhou *et al.*, “Primus: Fast and robust centralized routing for large-scale data center networks,” in *Proc. IEEE INFOCOM*, 2021, pp. 1–10.
[7] N. Christensen *et al.*, “Latte: Improving the latency of transiently consistent network update schedules,” *SIGMETRICS Perform. Eval. Rev.*, vol. 48, no. 3, p. 14–26, 2021.
[8] X. Jin *et al.*, “Dynamic scheduling of network updates,” *Comput. Commun. Rev.*, vol. 44, no. 4, pp. 539–550, 2014.
[9] B. Zhao *et al.*, “RuleTailor: Optimizing flow table updates in OpenFlow switches with rule transformations,” *IEEE Trans. Netw. Service Manag.*, vol. 16, no. 4, pp. 1581–1594, 2019.
[10] G. Li *et al.*, “Update algebra: Toward continuous, non-blocking composition of network updates in SDN,” in *Proc. IEEE INFOCOM*, 2019, pp. 1081–1089.
[11] M. Reitblatt *et al.*, “Abstractions for network update,” in *Proc. ACM SIGCOMM*, 2012, pp. 323–334.
[12] H. Habibi Gharakheili *et al.*, “iTeleScope: Softwarized network middle-box for real-time video telemetry and classification,” *IEEE Trans. Netw. Service Manag.*, vol. 16, no. 3, pp. 1071–1085, 2019.
[13] G. Zhao *et al.*, “Achieving fine-grained flow management through hybrid rule placement in SDNs,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 728–742, 2021.
[14] H. Zhou *et al.*, “Scheduling relaxed loop-free updates within tight lower bounds in sdns,” *IEEE/ACM Trans. Netw.*, vol. 28, no. 6, pp. 2503–2516, 2020.
[15] R. Mahajan and R. Wattenhofer, “On consistent updates in software defined networks,” in *ACM HotNets*, 2013, pp. 20:1–20:7.
[16] A. Ludwig *et al.*, “Transiently secure network updates,” *Perform. Eval. Rev.*, vol. 44, no. 1, pp. 273–284, 2016.
[17] J. Rasley *et al.*, “Planck: Millisecond-scale monitoring and control for commodity networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, p. 407–418, 2014.
[18] A. Ludwig *et al.*, “Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies,” in *Proc. ACM HotNets*, 2014, pp. 15:1–15:7.
[19] J. McClurg *et al.*, “Efficient synthesis of network updates,” in *Proc. ACM SIGPLAN*, 2015, pp. 196–207.
[20] S. A. Amiri *et al.*, “Transiently consistent SDN updates: Being greedy is hard,” in *Springer SIROCCO*, 2016, pp. 391–406.
[21] S. Brandt *et al.*, “On consistent migration of flows in SDNs,” in *Proc. IEEE INFOCOM*, 2016, pp. 1–9.
[22] K.-T. Foerster *et al.*, “Survey of consistent software-defined network updates,” *IEEE Commun. Surv. Tutor.*, vol. 21, no. 2, pp. 1435–1461, 2018.
[23] N. P. Katta *et al.*, “Incremental consistent updates,” in *Proc. ACM HotSDN*, 2013, pp. 49–54.
[24] M. Reitblatt, N. Foster *et al.*, “Consistent updates for software-defined networks: Change you can believe in!” in *Proc. ACM HotNets*, 2011.
[25] R. Sukapuram and G. Barua, “ProFlow: Proportional per-bidirectional-flow consistent updates,” *IEEE Trans. Netw. Service Manag.*, vol. 16, no. 2, pp. 675–689, 2019.
[26] S. Vissicchio and L. Cittadini, “FLIP the (flow) table: Fast lightweight policy-preserving SDN updates,” in *Proc. IEEE INFOCOM*, 2016, pp. 1–9.
[27] K. T. Förster *et al.*, “Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes,” in *Proc. IFIP Networking*, 2016, pp. 1–9.
[28] A. Ludwig *et al.*, “Scheduling loop-free network updates: It’s good to relax!” in *Proc. ACM PODC*, 2015, pp. 13–22.
[29] R. Gandhi *et al.*, “Catalyst: Unlocking the power of choice to speed up network updates,” in *Proc. ACM CoNEXT*, 2017, pp. 276–282.
[30] J. Zheng *et al.*, “Minimizing transient congestion during network update in data centers,” in *Proc. IEEE ICNP*, 2015, pp. 1–10.
[31] H. Xu *et al.*, “Joint route selection and update scheduling for low-latency update in SDNs,” *IEEE/ACM Trans. Netw.*, vol. 25, no. 5, pp. 3073–3087, 2017.
[32] J. Zheng *et al.*, “Congestion-minimizing network update in data centers,” *IEEE Trans. Services Comput.*, vol. PP, no. 99, pp. 1–1, 2017.
[33] A. Basta *et al.*, “Efficient loop-free rerouting of multiple SDN flows,” *IEEE/ACM Trans. Netw.*, vol. 26, no. 2, pp. 948–961, 2018.
[34] T. Qu *et al.*, “Efficient event scheduling of network update,” *IEEE Trans. Netw. Service Manag.*, vol. 17, no. 1, pp. 416–429, 2020.
[35] M. Tsai *et al.*, “Enabling efficient and consistent network update in wireless data centers,” *IEEE Trans. Netw. Service Manag.*, vol. 16, no. 2, pp. 505–520, 2019.
[36] M. Dolati *et al.*, “Consistent sdn rule update with reduced number of scheduling rounds,” in *Proc. IEEE/ACM/IFIP CNSM*, 2018, pp. 254–260.
[37] Open vSwitch, “Design Decisions In Open vSwitch,” https://docs.openvswitch.org/en/latest/topics/design/, online; accessed November 2020.
[38] S. Jain *et al.*, “B4: Experience with a globally-deployed software defined WAN,” in *Proc. ACM SIGCOMM*, 2013, pp. 3–14.
[39] L. Liberti, “Compact linearization for binary quadratic problems,” *4OR*, vol. 5, no. 3, pp. 231–245, 2007.
[40] N. Karmarkar, “A new polynomial-time algorithm for linear programming,” in *Proc. ACM STOC*, 1984, p. 302–311.
[41] V. Arora *et al.*, “A limited-backtrack greedy schema for approximation algorithms,” in *Proc. springer FSTTCS*, 1994, pp. 318–329.
[42] B. Lantz *et al.*, “A network in a laptop: Rapid prototyping for software-defined networks,” in *Proc. ACM HotNets*, 2010, pp. 19:1–19:6.

Algorithm 7: OrderCompute

```
Input  : $\mathcal{L}_f^1, \mathcal{L}_f^2, P_f(t), \boldsymbol{\mathcal{S}}$
Output: A valid assignment for $o_w^f(t)$
1  For all $w \in P_f(t)$, set $o_w^f(t)$ using ascending integers
2  $\mathcal{D}st \leftarrow P_f(t)$
3  for $w \notin P_f(t)$ and $o_w^f(t)$ is not set do
4      Q ← queue(($w$, []))
5      while True do
6          $w, p \leftarrow$ Q.dequeue()
7          if $w \in \mathcal{S}$ then
8              $v \leftarrow \ell.t$ such that $\ell \in \mathcal{L}_f^2$ and $\ell.h = w$
9          else
10             $v \leftarrow \ell.t$ such that $\ell \in \mathcal{L}_f^1$ and $\ell.h = w$
11         $p' \leftarrow p + [w]$                    // Append $w$ to $p$
12         Q.enqueue(($v$, $p'$))
13         if $v \in \mathcal{D}st$ then
14             $\forall w' \in p' : o_{w'}^f(t) \leftarrow o_v^f(t)$ and break
15         else if $v \in p'$ then
16             $\forall w' \in p' : o_{w'}^f(t) \leftarrow 1$ and break
17     $\mathcal{D}st$.add($p'$)
18 Return $\{o_w^f(t)\}$
```

[43] Gurobi solver. [Online]. Available: https://www.gurobi.com

[44] Update dataset. [Online]. Available: http://net.t-labs.tu-berlin.de/~stefan/netup.tar.gz

[45] Ryu: A component-based software defined networking framework. [Online]. Available: https://osrg.github.io/ryu/index.html

[46] R. Kleinberg, "Notes on Approximation Algorithms," www.cs.cornell.edu/courses/cs6820/2013fa/handouts/approx_algs.pdf, 2012, online; accessed July 2019.

Appendix A

We use the following rules to compute $o_w^f(t)$ for switch w in time slot t:

Case I. $\boldsymbol{w \in P_f(t)}$**:** If w is the k-th (counting from 1) switch in the current path of flow f, we set $o_w^f(t) = k$.

Case II. $\boldsymbol{w \notin P_f(t)}$ **and** $\boldsymbol{P_f(t) \cap P_f(w,t) \neq \phi}$**:** In this case, we define $o_w^f(t) = \min\{o_{w'}^f(t) | w' \in P_f(t) \cap P_f(w,t)\}$.

Case III. $\boldsymbol{w \notin P_f(t)}$ **and** $\boldsymbol{P_f(t) \cap P_f(w,t) = \phi}$**:** This happens when a subset of switches that are not in $P_f(t)$ form a loop. For these switches, we define $o_w^f(t)$ to be 1.

Algorithm 7, named OrderCompute, computes the values of $o_w^f(t)$. First, it assigns the value for switches in the current path $P_f(t)$ based on Case I. Then, it stores switches of the current path in a set called $\mathcal{D}st$ in line 2. The value for other switches is computed based on Case II or Case III and a breadth-first search algorithm in lines 3 to 17. Specifically, the algorithm finds the shortest path from each remaining switch to the nearest switch in $\mathcal{D}st$. If the path exists, $o_w^f(t)$ for all switches along the path is set to the value of the nearest switch in $\mathcal{D}st$ (see line 14 and refer to Case II). When the path does not exist, the algorithm finds a set of switches that form a loop (see Case III) and sets the value of $o_w^f(t)$ to 1 for all retrieved switches (see line 16). Note that in each iteration of the while loop in line 5 either a path or a loop is computed. Then, the value of $o_w^f(t)$ is set for all the visited switches. They would not appear in the subsequent iterations because they are added to $\mathcal{D}st$. Thus, the complexity of OrderCompute is $O(|W_f|)$.

Appendix B

In each run of the SchedulerFinder, the value of $\widetilde{x}_f^w(t)$ for one w-t pair is reduced to zero. There are at most T_f time slots and $|\mathcal{W}_f|$ switches. Thus, the time complexity of TimeX is $O(T_f|\mathcal{W}_f|T_{SF})$, where T_{SF} shows the complexity of subroutine SchedulerFinder. The algorithm performs at most Γ backtracks to fix the set of updated switches in a specific time slot. The total number of time slots is at most T_f. Thus, the time complexity of SchedulerFinder is $O(\Gamma T_f T_{SSU})$, where T_{SSU} denotes the time complexity of subroutine SwUpdate. SwUpdate calls OrderCompute that runs in $O(|\mathcal{W}_f|)$ (see Appendix A) once at the beginning. Then, for each switch w in the current path, which are at most $|\mathcal{W}_f|$, the procedure in lines 5 to 26 repeats. In each iteration, a path from $w \in P_f(t)$ to d_f is computed. Since the size of $P_f(t)$ is $O(|\mathcal{W}_f|)$, there are at most $O(2|\mathcal{W}_f|)$ edges in the initial and final paths of flow f, and priority queue insertion complexity is $O(\log|\mathcal{W}_f|)$, the total runtime of SwUpdate is $O(|W_f| + |\mathcal{W}_f|^2 \log|\mathcal{W}_f|)$. Thus, SchedulerFinder runs in $O(\Gamma T_f |\mathcal{W}_f|^2 \log|\mathcal{W}_f|)$. Thus, TimeX runs in $O(|\mathcal{F}| T_f^2 \Gamma |\mathcal{W}_f|^3 \log|\mathcal{W}_f|)$.

Appendix C

Since MCMU is NP-hard to approximate, we consider the sub-class of the problems that SwUpdate has the potential to send a rule-update message to all switches that satisfy conditions (15)-(17). Then, the probability of sending a rule-update message in a specific time slot to a switch is proportional to the fractional values $\widetilde{x}_f^w(t)$. So, we can show that, for each subset $C_i \in C$, TimeX increases the update makespan by at most a factor $\alpha = \frac{3\log 2(W-1)}{\log\log 2(W-1)}$, with the probability of at most 1/2. W is the size of the largest set $\mathcal{W}_f$ for a flow $f \in C_i$. To this end, for a given switch $w \in \mathcal{W}_f$, define independent random variables $\Delta_f^w(1), \ldots, \Delta_f^w(T_i)$ to be equal to $\frac{t}{M(i)}$ if w is updated in time slot t and equal to 0 otherwise. The probability of sending a rule-update message to switch w in time slot t is $\widetilde{x}_f^w(t)$, *i.e.*, the optimal value of the variable $x_f^w(t)$ in the relaxed problem. Therefore, the expectation of the sum of these random variables for each switch w is,

$$\mathbb{E}[\sum_{t\in\{1,\ldots,T_i\}} \Delta_f^w(t)] = \sum_{t\in\{1,\ldots,T_i\}} \Delta_f^w(t)\widetilde{x}_f^w(t) = \sum_{t\in\{1,\ldots,T_i\}} \frac{t\widetilde{x}_f^w(t)}{M(i)}, \quad (18)$$

which is at most 1 because of the constraint (13). Applying the Chernoff Bound from [46] with $N = 2(|\mathcal{W}_f| - 1)$, we obtain,

$$\mathbb{P}\{\Delta_f^w(1) + \cdots + \Delta_f^w(T_i) \geq \alpha\} < \frac{1}{2(|\mathcal{W}_f| - 1)}. \quad (19)$$

The following equality can characterize the time slot in which switch w receives the rule-update message of flow f,

$$M(i) \sum_{t\in\{1,\ldots,T_i\}} \Delta_f^w(t) = \sum_{t\in\{1,\ldots,T_i\}} t\hat{x}_f^w(t) \quad (20)$$

where, $\hat{x}_f^w(t)$ is the integer value of decision variable $x_f^w(t)$ that is determined in the process of solving MCMU. Consequently,

$$\mathbb{P}\{\sum_{t\in 1..T_i} t\hat{x}_f^w(t) \geq \alpha M(i)\} < \frac{1}{2(|\mathcal{W}_f| - 1)}. \quad (21)$$

By the Union Bound, the probability of $\sum_{t\in\{1,\ldots,T_i\}} t\hat{x}_f^w(t)$ exceeding $\alpha M(i)$ for all switches in $\mathcal{W}_f - \{d_f\}$ is at most 1/2. Equivalently, for all switches, $\sum_{t\in\{1,\ldots,T_i\}} t\hat{x}_f^w(t)$ is less than $\alpha M(i)$ with probability of at least 1/2.