Abstract—Software-Defined Networks (SDNs) introduce great flexibilities in how packet routes can be defined and changed over time, and enable a more fine-grained and adaptive traffic engineering. The recently introduced support for more accurate synchronization in SDNs further improves the degree of control an operator can have over the packets’ forwarding paths, and also allows to avoid disruptions and inconsistencies during network updates, i.e., during the rerouting of flows. However, how to optimally exploit such technology algorithmically — to efficiently schedule the update of multiple flows in such timed SDNs — while accounting for possible interference and congestion, is not well-understood today.

We in this paper initiate the study of the fundamental problem of how to reroute the updates of multiple network flows in a synchronized SDN in a congestion-free manner. We rigorously prove that the problem is NP-hard for flows of unit size and network links with unit delay. We also show that a greedy approach to update the network can delay the update significantly. Our main contribution is the first solution to this problem: Chronicle. Our approach is based on a time-extended network construction and resource dependency graph, which is implemented by Openflow 1.5 using the scheduled bundles feature. Evaluation results show that Chronicle can reduce the makespan by 63% and reduce the number of changed rules by 50% compared to state-of-the-art.

I. INTRODUCTION

The more direct, fine-grained and adaptive traffic engineering enabled by Software-Defined Networks (SDNs) is one of the key benefits of this new networking paradigm [10]. However, reaping the benefits of a more adaptive network control is still challenging in practice due to the inherent asynchrony in the communication between SDN controllers and switches, and also between the switches themselves [18]. Indeed, many potential disruptions (transient or even permanent), inconsistencies, and instabilities have been identified and studied empirically and analytically over the last years, making the SDN update problem an active area of research [13].

Network updates are not only relevant in the context of a more adaptive and fine-grained traffic engineering (typically, for minimizing network loads), but the ability to quickly and consistently reroute flows is crucial for correctness, availability, and performance more generally. For example, network update problems arise due to changes in the network’s (security) policy, upon link failures, or during maintenance work.

The recent introduction of a notion of time in SDNs and the resulting more accurate synchronization, enabled timed updates in OpenFlow [25]: updates which can be scheduled accurately in time and hence allow to mitigate, or even avoid entirely, the above problems. In particularly, it has been shown that so-called flow swaps [25], results in significantly less packet loss during updates.

While synchronized SDNs enable faster and more consistent network updates, they still pose a challenging algorithmic problem which is hardly understood today. On the one hand, it is desirable that the update, that is, the rerouting of flows is completed fast. On the other hand, it is important that the update is congestion-free, which implies that the update schedule of flows needs to be jointly optimized. Due to capacity constraints and given link latencies, flows may temporarily interfere, resulting in packet loss and hinders performance.

Our contributions: This paper initiates the study of the fundamental problem of how to schedule the updates of multiple network flows in a synchronized SDN in a congestion-free manner. We show that the problem is NP-hard for flows of unit size and network links with unit delay. We also show that a greedy approach to update the network can significantly delay the update. Our approach is based on a time-extended network construction and resource dependency graph. We implement our system in Openflow 1.5 using the scheduled bundles feature, and evaluate its feasibility and efficiency on a small-scale testbed and using large-scale simulations.

First and foremost, the scheduling of multiple flow updates raises the question of the time horizon to be considered. We use the time-extended network to capture the dynamic process of flow transmission during network updates. Based on this, we ask for accurate time schedules—specifying an update time point for each switch and flow—such that the total update time (the makespan) is minimized and congestion-freedom is ensured at any moment in time. We formulate this problem as an optimization program in the time-extended network and prove its hardness.
Our second contribution is Chronicle, a heuristic scheduling algorithm. The key idea is to first divide all network update instances into small update blocks, then build the dependency relations among blocks. Based on the constructed time-extending network, we adjust the update time and merge the common update blocks accordingly. Finally we construct a resource dependency graph among the update blocks and schedule these blocks in a time domain.

We evaluate Chronicle using both a prototype implementation and large-scale simulations. We develop a prototype using the new scheduled bundles feature of Openflow 1.5. We use OFSoftSwitch and Dpctl [5] as Openflow switches and the controller. Our evaluation results show that Chronicle can reduce the update time by 63% and reduce the number of changed rules by 50% compared to state-of-the-art. At the same time, Chronicle can avoid transient congestion, save flow table space and provide a near optimal solution.

**Novelty:** The need for fast and consistent network update mechanisms has been articulated well in literature in various contexts, such as for security [23], performance [17], and dependability [21], [22], [33] reasons. Most existing literature focuses on “interactive” update mechanisms, involving the controller which monitors the progress of the update (e.g., using acknowledgements) before deciding to start the next stage of the update. In particular, existing approaches can be roughly classified into (1) two-phase protocols [7], [16], [22], [30] in which the controller first installs the new rules before tagging packets with the new path at the ingress port, ensuring that each packet either takes the old or the new route, but never a combination of both; (2) node-ordering protocols [18], [12] where the controller updates (subsets of) switches one-by-one, such that transient inconsistencies are avoided (without the need for tagging). While solutions between the two worlds are emerging [31], these approaches have in common that they need to rely on interactions with the controller to implement synchronization.

Our approach is enabled by technologies such as Time4 [25], [27] which allow us to synchronize network updates using accurate time [26]. To the best of our knowledge, the only algorithmic study of the network update problem in timed SDNs is by Zheng et al. [32], which however focuses on a single flow. While constituting an interesting first step, we anticipate many situations in which multiple flows need to be updated simultaneously, e.g., upon a policy change or link failure, or even a new traffic engineering optimization. Scheduling multiple flow updates simultaneously however is significantly harder since different flows can interfere in complex (potentially combinatorial) ways at different links.

II. BACKGROUND AND MOTIVATION

We consider a network where a controller updates the forwarding rules at the switches whenever a route changes. Fig. 1(a) illustrates a simple example: there are five switches \(v_1, \ldots, v_5\) in the network. The link capacity of \(\langle v_2, v_5 \rangle\) is assumed to be two units and the rest is one unit. The propagation delay of link \(\langle v_3, v_4 \rangle\) is assumed to be three time units and the rest are one time unit. That is, if one unit of flow leaves switch \(u\) at time \(t\) on the link \(\langle u, v \rangle\), one unit of flow arrives at switch \(v\) at time \(t + \sigma_{u,v}\), where \(\sigma_{u,v}\) is the propagation delay between \(u\) and \(v\). We use the notion of dynamic flow to represent the propagation of packets of a flow in a time domain [14]. In our example, the demand of two “dynamic flows” colored as red and green are both one unit, which are both routed from the source \(v_1\) to the destination \(v_5\). The initial routings are depicted as two solid red and green lines and the final routings are depicted as two dashed red and green lines. With dynamic flows, the utilization of a link varies over time. As discussed before, prior work on the network update problem usually relies on one of two fundamental update techniques: two-phase update protocols and node ordering protocols.

**Two-phase update protocols:** In the first phase, new rules—whose matching fields use the new version tag that corresponds to the second stage—are added. During this phase, flows are still forwarded according to existing rules as packets are still stamped with the old version tag of the first stage. Once the update is done for all switches, the protocol enters the second phase, when we stamp every incoming packet with the new version tag. At this point, the new rules become functional, and old rules are removed by the controller. Reitblatt et al. [30] initiated this line of research by introducing a two-phase commit protocol that preserves consistency when changing between two different routing configurations. Based on this, SWAN [16] and zUpdate [22] try to find a congestion-free two-phase update plan. SWAN shows that if each link has a certain slack capacity, there always exists a congestion-free update sequence. This condition is too strong to always hold in practice. Furthermore, Brandt et al. [7] analyze the condition that a congestion-free update sequence exists, with additional complexity studies in [11]. As the update plan is not unique, Dionysus [18] seeks to determine the fastest update sequence according to different runtime conditions of switches.

A two-phase update procedure in our example of Fig. 2(b) is: in the first phase, the routes for red and green flows corresponding to new version tag are updated; in the second phase, we change the version tag at the source switch \(v_1\) and swap the red and green flows into their final path. Fig. 2(c) shows a possible asynchronous update case, where the time difference between two phases is assumed to be one time unit. We can observe that the congestion happens at the link \(\langle v_4(t_4), v_5(t_5) \rangle\) since one unit capacity of link \(\langle v_4, v_5 \rangle\) cannot accommodate two flows at the same time.

**Node ordering protocols:** At each round, the controller waits until all the switches have completed their updates, and only then invokes the next round. Ludwig et al. [24] aim to minimize the number of sequential controller interactions when transitioning from the initial to the final update stage. The authors prove that finding a shortest node ordering sequence that avoids forwarding loops is NP-hard. Furthermore, they introduce a notion of relaxed loop-freedom, which provides an interesting consistency-runtime tradeoff. Another work by Ludwig et al. [23] considers secure network updates in the presence of middleboxes [29]. The authors try to find a node ordering sequence that preserves a specific security policy.
A possible node ordering sequence in our example of Fig. 2(b) is: Fig. 1(c1) → (c2). In the first round, $v_2$ (the route for red flow), $v_3$ (the route for green flow) and $v_4$ (the route for green flow) are updated asynchronously. Then $v_1$ (the route for both red and green flows) and $v_2$ (the route for green flow) are updated in the second round. Due to the asynchronous nature of the data plane, the new route for $v_2$ (green flow) may become functional earlier than that for $v_1$ (red and green flows) in the second round as shown in Fig. 1(c2). At this point, the red and green flows are routed through the paths $\langle v_1, v_3, v_4, v_5 \rangle$ and $\langle v_1, v_2, v_4, v_5 \rangle$ respectively. Here the red and green flows together would result in a transient congestion on the link $(v_4, v_5)$ as the sum of flow demand is two units, which is beyond the one unit link capacity (Fig. 2(b)).

**Timed update protocols**: Mizrahi et al. [26] design a practical approach to implement accurately scheduled network updates in TCAM with the order of microseconds. Based on this, they propose a time synchronization protocol between controller and data plane, which uses accurate timing to trigger network updates [25], [27], [28]. Specifically, the data plane is firstly synchronized by Network Time Protocol. Then the controller sends the update commands to the switch and these update commands are stored into a temporary staging area without taking effect. Finally the update commands will be applied to the switch at a specific time point. The current OpenFlow protocol [3] has already provided APIs for time-based network updates. Though the idea of timed update has surfaced in the literature recently, the only algorithmic study of the timed update is by Zheng et al. [32], which however focuses on a single flow. To this end, Zheng et al. prove that minimizing the makespan is NP-hard, and design a heuristic algorithm to perform network update for a single flow. The novelty of our work lies in the first study and comprehensive exploration of the design of timed scheduling algorithms for multiple flows. We also extend the hardness results of Zheng et al. [32] to the case of multiple flows, in particular we show that it is NP-hard even for flows of unit size and network links with unit delay.

A timed update schedule (Fig. 1(d1) → (d2) → (d3) → (d4) → (d5) → (d6)) can effectively solve our problem. Firstly, the route of $v_1$ for both red and green flows are updated at $t_0$. Then, the route of $v_3$ for the green flow is updated at $t_1$. Finally the routes of $v_2$ and $v_4$ for green flow are updated simultaneously at $t_3$. The congestion-free condition is ensured at any moment in time as shown in the time-extended network of Fig. 2(d). This timed schedule can be acceptable in practice because the flow table rules can be updated accurately on the order of one microsecond [26]. In addition, the controller can send all the update commands at a time and the update behavior for each switch is triggered by a pre-defined time instant, which can significantly decrease the time overhead resulting from wait-involve mechanism of node ordering protocols. Also we only modify the action in the flow table during the update process, where we do not require additional flow table space headroom and overcome the drawback of two-phase update protocols.

Lastly, we note that there can also be other objectives for consistent updates, such as e.g. maintaining service availability for inter-domain routes, which is investigated by SDN-LIRU [34]. We refer to [13] for a general overview.

**III. An Optimization Framework**

We introduce our optimization framework for the minimum update time problem for multiple flows in this section.

**A. Dynamic Flow Model and Problem Formulation**

Before formulating the problem, we first present our network model. A network is a directed graph $G = (V, E)$, where $V$ is the set of switches and $E$ the set of links with capacities $C_{u,v}$ and transmission times $\tau_{u,v}$ for each link $(u, v) \in E$. For each flow $f$, the network contains two paths: $p^f_{\text{init}}$ and $p^f_{\text{fin}}$. The former is the old routing path which is depicted as a solid line in our example and the latter is the new routing path depicted as a dashed line. We use different colors to distinguish different flows. Both of $p^f_{\text{init}}$ and $p^f_{\text{fin}}$ have a common source $v^+$ and destination $v^-$. For convenience, we summarize important notations in Table I. Let us introduce three related notations first.
The set of dynamic flow

The acyclic directed network graph

The set of time point.

(a) The time-extended network model that represents whether there exists a congestion-free update schedule for the red and green flows within six time steps, where \( T = \{t_0, t_1, t_2, t_3, t_4, t_5\} \). The value of link capacity and propagation delay are the same as the setting in Fig. 1(a).

∀

(b) A possible update sequence in the time-extended network resulting from node ordering protocols as shown in Fig. 1(c1) \(\rightarrow\) (c2). We assume the time difference between the first round and the second round is one time step. The congestion happens at the link \((v_4(t_2), v_5(t_4))\) and \((v_4(t_4), v_5(t_5))\).

(c) A possible update sequence in the time-extended network resulting from two-phase update protocols. We assume the time difference between the first phase and the second phase is one time step. We can observe that the congestion happens at the link \((v_4(t_4), v_5(t_5))\).

(d) We first update switch \(v_1\) (red and green flows) and \(v_2\) (red flow) at \(t_0\). Then we update \(v_3\) (green flow) at \(t_1\). Finally we update \(v_5\) (green flow) and \(v_4\) (green flow) at \(t_3\). The whole procedure is congestion-free at any moment in time. For simplicity, we do not draw the links once the update is done.

Fig. 2. Illustration of two-phase update protocols, node ordering protocols and our timed schedule shown in the time-extended network.

| \(F\) | The set of dynamic flow \(f\) |
| \(V\) | The set of switches \(v\) |
| \(E\) | The set of links \(\langle u, v \rangle\) |
| \(G\) | The acyclic directed network graph \(G = (V, E)\) |
| \(t_i\) | The time point, \(t_{i+1} > t_i\) |
| \(T\) | The set of time point. \(T = \{t_0, t_1, \ldots, t_n\}\) |
| \(E^\tau\) | The set of flows in the time-extended network |
| \(V^\tau\) | The set of switches \(v(t)\), where \(v \in V\) and \(t \in T\) |
| \(E^\tau\) | The set of links \(\langle u(t_i), v(t_i) \rangle\) |
| \(G^\tau\) | The time-extended network \(G^\tau = (V^\tau, E^\tau)\) |
| \(C_{u,v}\) | The capacity of link \(\langle u, v \rangle\) |
| \(p_f^{\text{init}}\) | The initial path for the dynamic flow \(f\) |
| \(p_f^{\text{final}}\) | The final path for the dynamic flow \(f\) |
| \(d_f\) | The demand of the dynamic flow \(f\) |
| \(n\) | The number of the switches. \(n = |V|\) |
| \(\sigma_{u,v}\) | The transmission delay for the link \(\langle u, v \rangle\). |

| TABLE I | KEY NOTATIONS IN THIS PAPER. |

Definition III.1. Dynamic flow [14]: A dynamic flow on \(G\) is a function \(f : E \times T \rightarrow Z_+\) (\(Z_+\) represents the set of positive integers) that satisfies the following conditions for \(\forall v \in V\) and \(\forall t \in T\).

\[
\sum_{(u,v) \in E^+(v), t-\sigma_{u,v} \geq 0} x_{u,v}(t-\sigma_{u,v}) - \sum_{(u,v) \in E^-(v)} x_{u,v}(t) = \begin{cases} -d_f & v = v^- , \forall t \in T \\ 0 & \forall v \in V - \{v^-, v^+\} , \forall t \in T \\ d_f & v = v^+ , \forall t \in T \end{cases}
\]

(1)

Definition III.2. Congestion-free condition: The congestion-free condition holds if and only if inequations (2) always hold for \(\forall t \in T\) throughout the update process.

\[
0 \leq x_{u,v}(t) \leq C_{u,v}, \forall (u,v) \in E, \forall t \in T \tag{2}
\]

The conservation condition (1) indicates that if one unit of dynamic flow leaves switch \(u\) at \(t - \sigma_{u,v}\) on link \(\langle u, v \rangle\), one unit of flow arrives \(v\) at \(t\). Here \(E^+(v)\) and \(E^-(v)\) represent the set of links incoming and outgoing to the switch \(v\), respectively. The notation \(d_f\) is the flow demand, which is a positive integer. The set \(T\) is measured in discrete steps, where \(T = \{t_0, t_1, \ldots, t_n\}\). \(x_{u,v}(t)\) characterizes the load on the link \(\langle u, v \rangle\) at \(t\).

The conservation condition (1) indicates that one unit of dynamic flow leaves switch \(u\) at \(t - \sigma_{u,v}\) on link \(\langle u, v \rangle\), one unit of flow arrives \(v\) at \(t\). Here \(E^+(v)\) and \(E^-(v)\) represent the set of links incoming and outgoing to the switch \(v\), respectively. The notation \(d_f\) is the flow demand, which is a positive integer. The set \(T\) is measured in discrete steps, where \(T = \{t_0, t_1, \ldots, t_n\}\). \(x_{u,v}(t)\) characterizes the load on the link \(\langle u, v \rangle\) at \(t\).

The conservation condition (1) indicates that one unit of dynamic flow leaves switch \(u\) at \(t - \sigma_{u,v}\) on link \(\langle u, v \rangle\), one unit of flow arrives \(v\) at \(t\). Here \(E^+(v)\) and \(E^-(v)\) represent the set of links incoming and outgoing to the switch \(v\), respectively. The notation \(d_f\) is the flow demand, which is a positive integer. The set \(T\) is measured in discrete steps, where \(T = \{t_0, t_1, \ldots, t_n\}\). \(x_{u,v}(t)\) characterizes the load on the link \(\langle u, v \rangle\) at \(t\).

The conservation condition (1) indicates that one unit of dynamic flow leaves switch \(u\) at \(t - \sigma_{u,v}\) on link \(\langle u, v \rangle\), one unit of flow arrives \(v\) at \(t\). Here \(E^+(v)\) and \(E^-(v)\) represent the set of links incoming and outgoing to the switch \(v\), respectively. The notation \(d_f\) is the flow demand, which is a positive integer. The set \(T\) is measured in discrete steps, where \(T = \{t_0, t_1, \ldots, t_n\}\). \(x_{u,v}(t)\) characterizes the load on the link \(\langle u, v \rangle\) at \(t\).

The conservation condition (1) indicates that one unit of dynamic flow leaves switch \(u\) at \(t - \sigma_{u,v}\) on link \(\langle u, v \rangle\), one unit of flow arrives \(v\) at \(t\). Here \(E^+(v)\) and \(E^-(v)\) represent the set of links incoming and outgoing to the switch \(v\), respectively. The notation \(d_f\) is the flow demand, which is a positive integer. The set \(T\) is measured in discrete steps, where \(T = \{t_0, t_1, \ldots, t_n\}\). \(x_{u,v}(t)\) characterizes the load on the link \(\langle u, v \rangle\) at \(t\).
extended network example of Fig. 1(a), where $t_{-3}, \ldots, t_0$ represent the past time steps, $t_0$ represents the current time step, $t_1, t_2, \ldots$ represent the future time steps. The green flow on the link $\langle v_1(t_0), v_2(t_1) \rangle$ starts at current time step $t_0$, while the green flow on the link $\langle v_2(t_0), v_3(t_1) \rangle$ and the red flow on the link $\langle v_3(t_0), v_4(t_3) \rangle$ both start at past time step $t_{-1}$. The red solid line between $v_3$ and $v_4$ strides over three time steps as its link transmission delay we assumed in Fig. 1(a) are three time units. The rest only stride over one time step as its link transmission delay is one time unit. We can only update the switches in the current and future time steps and cannot update them in the past steps. The reason why we illustrate the past time steps there is that we require to check the congestion-free condition defined in (III.2). In Fig. 2(a), the red flow starting at past time step $t_{-4}$ will occupy the link bandwidth between $v_3(t_0)$ and $v_2(t_1)$, which makes a difference to the updates at current time step $t_0$.

Based on the above model and definition, we formulate the Minimum Update Time Problem for Multiple Flows (MUTP-MF) as an integer linear program in the time-extended network, where the initial (solid lines) and final (dashed lines) routing paths for each flow $f$ are given. We seek to find an optimal timed update sequence so as to minimize the total update steps, such that the congestion-free condition holds at any moment in time. Before formulating the problem, we first describe each constraint in detail.

### The congestion-free constraint:

$$\sum_{f \in F} y_{u(t_i), v(t_j)}^f \leq C_{u(t_i), v(t_j)}, \quad \forall (u(t_i), v(t_j)) \in E^T,$$

(3a)

The LHS of constraint (3a) characterizes the load of total flows at link $(u(t_i), v(t_j))$, which must be less than or equal to its capacity in order to meet the congestion-free condition defined in (III.2).

### The zero-one decision variables:

The zero-one integer variable $x_{u(t_i)}^f$ equals one when the routing configuration of switch $u$ for flow $f$ is updated at $t_i$ in the time-extended network, and equals zero otherwise.

$$x_{u(t_i)}^f \in \{0, 1\}, \quad \forall f \in F, \forall u(t_i) \in V^T \setminus \{t_n\},$$

(3b)

$$x_{u(t_n)}^f = 1, \quad \forall f \in F, \forall u(t_n) \in V^T.$$

(3c)

This optimization variable determines that which switch should be updated at which time point. The optimization variables $x_{u(t_i)}^f$ ($t_i \in \{T \setminus \{t_n\}\}$) need to be determined, while the variable $x_{u(t_n)}^f$ ($t_n$ is the last time step in set $T$) is known to be one as formulated in constraint (3c) since all updates should be complete before the last time step $t_n$.

$$y_{u(t_i), v(t_j)}^f = 1 - x_{u(t_i)}^f, \quad \forall f \in F, \langle u(t_i), v(t_j) \rangle \in p_{f_{init}}^f.$$  

(3d)

The zero-one integer variable $y_{u(t_i), v(t_j)}^f$ in constraint (3d) indicates whether the flow $f$ is routed through the link $\langle u(t_i), v(t_j) \rangle$ belonging to the initial path $p_{f_{init}}^f$. Obviously, it equals zero when the switch $u$ is updated at $t_i$, and equals one otherwise.

### On the contrary, the zero-one integer variable $y_{u(t_i), v(t_j)}^f$ in constraint (3e) indicates whether the flow $f$ is routed through the link $\langle u(t_i), v(t_j) \rangle$ belonging to the final path $p_{f_{fin}}^f$. It equals one when the switch $u$ is updated at $t_i$, and equals zero otherwise.

$$x_{u(t_i)}^f \geq x_{u(t_j)}^f, \quad \forall f \in F, t_i \geq t_j,$$

(3f)

The constraint (3f) captures the fact that the routing configuration at a specific switch for the flow $f$ remains unchanged once the update is complete. That is to say, we can only update the route from the initial to final path, not the reverse. For example in Fig. 2(d), once the route of switch $v_1$ for red flow is updated at $t_0$ ($x_{u(t_0)}^f = 1$), it will stay the same at the next time steps ($x_{u(t_1)}^f = 1, x_{u(t_2)}^f = 1, \ldots$).

#### Problem formulation of MUTP-MF:

The formulation of MUTP-MF is shown in (3). The objective is to minimize the number of elements in set $T$, i.e., the time steps during the update.

$$\text{minimize } |T|$$

subject to  

(3a), (3b), (3c), (3d), (3e), (3f).

At the beginning, the element of set $T$ is $t_0$. We iteratively add one time step $t_i$ into the set $T$ each time until we find a feasible solution or the number of elements in $T$ reaches a pre-defined threshold. The upper bound analysis of the number of elements in $T$ will be discussed in Theorem III.3. Note that it’s trivial to infer the transmission delay in SDNs. To measure the transmission delay between two switches, the controller can firstly generate the test packets via OFPT_PACKET_IN message to the source switch. These test packets will be stamped with a hitting time and forwarded by the pre-defined rules. Once the packets arrive at the destination switch, they will be stamped with a new hitting time and sent back to the controller via OFPT_PACKET_OUT message. Finally, the controller can infer the link transmission delay by comparing two hitting times. In addition, the switch’s processing delay for updating a forwarding rule in TCAM [4] can influence the accuracy of our model. However, recent work [6] shows that the update time can be predictable and a constant. This suggests that we can subtract a corresponding time offset from the outputs of our model, indicating that the time point triggering the update should be earlier than the resulting update time calculated from our model.

#### B. Theoretical Analysis

We establish the hardness of our problem MUTP-MF below. We start with Theorem III.1, where we show that deciding the feasibility in general is NP-hard, followed by Theorem III.2, where we prove that even for the special case of unit size flows/delays, optimization is NP-hard. Afterwards, Corollary III.1 provides insights why a simple greedy approach is not practicable, whereas Theorem III.3 gives bounds on the complexity of the time-extended network.
The feasibility of MUTP-MF is NP-hard.

Proof: Consider a special case of MUTP-MF as shown in Fig. 3. The capacity of all links is $C$ units, and the delay of all links is one unit. There are $k$ green flows each with demand $d_i$ and $\sum_{i \in \{1, 2, \ldots, k\}} d_i = C$. They are routed through the initial path $\langle s_1, v, w, t \rangle$ and will be moved into their final path $\langle s_1, u, t \rangle$. The red flow with demand $\frac{C}{2}$ is routed through the initial path $\langle s_2, u, t \rangle$ and will be moved into the final path $\langle s_2, w, t \rangle$. The objective is to assign a update time point for $k + 1$ flows such that the congestion-free condition holds at any moment in time. The only way to do this is to firstly update the green flows with total demand $\frac{C}{2}$ at $t_0$ and keep half of the link capacity at $\langle w, t \rangle$ vacant. Then we update the red flow and the rest of the green flows simultaneously at $t_1$, where the time difference between $t_1$ and $t_0$ is one unit time.

We construct a polynomial reduction from the partition problem [8] to it. Consider a partition instance consisting of $k$ items, each with a value $a_i$ and $\sum_{i \in \{1, 2, \ldots, k\}} a_i = C$. Each item $i$ corresponds to one of $k$ green flows in the example of Fig. 3, where $a_i = d_i$, $i \in \{1, 2, \ldots, k\}$. Therefore, any feasible partition of the items corresponds to the updates of $k$ green flows in two time steps, and vice versa. The routing update in the first time step forms one set of the partition, and that in the second time step forms the other.

![Fig. 3. Topology used for the reduction from Partition to MUTP-MF.](image)

The hardness of Theorem III.1 relies on different flow sizes, but even for unit size flows, MUTP-MF is NP-hard as well.

Theorem III.2. MUTP-MF is NP-hard, even for flows and delays of unit size.

Proof: Our reduction from 3-SAT [19] will feature four gadgets, called blocking-gadget (Fig. 4(a)), delay-gadget (Fig. 4(b)), variable-gadget (Fig. 4(c)) and clause-gadget (Fig. 4(d)). The demand of all flows and the delay of all links will be of unit size in this proof.

A blocking-gadget consists of a flow where the only update is performed at the source, with the old and new path being link-disjoint. By adjusting the length of the old path, we can block one unit of capacity on a link for any desired time. As such, for the other (gadget-) constructions, we can assume any capacity restrictions for new paths, as the new paths will become eventually feasible once the blocking flows leave.

A variable-gadget consists of two flows (truth assignments) that share a path to their destination on the old path of capacity two, but the new paths are split into a “true” and a “false” path, merging into a joint path to their destination, all of capacity one. As such, only either the true or the false flow can update for now, as they collide on the joint path of capacity one.

A clause-gadget consists of three flows (literals) that share an old path to their destination of capacity three, but on the new path, they first share a link of capacity one, and then split their paths, each traversing the corresponding true or false path of their variable-gadget for one link, then reaching their destination. By making the true and false paths sufficiently long enough in the variable-gadget, the paths of literal-flows from different clause-gadgets will not overlap.

Observe that only one literal-flow of each clause can update, but if no flow from a variable-gadget updates, every clause-gadget can update one literal-flow without eventual congestion. To prevent this, we introduce the delay-gadget with one flow, where the old path has a length of three and the new path has a length of two, but they share only the last link. Updating the flow introduces twice the utilization of the last link for one time unit, after one time unit. We add a delay-gadget each to the end of the old paths of the variable- and clause-gadgets, only sharing the last link, increasing that link’s capacity by one. As the old paths are fully utilized, the delay-gadget cannot update until a flow from the respective gadgets updates.

Hence, all delay-gadgets can’t update for now, unless one literal-flow from each clause and one flow from each variable updates. However, finding such an update is equivalent to solving the corresponding 3-SAT instance. Let $t$ be the earliest time when the last delay-gadget can update in case the 3-SAT instance is satisfiable and the blocking-gadgets have infinite path lengths. By adjusting the blocking-gadgets appropriately to “free” their blocked capacity for time $t$, it is NP-hard to decide if all nodes can update by time $t$.

Corollary III.1. For unit size flows and delays, computing the maximum number of updates at each time step is NP-hard. Furthermore, a maximum greedy update can increase the update sequence length by a factor of $\Omega(|V|)$.

Proof: We adapt the proof of Theorem III.2. Observe that by adjusting the lengths of the old paths in the blocking gadget, even approximating the time of the last update is NP-hard: in particular, by reducing the blocking-gadgets to a constant number (that traverses multiple blocking links), their path lengths can be polynomially increased, i.e., for any fixed $\varepsilon > 1$, a $O(|V|^{1-\varepsilon})$-approximation is NP-hard. At the cost of approximation bounds, the number of flows can also be reduced to a constant number, by stitching their paths together, since every link only “hosts” a constant number of flows.

Furthermore, the above proof construction only needs a slight modification to make maximizing a greedy update NP-hard, i.e., maximizing the number of updates at once. We add a large number of modified delay-gadgets to the old paths of the variable-gadgets, s.t. they can only be updated right away if their corresponding variable-gadget also updates one of its flows right away. As such, a maximum update at time zero involves updating one flow from each variable-gadget, and then a maximum number of literal-flows. However, computing the maximum number for those is NP-hard.

Even worse, a single greedy update (even of maximum size) can arbitrarily delay the last update. Consider a topology of successively connected delay-gadgets (just consisting of an old path of length two and a new path of length one), with a single link $e$ at the end, with all links having a capacity of two, traversed by a single flow of unit size. Updating them all at once creates, after one time unit, an utilization of two on
for a time equivalent to the number of delay-gadgets. We now add in parallel a second flow traversing a new disjoint delay-gadget, followed by also traversing $e$. If we update the second flow first, wait one time unit, and then update all others simultaneously, we are done. However, by picking a maximum update first, consisting of all updates from the delay-gadgets for the first flow, the second flow has to wait a linear time. Observe that both above considerations also hold when asking for the maximum number of updates in a fixed timespan.

As such, we need a more intricate heuristic than just greedily “updating as much as we can”, which will be discussed in Sec. IV.

**Theorem III.3.** The maximum time steps in set $T$ are bounded by

$$|T| \leq \sum_{f \in F} \sum_{v \in \mathcal{F}_{init}} \sum_{p \in e, v \in B} \arg\max_{p \in e, v \in B} \phi(p)$$

where $p_{init}$ and $p_{fin}$ represent initial and final path for flow $f$, function $\phi(\cdot)$ refers to the sum of link transmission delay.

**Proof:** The switches in the time-extended network have to wait some time steps in order to ensure that the congestion-free condition holds at any moment in time. We denote by $t_{v,f}$ the waiting time steps for switch $v$ and flow $f$ in the time-extended network and thus we have

$$|T| \leq \sum_{f \in F} \sum_{v \in B} t_{v,f} \leq \sum_{f \in F} \left( \sum_{v \in A} t_{v,f} + \sum_{v \in (B \setminus A)} t_{v,f} \right)$$

(4)

where $A = \{v|v \in p_{init} \cap p_{fin}\}$ that represents the set of switches both in the initial path $p_{init}$ and final path $p_{fin}$, and $B = \{v|v \in p_{init} \cup p_{fin}\}$ that represents the set of switches either in the initial path $p_{init}$ or final path $p_{fin}$. Obviously, each update in set $\{B \setminus A\}$ does not need to wait and we obtain $\sum_{v \in (B \setminus A)} t_{v,f} = 0, \forall f \in F$. Combining inequation (4), we have the following.

$$|T| \leq \sum_{f \in F} \sum_{v \in A} t_{v,f} \leq \sum_{f \in F} \sum_{v \in A} \arg\max_{p \in e, v \in B} \phi(p)$$

where the function $\phi(p)$ refers to the sum of link transmission delay on the path $p$. The notation $p$ represents a mixed path traveling through the switches either in the initial path $p_{init}$ or final path $p_{fin}$.

**IV. THE CHRONICLE ALGORITHM**

In this section we design a scheduling algorithm to find a feasible update sequence in polynomial time. We firstly explain the high level working of our algorithm. To increase the flexibility, we divide the network update instance into small update blocks that can be scheduled individually in the time-extended network. Fig. 5 shows an example of update blocks for the update instance in Fig. 1(b). The green flow has two blocks $\{(v_1, v_2), (v_1, v_3, v_2)\}$ and $\{(v_2, v_3), (v_2, v_4, v_5)\}$, while the red flow has only one block $\{(v_1, v_3, v_4, v_5), (v_1, v_2, v_5)\}$. The block concept introduces more convenience as we only need to update the rules in the first switch of a block. The rule update operation for the intermediate switches is trivial, and we can update all of them at initial time $t_0$. Taking Fig. 5 as an example, the rule update operation for the red flow in switch $v_2$ (Fig. 5(a)), that of the green flow in switch $v_3$ (Fig. 5(b)) and that of the green flow in switch $v_4$ (Fig. 5(c), all belong to this case and can be updated at initial time $t_0$. Next we map these blocks into the time-extended network (Fig. 6) and establish the dependency relations among them, in which one’s initial path and the other’s final path have common links. Since the initial path of the update block for the red flow starting at $t_{-3}$ has a common link $\langle v_1, v_5 \rangle$ (between $t_1$ and $t_2$) with the final path of the second update block for the green flow, we add one more update block starting at $t_{-3}$ (Fig. 6(a)) in the time-extended network. Note that the difference between the update block in Fig. 6(a) and Fig. 6(b) is just the starting time of switch $v_1$. They can be merged into one update block which will be discussed soon. In Fig. 6, the update of block

![Fig. 4. (a) Blocking-gadget. The link $(v, t)$ is blocked by one unit of traffic until $k$ time units when $s$ is updated at $t_0$. (b) Delay-gadget. The flow can only be updated when there is additional free capacity at link $(w, t)$. (c) Variable-gadget. The capacity of link $(p, q)$ is two units and the rest is one unit. Due to the capacity of link $(v, w)$ being one unit, we cannot update $s_{x_1}$ and $s_{\neg x_1}$ at the same time. (d) Clause-gadget for $(x_1 \lor x_2 \lor \neg x_3)$. As the capacity of link $(p, q)$ is one unit, only one of the three flows can be updated. However, this flow needs free capacity in its variable-gadget path.

![Fig. 5. The illustration of update blocks in Fig. 1(b): (a) Update block for red flow (b) the first update block for green flow (c) the second update block for green flow.](image-url)
a) should be earlier than that of block (d) as one unit capacity of link \( \langle u_4, v_5 \rangle \) cannot accommodate red and green flows simultaneously. Since block (a) starts at the past time point \( t_{-3} \), we add the offset of three time units for each update block in order to ensure that all the blocks start at current or future time steps. After the adjustment process is done, we merge the common blocks (update blocks shown in Fig. 6(a) and Fig. 6(b)) and construct the resource dependency graph as shown in Fig. 7. Based on this graph, we can detect the deadlocks (dependency cycles) and output a feasible update schedule.

Before describing the concrete algorithms, let us introduce three related notations first.

**Definition IV.1. Update block:** A update block \( o_f^i(t_i) \) for flow \( f \) contains two edge disjoint paths \( p_1 \) and \( p_2 \) starting and ending at the common node \( u \) and \( v \), where \( p_1 \cap p_2 = \{u, v\} \), \( p_1 \in p_f^{init} \), \( p_2 \in p_f^{fin} \).

Combining the motivating example in Fig. 1(b), four update blocks (Fig. 6) in the time-extended network are (a) \( o_{f_1}^i(t_{-3}) \), (b) \( o_{f_1}^i(t_0) \), (c) \( o_{f_2}^i(t_0) \), and (d) \( o_{f_2}^i(t_0) \). Each update block starts and ends at a common switch in the initial and final path. The \( \rightarrow \) operator captures the update order between two blocks. For example, the notation \( o_1 \rightarrow o_2 \) represents that the update time of block \( o_1 \) should not be later than that of block \( o_2 \). Otherwise, the congestion-free condition will be violated.

**Definition IV.2. Resource dependency graph:** A resource dependency graph captures the dependent relation between the block \( o_f^i(t_i) \) and the link \( \langle u, v \rangle \).

Fig. 7 shows a resource dependency graph example corresponding to the update blocks in Fig. 6. There are two types of rectangles in the graph — the link rectangle and the update block rectangle. The number in the link rectangle indicates the residual capacity \( C'_{u,v} \) on the link \( \langle u, v \rangle \) at the current time step, while that in the update block rectangle represents the flow demand. For a specific update block \( o_f^i(t_i) \), the incoming edges come from the links in the initial path, while the outgoing edges point to the links in the final path. The construction procedure is shown in Algorithm 1.

![Algorithm 1: Constructing the resource dependency graph](image)

**Algorithm 1: Constructing the resource dependency graph**

**Input:** The set of all update blocks \( O \); the initial path \( p_f^{init} \) and the final path \( p_f^{fin} \) for each flow \( f \in F \).

**Output:** The resource dependency graph \( G_o \).

1: \( G_0 = \emptyset, C'_{u,v} = 0 \)
2: for each \( o_f^i(t_i) \in O \) do
3: for each \( \langle u, v \rangle \in p_f^{init} \) do
4: \( G_0 = G_0 \cup \{\langle u, v \rangle \rightarrow o_f^i(t_i)\} \)
5: \( C'_{u,v} = C_{u,v} - d_f^i \)
6: for each \( \langle u, v \rangle \in p_f^{fin} \) do
7: \( G_0 = G_0 \cup \{O_f^i(t_i) \rightarrow \langle u, v \rangle\} \)

![Fig. 8. A deadlock example.](image)

**Definition IV.3. Deadlock:** A deadlock indicates that we cannot find a feasible update schedule in the network.

A deadlock forms if two conditions \( \sigma_{s_1,u} > \sigma_{s_2,u} \) and \( \sigma_{s_1,w} > \sigma_{s_2,w} \) hold at the same time shown in Fig. 8. On one hand, the condition \( \sigma_{s_1,u} > \sigma_{s_2,u} \) indicates that the update time of \( s_1 \) should be earlier than that of \( s_2 \) to avoid congestion at the path \( \langle w, z, t \rangle \). On the other hand, the condition \( \sigma_{s_1,w} > \sigma_{s_2,w} \) indicates that the update time of \( s_1 \) should be later than that of \( s_2 \) in order to avoid congestion at the path \( \langle u, v, t \rangle \). This is a contradiction as we cannot find a feasible update time point for each switch. Fig. 9 captures this case in the time-extended network, where \( \sigma_{s_1,w} = \sigma_{s_1,u} = 2 \) and \( \sigma_{s_2,w} = \sigma_{s_2,u} = \sigma_{w,z} = \sigma_{v,t} = 3 \).

The complete process of our scheduling algorithm is shown in Algorithm 2. We first calculate the set of all update blocks \( \{o_f^i(t_i)\} \) starting at time step \( t_0 \) (line 1). Then we add the set of update blocks \( \{o_f^i(t_{-x})\} \) at the past time steps whose
update time points should be earlier than that in \( \{ o^f_j(t_0) \} \) (line 2). After that, we construct the set \( O \) and adjust each update block such that all of them start at current or future time steps (lines 3–4). When the merge operation is done, we check whether the equation \( o^f_j(t - \alpha) = o^f_j(t) \) can be established or not. If this condition holds, the algorithm stops because a deadlock forms, and we cannot schedule the update block \( o^f_j \) at two different time points \( (t - \alpha \text{ and } t) \) simultaneously (lines 5–7). Next we apply Algorithm 1 to construct the resource dependency graph \( G_0 \) and schedule each update block step by step (lines 8–20). In each time step, we apply Algorithm 3 to obtain the independent set \( \hat{O} \) and try to update them using Algorithm 4 (lines 10–11). If all updates are feasible, we add \( \sigma \) time steps for the rest of each update block, where \( \sigma \) is the maximum path delay obtained from Algorithm 4 (lines 15–16).

For the rest of the dependent update block, we update using Algorithm 4 as well (lines 17–18). When the loop terminates and the set \( G_o \) is empty, our algorithm outputs a feasible update sequence. Otherwise, it indicates that a feasible solution does not exist (lines 22–24). For convenience, the main steps are illustrated in Table II.

**Algorithm 3: Finding the set of independent blocks**

**Input:** The resource dependency graph \( G_o \), the set of all update blocks \( O \), the time step \( t \).

**Output:** The set of independent update blocks \( \hat{O} \).

```
1: for each \( o^f_j(t) \in O \) do
2:   for each \( \langle u, v \rangle \in p^f_{j,t} \) do
3:     if \( C^u_{u,v} < d_f \) then
4:       continue
5:     \( \hat{O} = \hat{O} \cup \{ o^f_j(t) \} \)
6: end
7: return \( \hat{O} \)
```

Algorithm 3 describes the procedure of finding the set of independent update blocks. For each one, if it can directly move to the final path without link capacity violation, we add it into set \( \hat{O} \) and the algorithm enters into the next loop.

**Algorithm 4: Updating the resource dependency graph**

**Input:** The set of update blocks \( O \); the initial path \( p^f_{init} \) and the final path \( p^f_{fin} \) for each flow \( f \in F \).

**Output:** The resource dependency graph \( G_o \) and an indicator variance that indicates whether the update is feasible or not.

```
1: \( \sigma_{\text{max}} = \sigma = 0 \)
2: for each \( o^f_{j,t} \in O \) do
3:   Update \( o^f_j \) at \( t \)
4:   for each \( \langle u, v \rangle \in p^f_{j,t} \) do
5:     \( G_o = G_o \setminus \{ \langle u, v \rangle \rightarrow o^f_j(t) \} \)
6:   \( C^u_{u,v} = C^u_{u,v} + d_f \)
7:   \( \sigma = \sigma + \sigma_{u,v} \)
8:   for each \( \langle u, v \rangle \in p^f_{j,t} \) do
9:     \( G_o = G_o \cup \{ o^f_j(t) 
10: \) \( \rightarrow \langle u, v \rangle \} \)
11:   \( C^u_{u,v} = C^u_{u,v} - d_f \)
12: if \( C^u_{u,v} < 0 \) then
13:   return \(-1\)
14: if \( \sigma > \sigma_{\text{max}} \) then
15:   \( \sigma_{\text{max}} = \sigma \)
16: \( \sigma = 0 \)
```

Algorithm 4 shows how to update a resource dependency graph. A possible schedule for a specific update block is...
that the residual capacity $C^r_{u,v}$ of all links in the final path can accommodate the flow demand. If so, the delay $\sigma_{\text{max}}$ is returned and will be used in Algorithm 2. Otherwise, the integer $-1$ is returned, indicating that the update is infeasible (lines 11-12). When the update is done, the residual capacity $C^r_{u,v}$ of all links on the initial (final) path will be increased (decreased) by a flow demand (lines 4-10).

Based on the above, we have the following theorem.

**Theorem IV.1.** The timed update sequence obtained from Algorithm 2 is congestion-free.

### V. Experimental Evaluation

We evaluate our scheduling algorithm using both prototype implementation and large-scale simulation.

**Benchmark schemes:** We compare the following schemes with our algorithm.

- **TPP:** The two-phase update protocol [30] where we use VLAN ID as the version number in our experiments.
- **NOP:** The node ordering protocol [24] that avoids black holes and forwarding loops [18].
- **Chronicle:** Our scheduling algorithm in Algorithm 2.
- **OPT:** The optimal solution of the MUTP integer program obtained using branch and bound.

We use two types of trace — the uniform trace generated in [1] and the production data mining trace [15] — in our evaluation, and the flow volume distribution is shown in Tab. III. We change the flow demand to simulate traffic in [1] and the production data mining trace [15] — in our experiments. We evaluate our scheduling algorithm using both prototype implementation and large-scale simulation.

**TABLE III**

<table>
<thead>
<tr>
<th>Flow volume</th>
<th>1-10K</th>
<th>10-100K</th>
<th>0.1-1M</th>
<th>1-10M</th>
<th>&gt; 10M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform [1]</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
</tr>
<tr>
<td>Data mining [15]</td>
<td>81.25%</td>
<td>2.15%</td>
<td>7.95%</td>
<td>4.15%</td>
<td>4.5%</td>
</tr>
</tbody>
</table>

**A. Implementation and Testbed Emulations**

**Implementation:** We develop a prototype of our algorithm using OFSoftSwitch and Dpctl [5] as Openflow switches and the controller. Now we describe how to perform accurate timing in our algorithm. We first obtain a solution to MUTP using Algorithm 2. Next we send update messages to each switch. We first send an OFPBCT_OPEN_REQUEST message to open a bundle, and then send a sequence of OFPT_BUNDLE_ADD_MESSAGE messages in order to modify the rules. Modifications are stored in a temporary staging area without taking effect. Next we close the bundle. Finally when a bundle is committed, the modifications will be applied to the switch at a specific time point.

**Testbed setup:** Our experiments are performed using a 5-server testbed, equipped with two Intel E5-2650 CPUs with 12 cores and 64 GB memory. Each server runs a software-based Openflow switch [5]. We adopt a small scale topology with 5 switches and seven 1 Gbps links as illustrated in Fig. 1(a). We use the Network Time Protocol (NTP) to synchronize the clocks of all the switches. The scheduled bundles feature [3] is used to guarantee accurate timing. We use pktgen to generate different numbers of flows according to the trace information shown in [1] and [15]. The aggregate flow rate is 1 Gbps in total. The forwarding rules are installed and updated via Dpctl API [5].

**Experiment results:** Since the flow volume differences in [1] and [15] cannot have a significant impact on the update time, we only use one figure to show them. Fig. 10 shows the total update time for different schemes. As Openflow barrier feature cannot provide accurate acknowledgments [20] to indicate the completion of an update operation, we use tcpdump—a powerful packet analyzer—to confirm when the new rules take effect. We can observe that the update time of TPP and NOP is around 55 ms on average, while Chronicle is around 20 ms. Specifically, the update time of Chronicle, NOP and TPP is 21.59 ms, 53.96 ms and 58.19 ms, respectively, when the number of flows is 20, while that of them is 21.80 ms, 57.29 ms and 59.86 ms, when the number of flows is 100. Chronicle can hence reduce the update time by 63% compared to NOP and TPP. This demonstrates that Chronicle can leverage the benefits of accurate timing to accelerate the update process, reducing the time overhead resulting from the wait invoke pattern.

In Fig. 11, we measure flow completion time based on data mining workloads. In our experiments, the number of flows is fixed at 1000 and the maximum length for each switch port is 250KB. We can observe that the average FCT of Chronicle, NOP and TPP is 227.6ms, 488.2ms and 614.6ms, respectively. Obviously, NOP and TPP have more FCT than Chronicle. This is because the number of congested links using NOP and TPP is greater than that using Chronicle, due to the asynchronous network updates, which results in more FCT. Chronicle takes advantage of accurate timing to reduce congestion during network updates and thus has a better FCT. In addition, we note that FCT for TPP is slightly longer due to packet tagging.

To assess the sensitivity of different flow volumes, we define DFCT for TCP flows as the following equation.

$$DFCT = FCT_{\text{update}} - FCT_{\text{normal}}$$

where $FCT_{\text{update}}$ and $FCT_{\text{normal}}$ indicate the flow completion time with and without network updates. Fig. 12 and Fig. 13 show the DFCT results for data mining workloads with smaller and larger link delay, respectively. In Fig. 12, we can observe that the flow with larger volume has more DFCT. The reason for this is that only the flow with larger volume has the opportunity to enter the congestion avoidance phase and...
avoidance phase and thus have more opportunities to enter congestion. We can observe that the flows routed through the link network updates and thus has a better FCT updates, which result in more congested flows than that using Chronicle due to the asynchronous network update. Specifically, the average DFCT of Chronicle, NOP and TPP is 0.78ms, 0.91ms and 1.87ms when start phase can be restarted. Specifically, the average DFCT of Chronicle due to the asynchronous network updates, which result in more congested flows. This demonstrates that Chronicle takes full advantage of accurate timing and completely lead to more congested flows.

### B. Simulation

We also conduct extensive simulations to evaluate our algorithm at scale.

**Setup.** In addition to the small-scale topology used in our testbed, here we use a large-scale synthetic scale-free topology that is randomly produced by the scale_free_graph function [2]. There are 100 switches and 586 10 Gbps links in total. We generate different numbers of flows according to the flow volume information in [1] and [15] with demand ranging from 100 Mbps to 800 Mbps. Accordingly, we adjust the link capacity in order to ensure that the congestion-free condition holds in both initial and final stages. The link delay between switches is set to be a random number ranging from 1 to 50 time units. We run the algorithms on a server with Intel Xeon CPU and 15 GB memory. Each data point is an average of ten runs.

**Experiment results:** We first investigate the number of congested flows with different numbers of network flows during the entire update process. We can see that in Fig. 14, as the number of flows increases, NOP and TPP yield significantly more congested flows, while Chronicle yields zero all the time. Specifically in Fig. 14(a), the number of congested flows for NOP and TPP is 471 and 478 respectively, when the number of flows is 2500. The congested flows for NOP and TPP account for around 20% of the total flows in the network. Furthermore, we conduct experiments to show the number of congested flows with different numbers of switches. In this settings, we fix the number of flows to 1000 and vary the number of switches in each run. As shown in Fig. 15, we observe that using NOP and TPP in small-scale networks can lead to more congested flows. This demonstrates that Chronicle takes full advantage of accurate timing and completely avoids congestion by assigning different update points for each flow.

Fig. 16 shows the maximum link utilization varies with time across active links for different schemes. For this sim-
when the time step is 25. We observe that Chronicle and NOP can save over 35% more rules than TPP on average as shown in Fig. 19. Note that we only show the number of rules during the update; the old rules will be removed by the controller when the update is done. Note that these results become inaccurate for switches that apply longest prefix matching or wild-card rules. However, such rules are increasingly being substituted with exact match rules in SDNs [18].

Fig. 20 shows the number of changed rules during update. We define the number of changed rules as the number of rules that needs to be added, modified or deleted during the update. Essentially this measures the number of operations, as well as the number of flow table entries required to perform the update. We observe that TPP induces more changed rules than NOP and Chronicle. When the number of flows is 2000, the changed rules of TPP, NOP and Chronicle are 16569, 9069 and 9069 respectively. TPP has almost twice the amount of changed rules as that of NOP and Chronicle. This is because TPP relies on different version numbers to indicate two stages during the update. This process involves more update (add/remove) operations compared to NOP and Chronicle.

Finally, we show the optimality gap in percentile. Fig. 21 shows the box plot of the optimality gap between Chronicle and OPT as the number of flows increases. We can see that the optimality gap in the worst-case is 30% and that in the average-case is 20%. In general, the difference between Chronicle and OPT can be accepted in practice.

VI. CONCLUSION

This paper initiated the algorithmic study of minimizing the makespan in timed SDNs, an interesting optimization opportunity introduced by recent SDN technology. We proved that the problem is NP-hard in general and proposed Chronicle to find a feasible update sequence in polynomial time. Our evaluation results show that Chronicle can significantly reduce the update makespan. We believe that our work opens interesting directions for future research. In particular, while our algorithms and prototype show the potential of such optimizations, it will be interesting to chart a more comprehensive
landscape of the tradeoff between update time and quality of the resulting schedule. We could approach it analytically, but also in different experimental case studies or considering randomized approaches (which may tolerate a short and small amount of congestion if it reduces the update time).

REFERENCES


Jiaqi Zheng is currently a Research Assistant Professor from Department of Computer Science and Technology, Nanjing University, China. His research interests include data center networks, network function virtualization, distributed systems, Internet streaming and urban computing.

Bo Li Bo Li received the B.S. degree from the Department of Computer Science and Engineering at the Nanjing University of Science and Technology, China, in 2016. He is a 3rd-year M.S. student in Nanjing University, China. His research interests include distributed networks and systems.

Chen Tian is an associate professor with State Key Laboratory for Novel Software Technology, Nanjing University, China. He was previously an associate professor with School of Electronics Information and Communications, Huazhong University of Science and Technology, China. Dr. Tian received the BS (2000), MS (2003) and Ph.D. (2008) degrees at Department of Electronics and Information Engineering from Huazhong University of Science and Technology, China. From 2012 to 2013, he was a postdoctoral researcher with the Department of Computer Science, Yale University. His research interests include data center networks, network function virtualization, distributed systems, Internet streaming and urban computing.

Klaus-Tycho Förster is a Postdoctoral Researcher at the University of Vienna, working with Stefan Schmid. He was previously a PostDoc at Aalborg University, Denmark, and a Visiting Researcher at Microsoft Research, Redmond, USA, with Rataf Mahajan. He obtained his PhD degree (2016) from ETH Zurich, advised by Roger Wattenhofer. His research interests revolve around algorithms and complexity in the areas of networking and distributed computing.

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.
Stefan Schmid is an Associate Professor at Aalborg University, Denmark. He received his MSc (2004) and PhD (2008) from ETH Zurich, Switzerland. Subsequently, Stefan Schmid worked as postdoc at TU Munich and the University of Paderborn (2009). From 2009-2015, he was a senior research scientist at the Telekom Innovations Laboratories (T-Labs) in Berlin, Germany. His research interests revolve around the fundamental algorithmic problems of networked and distributed systems.

Guihai Chen is a distinguished professor of Shanghai Jiao Tong University. He earned BS degree in computer software from Nanjing University in 1984, ME degree in computer applications from Southeast University in 1987, and PhD degree in computer science from the University of Hong Kong in 1997. He has a wide range of research interests with focus on parallel computing, wireless networks, data centers, peer-to-peer computing, high-performance computer architecture and data engineering.

Jie Wu is the Associate Vice Provost for International Affairs at Temple University. He also serves as the Chair and Laura H. Carnell professor in the Department of Computer and Information Sciences. His current research interests include mobile computing and wireless networks, routing protocols, cloud and green computing, network trust and security, and social network applications. Dr. Wu is a CCF Distinguished Speaker and a Fellow of the IEEE. He is the recipient of the 2011 China Computer Federation (CCF) Overseas Outstanding Achievement Award.

Rui Li received his M.S. Degree from Central South University in 2004 and Ph.D. Degree in Computer Science from Hunan University in 2012. He received the Outstanding Young Teacher Award of Hunan Province in 2009 and the Distinguished Young Teacher Award by luaxiong Liu Funding in 2010. He is currently an Associate Professor of Dongguan University of Technology. His research interests are in cloud computing, sensor networks, and security.