Computers, Materials & Continua

Tech Science Press

Doi:10.32604/cmc.2025.069533

ARTICLE

P4LoF: Scheduling Loop-Free Multi-Flow Updates in Programmable Networks

Jiqiang Xia[1], Qi Zhan[1], Le Tian[1,2,3,*], Yuxiang Hu[1,2,3] and Jianhua Peng[4]

[1]Information Engineering University, Zhengzhou, 450001, China
[2]National Key Laboratory of Advanced Communication Networks, Zhengzhou, 450001, China
[3]Key Laboratory of Cyberspace Security, Ministry of Education, Zhengzhou, 450001, China
[4]Purple Mountain Laboratories, Nanjing, 210000, China

*Corresponding Author: Le Tian. Email: le.tian2019@outlook.com

Received: 25 June 2025; Accepted: 28 August 2025; Published: 10 November 2025

ABSTRACT: The rapid growth of distributed data-centric applications and AI workloads increases demand for low-latency, high-throughput communication, necessitating frequent and flexible updates to network routing configurations. However, maintaining consistent forwarding states during these updates is challenging, particularly when rerouting multiple flows simultaneously. Existing approaches pay little attention to multi-flow update, where improper update sequences across data plane nodes may construct deadlock dependencies. Moreover, these methods typically involve excessive control-data plane interactions, incurring significant resource overhead and performance degradation. This paper presents P4LoF, an efficient loop-free update approach that enables the controller to reroute multiple flows through minimal interactions. P4LoF first utilizes a greedy-based algorithm to generate the shortest update dependency chain for the single-flow update. These chains are then dynamically merged into a dependency graph and resolved as a Shortest Common Super-sequence (SCS) problem to produce the update sequence of multi-flow update. To address deadlock dependencies in multi-flow updates, P4LoF builds a deadlock-fix forwarding model that leverages the flexible packet processing capabilities of the programmable data plane. Experimental results show that P4LoF reduces control-data plane interactions by at least 32.6% with modest overhead, while effectively guaranteeing loop-free consistency.

KEYWORDS: Network management; update consistency; programmable data plane; P4

1 Introduction

The rapid growth of distributed data-centric applications and AI workloads has led to a significant increase in communication traffic, placing immense demands on network infrastructure [1]. The performance of these systems relies on low-latency, high-throughput connectivity, making it essential to optimize resource utilization and operational agility. To meet these requirements, great efforts have been made to render communication networks more flexible and programmable, especially in the context of Software Defined Networking (SDN) architectures [2]. These technologies enable traffic routes to be adjusted in real time based on changing demands, including load balancing, traffic engineering, and fault recovery. However, this architectural programmability introduces critical challenges in maintaining operational consistency during updates. A primary concern arises when transitioning between routing configurations. The asynchronous and distributed nature of network devices can lead to temporary inconsistencies in their forwarding states [3,4]. For instance, during efforts to mitigate congestion or optimize paths, the controller implements new routing rules to replace outdated ones. Unfortunately, the absence of atomic synchronization across

devices results in staggered rule activations: some nodes may adopt the new configuration while others continue to use outdated rules. These hybrid forwarding states can cause forwarding loops, leading to unexpected network performance degradation like packet loss, TCP reordering, and reduced reliability [3].

In recent years, researchers have investigated a variety of studies on how to consistently update network configurations. Existing solutions can be classified into three primary categories: two-phase update, ordering-based update, and time-based update. Reitblatt et al. [5] first propose the two-phase update solution that simultaneously maintains the initial and new configurations via packet tagging. Subsequent improvements in two-phase methodology [6,7] have primarily focused on mitigating inherent memory overheads caused by supplementary flow rules. In contrast, ordering-based update solutions eliminate requirements for packet labeling or costly TCAM memory consumption [8,9]. This paradigm achieves network path migration through strategic scheduling of configuration update sequences. Furthermore, unlike time-based update approaches [10,11] that rely on precise clock synchronization, ordering-based approaches gradually conduct rule updates among switches according to the scheduled update sequence. Therefore, the ordering-based multi-round update schemes have been widely studied and applied. These solutions can be briefly detailed as follows: the nodes arranged in the $t+1$ round can be updated only after the nodes in the t round have been updated, and all the nodes within the same update round can be updated concurrently.

However, existing ordering-based update approaches still encounter several critical challenges. First, while practical network updates often involve concurrent modifications to multiple flows [12], most previous studies primarily focus on maintaining consistency for single-flow updates [8,9,13]. For example, redistributing traffic in batches to individual servers in a Content Distribution Network [14] or redeploying a load balancing policy with a global network view in an SDN involves updating multiple flow rules. Second, the update process must minimize control-data plane interactions due to the inherent resource costs associated with switch reconfiguration and flow rules modifications. The controller should interact with the forwarding nodes as infrequently as possible to reduce the impact of the update process on the normal communication of the network. Third, considering the complexity of multi-flow update, the dependency of update sequences between nodes may constitute a deadlock [15,16]. The emergence of programmable data plane, enabled by domain-specific languages like P4 [17], offers promising solutions through flexible packet processing and fine-grained forwarding control [18]. This provides a possible solution to solve the update deadlock problem and makes update consistency guaranteed efficiently.

To address these problems, we propose P4LoF, an efficient loop-free update approach that reroutes multiple flows with minimal control-data plane interactions. First, we analyze the update dependency in the single-flow update scenario and build the shortest update dependency chain based on a greedy-based algorithm. Then, we analyze the complexity of multi-flow update and dynamically construct the dependency graph to compute the update sequences. To address the update deadlock dependencies in multi-flow update, we present a deadlock-fix forwarding model based on the flexible packet processing mechanism of the programmable data plane. Finally, we present the loop-free multi-round update plan based on the computed update sequence and the deadlock-fix forwarding model. We highlight the main contributions of our work as follows.

(1) We design a loop-free update solution, named P4LoF, which enables the controller to efficiently reroute multiple flows through minimal interactions with each switch in programmable networks.
(2) We propose a greedy-based algorithm to generate the shortest loop-free update dependency chain for each flow, then merge these chains into a dependency graph and solve it as a Shortest Common Super-sequence (SCS) problem to schedule loop-free multi-flow update.

(3) We present a deadlock-fix forwarding model to resolve deadlock dependencies of multi-flow update through consistent forwarding control.
(4) Based on real-world network topologies, we evaluate the update efficiency of P4LoF. Experimental results show that it significantly reduces the control-data plane interactions while guaranteeing the loop-free update consistency of multiple flows.

The rest of this paper is organized as follows. Section 2 presents the detailed clarification and analysis of the loop-free consistency under the single-flow/multi-flow update scenarios which construct the motivation of this paper. Section 3 provides the design overview of our proposed loop-free update solution P4LoF. Section 4 presents the experimental investigation and Section 5 introduces related work. Section 6 gives the conclusion of this paper.

2 Motivation

This section motivates the need for P4LoF by analyzing the loop-free update consistency through two practical example scenarios involving the rerouting of both single and multiple flows. The key notations involved in this paper are shown in Table 1.

Table 1: Notations in this paper

Notation	Description
P_{ini}^{f}	Initial path of flow f
P_{fin}^{f}	Final path of flow f
$f\left(s_i^{ini}\right)$	Initial rule of flow f in nodes_i
$f\left(s_i^{fin}\right)$	Final rule of flow f in nodes_i
$U\left[f\left(s_i\right)\right]$	Rule updating time of flow f in the node s_i
$\varphi_{[\alpha,\beta]}^{f}$	Dependency chain of flow f from node α to node β
$\Phi_{[\alpha,\beta]}$	The set of dependency chains of all flows from node α to node β
$N_{[\alpha,\beta]}$	The number of flows subjecting to $\Phi_{[\alpha,\beta]}$, $N = \left\|\Phi_{[\alpha,\beta]}\right\|$
$l_{[\alpha,\beta]}$	Length of the dependency chain, $l = \text{len}\left\{\varphi_{[\alpha,\beta]}^{f}\right\} = \text{len}\left\{\Phi_{[\alpha,\beta]}\right\}$
$S\left(f\right)$	Updating sequence of flow f

2.1 Motivation Examples

Example 1: Loop-Free Consistency in Single-Flow Updates. Fig. 1 illustrates the importance of loop-free consistency when updating a single flow's path. Flow f is scheduled to move from path $s_1 \to s_4 \to s_3 \to s_5$ (solid) to $s_1 \to s_2 \to s_3 \to s_4 \to s_5$ (dashed). The controller sends new rules to switches s_1, s_2, s_3, s_4 to implement this change.

However, network updates often experience delays and are unlikely to complete simultaneously. This lack of synchronization can create forwarding loops for flow f. For instance, if $U[f(s_3)] < U[f(s_4)]$ (meaning s_3 updates before s_4), packets sent from s_4 to s_3 might loop back to s_4, creating a transient loop and potentially causing network congestion. Loops can also form between multiple nodes with wider consequences.

Example 2: Loop-Free Consistency of Multi-Flow Update. The update consistency constraint of flow f in Fig. 1 has specified as $U\left[f\left(s_3\right)\right] < U\left[f\left(s_4\right)\right]$. Now we add a new flow $\widehat{f}$ whose update path is exactly opposite to flow f. Then the consistency constraint of flow $\widehat{f}$ is opposite to flow f (i.e., $U\left[\widehat{f}\left(s_4\right)\right] <$

$U\left[\widehat{f}\left(s_3\right)\right]$). The consistency constraint for both flow f and flow $\widehat{f}$ is obviously not satisfied by any node update sequence, since the update order of these two flows constitutes a deadlock dependency.

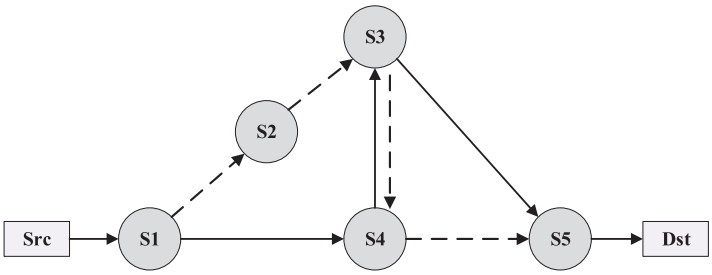


Figure 1: An example of the loop-free consistency update

2.2 Problem Analysis

This section analyzes the challenge of maintaining loop-free consistency during flow rule updates. We highlight the update dependency problem in single-flow updates, where improper sequences can create transient loops. The analysis extends to multi-flow scenarios, showing how conflicting dependency chains can lead to NP-hard deadlocks that worsen update efficiency.

2.2.1 Loop-Free Consistency of Single-Flow Update

The rule update problem is essentially a path switching forwarding problem with the forwarding node as the basic update unit, and loop-free consistency is the basic property to maintain in this process. The rule update of flow f shown in Fig. 1 is to switch the forwarding path $P_{ini}^{f} = \{src, s_1, s_4, s_3, s_5, dst\}$ to $P_{fin}^{f} = \{src, s_1, s_2, s_3, s_4, s_5, dst\}$. In this path switching process, if left uncontrolled, a transient forwarding loop is likely to occur, leading to network congestion and other problems and affecting the normal communication of the network. Before further analyzing and resolving the forwarding loop problem, we first define the forwarding loop in the single-flow update.

Definition 1: forwarding loop in single-flow update. Assume that the initial and final forwarding paths of flow f are P_{ini}^{f}, P_{fin}^{f} respectively where $P_{ini}^{f} = \left\{src, \ldots, s_i, \ldots, s_j, \ldots, dst\right\}$ and $P_{fin}^{f} = \left\{src, \ldots, s_j, \ldots, s_i, \ldots, dst\right\}$ (i ≠ j), and there are no forwarding loops in either path. If node s_j completes the update of the flow rule before node s_i, a loop is generated among $\left\langle s_i, \ldots, s_j \right\rangle$.

According to Definition 1, when both the old and new paths contain two nodes in the opposite order, to maintain the loop-free consistency property in the update process, the update scheme needs to ensure that node s_i always completes the update before the node s_j, i.e., there is a dependency relationship between the updates of nodes $U\left[f\left(\mathrm{s}_i\right)\right] < U\left[f\left(\mathrm{s}_j\right)\right]$. If the formation of the forwarding loop involves more nodes, the update order among these nodes will constitute a dependency chain.

2.2.2 Loop-Free Consistency of Multi-Flow Update

In a multi-flow update scenario, independent update paths can be managed using a single-flow update solution for parallel updates. However, if the paths intersect, it leads to an NP-hard problem due to potential conflicts in node update sequences between different flows [13]. We will define multi-flow update deadlock.

Definition 2: Multi-flow update deadlock dependency. Assume that there are no forwarding loops in the initial and final forwarding paths of each flow in the flow set F to be updated. If there exists a set of nodes $s_1, s_2, \ldots, s_k$ $(k \geq 2)$ satisfying the update dependency in the following formulation, then an update deadlock

is created among $f_1, f_2, \ldots, f_n$.

$$\left\{\varphi^{f_1}_{[s_1,s_2]}, \varphi^{f_2}_{[s_2,s_3]}, \ldots, \varphi^{f_n}_{[s_k,s_1]} | f_1, f_2, \ldots, f_n \in F, n \leq |F|\right\} \tag{1}$$

To address this problem, a simple solution is to use the flow rule as the basic update unit to complete updates to flow f and $\widehat{f}$ sequentially:

$$\left\{U\left[f\left(s_4\right)\right], U\left[\hat{f}\left(s_3\right)\right]\right\} < \left\{U\left[f\left(s_3\right)\right], U\left[\hat{f}\left(s_4\right)\right]\right\} \tag{2}$$

However, such an update approach increases interactions between the switches and the controller, reducing update efficiency. To address this, we propose using forwarding nodes as the basic update unit, allowing the controller to update all relevant flow rules at each node in a single interaction. However, maintaining consistency during multi-flow updates is challenging, as simple ordering can lead to deadlocks. Thus, achieving loop-free updates for multiple flows involves solving two key problems.

3 P4LoF Design

This section clarifies the design details of P4LoF and its workflow depicted in Fig. 2. We first describe the loop-free consistency problem in the single-flow update scenarios, and then analyze the complexity of the multi-flow loop-free updates. Finally, the multi-flow loop-free update scheme is given.

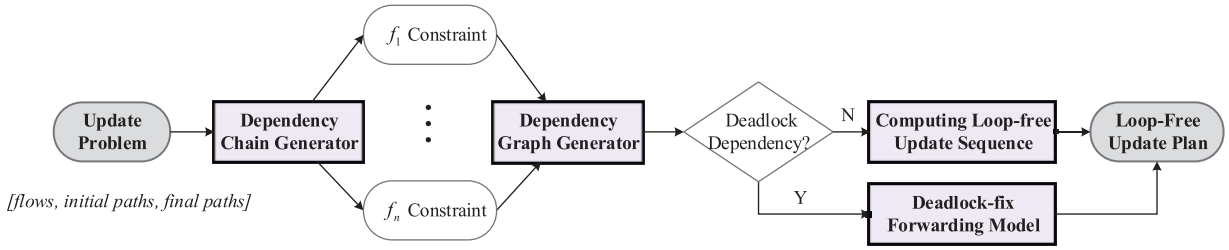


Figure 2: The main workflow of P4LoF

3.1 Dependency Chain Generator for Single-Flow Update

In this paper, we design a dependency generating algorithm to obtain the minimal length of the dependency chain based on the greedy algorithm. The minimal represents that no node can improve its dependency without some other node getting worse. There are no loops in the initial forwarding path of flow f in Definition 1. Therefore, we can construct a Directed Acyclic Graph (DAG) from the source node src to the destination node dst: if there is an initial flow rule configuration $s^{ini} = t$ in node s, a directed edge from s to t is added to the graph, indicating that packets arriving at node s are forwarded to node t. After that, we gradually add directed edges (new flow rule configurations) to this DAG and process it with a custom subroutine (Algorithm 1) to construct the minimal length dependency chain as follows. During dependency chain generation, each node transitions through global states $init$, $limbo$, and fin. Within the loop-detection subroutine (Algorithm 1), nodes use transient local states $unknown$, $seen$, and $visited$.

(1) Set the state of all nodes in the DAG to the initial state $init$;
(2) Traverse each node in the graph: For each node u, invoke Algorithm 1 to check if adding its final forwarding path (i.e., $u^{fin} = v$) introduces a loop. If no loop is detected, transition u to transient state $limbo$ and it is set as the starting node of a dependency chain; otherwise, the state of node u remains as state $init$.

(3) For each node u with state $limbo$, remove its initial forwarding path and set state to fin. Then invoke Algorithm 1 to check if any $init$ node v satisfies this condition: since the initial forwarding path of u is removed from the graph, the final forwarding path of node v is added without introducing a loop. If there is such a node v that satisfies this condition, its state is set to $limbo$. After that, we add v to the dependency chain starting with u and use it as a starting point for the next round of search. Otherwise, the search continues for other $limbo$ nodes.

(4) Repeat (3) until the states of all nodes are set as fin.

The dependency chain construction method allows nodes to be added only after their predecessor removes the initial forwarding rule, ensuring timely updates and minimal dependencies. In multi-flow updates, flows without intersecting dependencies can be updated in parallel using the dependency chain. However, due to complex dependencies, we dynamically create a multi-flow update dependency graph to enable loop-free updates.

Algorithm 1: Loop-detection subroutine of dependency generating algorithm

Input: The directed graph G_f of flow f, rule $u^{fin} = v$.
Output: Ture (loop-free) or False (got a loop).

$state\ (u) = seen, state\ (v) = seen$
set the states of other nodes in the graph as *unknown*
$\hat{G}_f = G_f + edge\ (u \to v)$
set node v as the start node
for each node w pointed by node v **do**
  if $state\ (w) = seen$ **then**
    return *False*
  else if $state\ (w) = unknown$ **then**
    $state\ (w) = seen$
    $Check\left(\hat{G}_f, w\right)$
    $state\ (w) = visited$
  end if
end for
return *True*

3.2 Deadlock Detection and Removal for Multi-Flow Update

This section presents how P4LoF resolves the deadlock dependency in multi-flow updates. We first transform complex flow-level constraints into a dependency graph model where switches are aggregated into supernodes and dependency chains manifest as directed edges. Subsequently, we break detected loops through enforcing temporary P4-based forwarding rules to maintain consistency.

3.2.1 Dependency Graph Generator

Existing studies usually take flow rules as the basic update unit and formulate the update process as a series of LPs [8,19] (Linear programming): minimizing the communication amount and delay between the controller and switches as the objective, and the consistency update property as a constraint. From the analysis above for the flow f and $\hat{f}$ updating process, it is clear that such an approach would significantly increase the number of control plane and data plane interactions, causing extensive network delays.

To address this problem, we replace those complex LPs with a dependency graph. In the single-flow update scenario, each node in the dependency chain is represented as the combination of node ID and flow ID, i.e., $s_i\_f_j$. For multi-flow update, all flow rules on the same switch (e.g., $s_i\_f_j, \ldots, s_i\_f_k$) are grouped into a single supernode, allowing simultaneous updates of all rules within the switch. In this context, the complex update constraints are reformulated as a dependency graph, where each supernode represents a switch-level update set, and directed edges encode consistency constraints between these sets. In the dependency graph, multiple consistency constraints that constitute a deadlock are visualized as a loop. Then update deadlock detection can be translated into how to discover loops in the dependency graph, which becomes much easier and can be solved in polynomial time by some searching algorithms (like BFS, DFS, etc.).

Further optimization in graph construction is achieved using a sliding window mechanism, which segments the sequence of switch-level update sets (supernodes) into smaller windows. Each window allows for incremental analysis of a subset of the graph, balancing computational latency and update efficiency. A larger window reduces control-data plane interactions by processing more supernodes in batches but increases computation time for resolving dependencies. Conversely, a smaller window allows for faster processing but leads to more frequent updates and coordination overhead. The window size can be dynamically adjusted based on real-time network conditions to optimize performance during low congestion or traffic bursts.

The sliding window size is dynamically adjusted in real-time to balance computational latency with update efficiency. In our implementation, we adjust the window size by simply adopting a threshold-based method. This adjustment strategy relies on three key network metrics: current network bandwidth utilization U_b and the minimum/maximum length of single-flow dependency chains C_{min}/C_{max}. We dynamically configure the length of the sliding window L_{win} based on the following equation.

$$L_{win} = \frac{C_{max} + C_{min}}{2} + f(U_b) \cdot \frac{C_{max} - C_{min}}{2}, f(U_b) = \begin{cases} 1, U_b < T_{low} \\ 0, U_b \in \left[T_{low}, T_{high}\right] \\ -1, U_b > T_{high} \end{cases} \tag{3}$$

When the network bandwidth utilization exceeds a predefined high threshold T_{high}, the window size is reduced to C_{min} to minimize computational delays and prevent further network congestion. Conversely, if bandwidth utilization falls below a low threshold T_{low}, the window size is increased to C_{max} to enhance the efficiency of control-data plane interactions. Otherwise, the window length is set as the mean value of C_{min} and C_{max}.

3.2.2 Deadlock Removal

After combining the constraints of each flow into a dependency graph and detecting the loop structures that constitute deadlocks from it, we need to fix those deadlocks to calculate the final global update sequence. For each deadlock $L = \{\Phi_{[n_1,n_2]}, \Phi_{[n_2,n_3]}, \cdots, \Phi_{[n_k,n_1]} | k \geq 2\}$ (n_i represents one node in the dependency graph), we break its loop by removing a set of dependency chains from the loop (sequence edges set from node α to node β in the dependency graph) to break its loop structure and thus obtain a sequence of loop-free updates of nodes.

$$\Phi_{[\alpha,\beta]} = \left\{ \varphi_1, \cdots, \varphi_{N_{[\alpha,\beta]}} | \varphi_i = \varphi_{[\alpha,\beta]}^{f_j}, 1 \leq i \leq N_{[\alpha,\beta]}, f_j \in F \right\} \tag{4}$$

Also, to reduce the additional overhead caused by the deadlock-fix model, we carefully choose to cut off the one dependency chain that contains the least number of flows in the loop.

$$l_{[\alpha,\beta]} = \min\left\{l_{[n_1,n_2]}, l_{[n_2,n_3]}, \cdots, l_{[n_k,n_1]}\right\} \tag{5}$$

For each flow corresponding to the cropped $\Phi_{[\alpha,\beta]}$, we configure additional forwarding rules in the starting node α of the dependency chain to ensure that the forwarding path of the traffic of each flow still meets the loop-free consistency requirement. The controller removes these additional rules from the nodes when each node in the loop that constitutes the deadlock completes its flow rule update. By introducing this additional forwarding mechanism (detailed in Section 3.3.2), we implement a sequential update scheme with nodes as the basic unit, which can maintain the loop-free consistency property during multiple flow updates simultaneously.

3.3 Loop-Free Multi-Flow Update

In practical networks, interactions between the control plane and the data plane require a certain resource cost. The more interactions the network has in the update process, the more it consumes switch resources. Besides, the time-consuming interaction process increases uncertainty and can further increase the security risk of network updates (e.g., rule loss, malicious tampering, etc.). Therefore, P4LoF needs to make each interaction cover as many nodes as possible to reduce resource consumption while ensuring the basic property of loop-free consistent updates.

3.3.1 Update Sequence Computing

We give some examples of dependency graph substructures shown in Fig. 3 to specify how we handle the multi-flow consistency update problem.

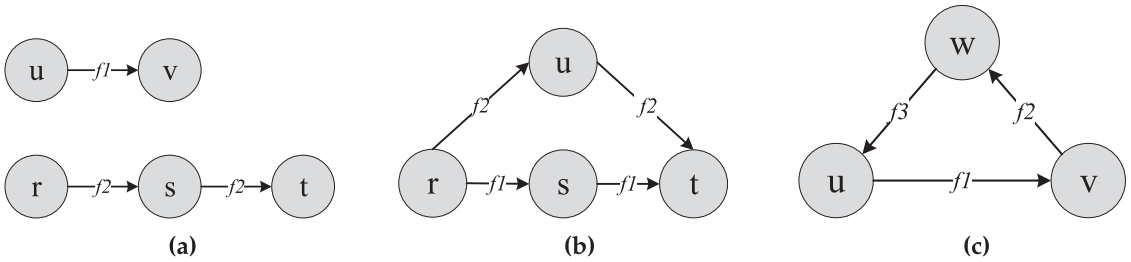


Figure 3: Dependency graph with different substructures. (**a**) Non-intersecting dependency; (**b**) Intersecting dependency w/o deadlock; (**c**) Deadlock dependency

Non-intersecting dependency. The dependency graph in Fig. 3a represents the update scenario where the consistency constraints of flow f_1 and f_2, do not exist for crossover nodes. In this case, the two flows can simply update each node in parallel according to their respective dependency chains (detailed in Section 2.2.1).

Intersecting dependency without deadlock. The dependency graph in Fig. 3b represents the update scenario where the consistency constraints of the update process of flow f_1, f_2 have intersecting nodes but do not constitute a loop. This dependency graph structure is a more common case in the update process. The update sequence of flow f_1, f_2 satisfies$\{r, s, t\} \subseteq S(f_1)$ and $\{r, u, t\} \subseteq S(f_2)$, respectively, then we can induce that $\{r, \{s, u\}, t\} \subseteq S(f_1, f_2)$, where nodes s,u belong to the dependency chain of flow f_1, f_2, respectively, and there is no dependency between the update order of the two nodes. P4LoF schedules similar nodes in the

same update round to improve the update efficiency. For solving the multi-flow update sequence problem with the dependency graph structure shown in Fig. 3b, P4LoF formulates the problem as calculating the Shortest Common Super-sequence (SCS) of a series of sequences.

The problem of calculating the SCS of two sequences can be simply described as follows: given the sequences Seq_1 and Seq_2, if some elements are removed from sequence S without changing the order of the elements in the sequence S, Seq_1 and Seq_2 can be obtained, and S is the common subsequence of Seq_1 and Seq_2. The shortest one of these supersequences is the SCS of Seq_1 and Seq_2. For the two sequences $Seq_1 = \{u_1, u_2, \ldots, u_m\}$ and $Seq_2 = \{v_1, v_2, \ldots, v_n\}$, their SCS can be calculated by a classical Dynamic Programming (DP) algorithm, which is formulated as,

$$L[m,n] = \begin{cases} m & n = 0 \\ n & m = 0 \\ L[m-1, n-1] + 1 & m, n > 0; u_m = v_m \\ \min\{L[m, n-1], L[m-1, n]\} & m, n > 0; u_m \neq v_m \end{cases} \tag{6}$$

where $L[m, n]$ represents the length of the SCS. Algorithm 2 details the calculation process which is mainly divided into two phases. First, it constructs a DP table L (lines 1–7) where $L[i][j]$ stores the SCS length for prefixes $Seq_1[1 \ldots i]$ and $Seq_2[1 \ldots j]$, initializing base cases (lines 1–2) and applying recurrence relations for character matches/mismatches. Second, it backtraces from $L[m][n]$ (lines 9–17) to build S by: 1) appending matching characters once, 2) selecting u_i or v_j for mismatches based on DP values, and 3) reversing the sequence (line 18).

Algorithm 2: Shortest common supersequence calculation

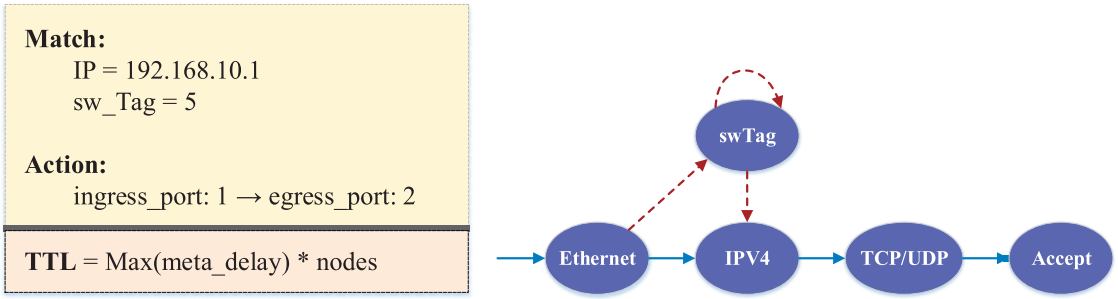
Input: Strings $Seq_1 = \{u_1, u_2, \ldots, u_m\}$ and $Seq_2 = \{v_1, v_2, \ldots, v_n\}$
Output: Shortest supersequence S and its length L

1: Initialize $L[i][0] \leftarrow i$ for $0 \le i \le m$ ▷ Initialize DP table
2: Initialize $L[0][j] \leftarrow j$ for $0 \le j \le n$
3: **for** $i \leftarrow 1$ **to** m **do**
4: **for** $j \leftarrow 1$ **to** n **do**
5: $L[i][j] \leftarrow 1 + \begin{cases} L[i-1][j-1] & \text{if } u_i = v_j \\ min(L[i-1][j], L[i][j-1]) & \text{otherwise} \end{cases}$
6: **end for**
7: **end for**
8: $(i, j) \leftarrow (m, n), stack \leftarrow \varnothing$ ▷ Reverse-order construction
9: **while** $i > 0$ **or** $j > 0$ **do**
10: **if** $i > 0$ **and** $j > 0$ **and** $u_i = v_j$ **then**
11: Push u_i to *stack*, $i \leftarrow i - 1$, $j \leftarrow j - 1$
12: **else if** $j = 0$ **or** ($i > 0$ **and** $L[i][j] = L[i-1][j] + 1$) **then**
13: Push u_i to *stack*, $i \leftarrow i - 1$
14: **else**
15: Push v_j to *stack*, $j \leftarrow j - 1$
16: **end if**
17: **end while**
18: $S \leftarrow$ reverse $(stack)$
19: **return** *S*, *L[m][n]* ▷ Return supersequence and length

Deadlock dependency. The dependency graph in Fig. 3c shows an update scenario where the consistency constraints of flows f_1, f_2, f_3 intersect at nodes and form a loop. This creates a deadlock: updating any node first violates the consistency constraints of the other two flows, meaning no valid update sequence simultaneously satisfies all constraints. To resolve this, we design the Deadlock-fix model. It breaks the deadlock while preserving multi-flow consistency by redirecting traffic along the shortest dependency chain within the deadlock structure. Specifically, node u is given an additional rule: upon receiving flow f_1, redirect it to node v. This enables the deadlock structure to be updated in the global sequence $\{v, w, u\} \subseteq S(f_1, f_2, f_3)$.

3.3.2 Deadlock-Fix Forwarding Model

To address the problem of conflicting node update order during multi-flow update, we propose the P4-based Deadlock-fix forwarding model: first, the controller adds swTag tags to each flow in the dependency chain at the flow entry based on the dependency chains set that need to be eliminated in the dependency graph. These tags correspond to the individual node identifiers (like switch ID) in the dependency chain. Then, before sending the flow rules to be updated to the data plane, the controller preloads additional forwarding rules in each switching node corresponding to the swTag (Fig. 4) and specifies the outgoing port of each flow as the forwarding port before the update, to ensure a loop-free update of the flows. Finally, the controller loads new configurations to the data plane based on the calculated node update sequence to complete the update of the data plane. In addition, the deadlock-fix forwarding model sets a TTL value for these additional flow rules (Fig. 4a) to ensure that these rules can release the occupied switch storage resources in time after each node that constitutes the update deadlock finishes updating the flow rules.



(a) A tagged flow rule instance **(b)** Parse graph of the forwarding model.

Figure 4: Processing logic of the deadlock-fix forwarding model

In the P4-based programmable data plane, the packet parsing process is abstracted as a finite state machine, where each state node represents a field in the packet header and the edges represent a transition between states. The parser sequentially traverses the packet headers to extract field values during execution, which are processed using a predefined match-action table. The parsing process of P4LoF is represented by the parser graph in Fig. 4b. From the graph, it can be seen that for the set of eliminated dependency chains $\Phi_{[\alpha,p]}$, the processing of the data packets in the network by each forwarding node on it contains two logics:

(1) **Processing of tagged traffic:** *swTag* stores the identification (7 bits) of the nodes on the dependency chains $\Phi_{[\alpha,\beta]}$, in a stack structure, and the corresponding bottom-of-stack flag bos (1 bit), which occupies n bytes (n is the number of nodes on $\Phi_{[\alpha,\beta]}$). When tagged traffic arrives at the respective tagged node, the deadlock-fix forwarding model, based on the preloaded forwarding rules, specifies the corresponding traffic to be forwarded according to the outport before the update and then pops up the current node identity. These *swTag* tags are transparent to the other untagged nodes.

(2) **Processing of untagged traffic:** the same process as for IP packets in traditional networks, forwarding based on Layer 3 routing or Layer 2 switching flow rules (solid blue line in Fig. 4b), thus ensuring that the introduction of the deadlock-fix forwarding model does not affect the normal communication of other traffic in the network that is not involved in the update.

3.3.3 Multi-Round Update Plan

Based on the above-proposed construction and processing methods of the dependency graph, we set the update rounds of the network configuration as follows.

Round 0: dependency-free nodes. According to Definition 1, inconsistent updates only cause forwarding loops on links where old and new paths overlap. If node w is solely on flow f's new path, its update cannot cause loops. P4LoF therefore schedules such nodes in update round 0, enabling simultaneous updates with round 1 without occupying additional rounds. Conversely, if w only belongs to the old path, directly updating its flow rules risks packet loss (blackholes) during the update for traffic arriving at this node. Finally, isolated nodes (unconnected to others in the dependency graph) also lack dependencies and can be updated concurrently with round 1 nodes.

Round 1 ~ O(m): nodes in the dependency graph. For the dependency graph substructure in Fig. 3b, we can get the general updating sequence by modeling it as a simple SCS problem and using the DP algorithm to solve it. Because of the tight dependency between nodes in the dependency chain, the general update plan is supposed to update one node per round. If the SCS is constructed of m characters (representing m nodes in the sequence), O(m) rounds may be needed to update those nodes in the dependency graph.

Rounds for multi-flow update deadlocks: none. By introducing additional forwarding rules to the loops that constitute the deadlock, we achieve sequential updating of each node in the deadlock while ensuring that each flow is forwarded without loops. The introduction of the additional forwarding mechanism destroys the original loop dependency structure of the deadlock and makes it degrade to a chain structure with weaker dependency among nodes, like Fig. 3a,b. Therefore, we can use the same method to handle these nodes and complete the update in the round 1 ~ O(m).

3.4 Complexity Analysis

The update mechanism of P4LoF mainly consists of three core components: a greedy-based dependency chain generating algorithm for single flow update, an SCS-based update sequence scheduling for multi-flow update, and a forwarding model for handling deadlock update sequences. In this section, we present a comprehensive time complexity analysis of the former two algorithms and a memory complexity analysis of the forwarding model, which is conducted with additional flow rules on the data plane.

3.4.1 Time Complexity Analysis of Single-Flow Dependency Generation

The dependency generating algorithm aims to create a minimal-length update dependency chain for a single flow f, ensuring that nodes are added to the chain as early as possible to prevent loops. For each node u, this algorithm invokes a loop-detection subroutine to check if adding the final forwarding path ($u^{fin} = v$) introduces a loop. In general, the time complexity is mainly relative to the network size, quantified by the number of nodes and edges in the graph $G = (V, E)$.

Loop-detection subroutine (Algorithm 1). Algorithm 1 performs a DFS-like traversal starting from v, marking node states to detect loops. In the worst case, DFS must visit all nodes and edges reachable from v. Let n, m denote the number of nodes and edges in the graph $G = (V, E)$ separately (i.e., $n = |V|, m = |E|$). The time complexity of Algorithm 1 is $O(n + m)$. Since edges are dynamically added into the graph G, the worst-case assumes a fully connected subgraph.

Main algorithm (dependency generating algorithm). In the second step of the main algorithm, we traverse all graph nodes and invoke Algorithm 1 for each. This results in $O(n(n+m))$ time, as each invocation could process the entire graph. Assuming the average dependency chain length is l, then the time complexity of single-flow dependency generation is $O(nl(n+m))$. In the worst case, an update dependency chain of length $O(n)$ is formed between n nodes. For example, the update scenario is to change the forwarding direction of a flow in the ring network topology. In this case, the time complexity of the main algorithm rises to $O(n^2(n+m))$.

Although the time complexity of our dependency generating algorithm is theoretically high, it shows efficient performance in practice since real-world topologies usually exhibit short dependency chains. For instance, the evaluation results in Section 4.1 show that over 95% of dependency chains terminate within 5 hops across various topologies. Therefore, the time complexity can be simplified to $O(n^2)$ in practice, where l is the average length of dependency chains ($l \ll n$).

3.4.2 Time Complexity Analysis of Multi-Flow Scheduling

For scheduling the multi-flow update sequences with the dependency graph structure shown in Fig. 3b, P4LoF formulates the problem as calculating the SCS of a series of sequences. SCS is known to have a polynomial time algorithm if the number of input sequences is constant, and to be NP-hard if the number of input sequences is not constant [20]. By utilizing a sliding window (fixed update sequence length), P4LoF reduces the computational complexity of resolving the SCS problem from NP-hard to polynomial time. For the k dependency chains of length l, the time complexity of resolving the corresponding SCS problem is $O(kl^k)$, which is polynomial, as long as the number of sequences (i.e., k) is constant [21]. Therefore, the time complexity of scheduling k dependency chains is $O(kw^k)$, where w is the sliding window size of P4LoF.

3.4.3 Memory Complexity Analysis of the Deadlock-Fix Forwarding Model

To address the deadlock dependency constructed by conflicting update sequences, we propose the P4-based Deadlock-fix forwarding model. Deadlock-fix forwarding model introduces constant-time P4 operations, with memory complexity bounded by $O(d)$ for d deadlocks in each window. Since each dependency deadlock is constructed by a series of dependency chains, the number of additional forwarding rules is linearly related to the sliding window size. Therefore, the memory complexity of the Deadlock-fix forwarding model is $O(dw)$.

4 Evaluation

In this section, we evaluate the effectiveness and performance of P4LoF based on real-world network topoologies. We implemented P4LoF with Python and P4. It currently supports deployment on the BMv2 model [22], a widely used software switch model for P4-based implementation. All experiments are conducted on LinuxOS (Ubuntu 20.04) with an Intel(R) Xeon(R) Gold 5218 CPU@2.30 GHz and 256 GB RAM. Our code is publicly available at https://github.com/Arcents/p4lof (accessed on 10 August 2025).

Experiment setup. The experiments use four public topologies, including Ebone and Sprintlink from Rocketfuel [23], and Nobel and Germany50 from SNDlib [24] (shown in Table 2). In each experiment, we first take all Equal-Cost Multi-Path (ECMP) routes to the destination nodes as the initial network configuration. Then 50% of the routing weights are randomly modified. Finally, the routing paths are updated again using the ECMP algorithm and set as the final configuration.

Table 2: Information of the adopted real-world topologies

Topology	Dst. nodes	Total nodes	Total edges	Average degree
Ebone	8	23	76	6.6
Germany50	10	50	176	7.0
Nobel	7	17	52	6.1
Sprintlink	21	44	166	7.5

Comparison methods. We evaluate P4LoF against state-of-the-art methods [6,8,25] that minimize control-data plane interactions. **GRD** [25] uses a heuristic based on solving the NP-hard problem of finding the maximum number of updatable switches per time slot to reduce control-data plane interactions. **RMS** [6] is A TCAM-free heuristic leveraging path reversal and merge optimization to minimize rule-update interactions while ensuring loop freedom. The work in [8] employs lazy cycle-breaking for loop-free updates. It has three variants: 1) **OPT**: Computes minimal-round schedules. 2) **LP_OPT**: Derives a theoretical lower bound using LP relaxation of the ILP. 3) **Rounding**: Develops a heuristic that rounds the fractional LP_OPT solution to achieve near-optimal schedules efficiently.

Evaluation metrics. Given N represents the number of flow rules to be updated. Let N_{add} denote the number of additional flow rules added by P4LoF, and N_{inter} represents the total number of control-data plane interactions. We choose the following metrics for evaluation:

(1) Memory overhead: $\frac{N_{add}}{N}$, the proportion of additional flow rules added by P4LoF.
(2) Communication overhead: $1 - \frac{N_{inter}}{N}$, the proportion of saved control-data plane interactions.
(3) Time consumption: the time for constructing the dependency graph, the core algorithm of scheduling the update plan, and the total processing time (including topology parsing time).

4.1 Single-Flow Update

We first evaluate the efficiency of P4LoF for single-flow update. Fig. 5 presents the experimental analysis of single-flow update dependency chains of P4LoF across four real-world topologies. The results demonstrate a sharp decline in dependency chain counts as chain length increases, demonstrating that the update dependencies of most flows are simple. Specifically, both Ebone (purple dashed line) and Nobel (blue solid line) exhibit near-zero dependency chains beyond length 6, while Germany50 (red solid line) achieves the same level at length 4. Sprintlink (green solid line) shows marginally longer chains due to its structural complexity yet maintains scalability within 5 hops. Notably, over 95% of chains terminate within ≤5 hops across all topologies. This ultra-short dependency profile confirms P4LoF's ability to minimize control-data plane interactions, directly addressing the operational efficiency and consistency challenges outlined in Section 2 while reducing the computation complexity of loop-free multi-flow update.

4.2 Multi-Flow Update

Fig. 6 presents comprehensive experimental results for multi-flow update across four network topologies, evaluating P4LoF's performance through three critical metrics.

Memory overhead. By integrating and optimizing the update sequence, P4LoF significantly reduces the number of flow rules that the switch needs to maintain during the update process. Additionally, to prevent deadlock when updating multiple flows, P4LoF installs a small number of extra forwarding rules in the switch. Fig. 6a quantifies this memory overhead as the ratio of additional flow rules to updated rules. As the sliding window size increases from 1 to 6, all topologies exhibit growth in memory overhead. Ebone

incurs the steepest rise, surging from 2.8% to 18.9%, while Germany50 follows a similar trajectory from 1.7% to 14.1%. Nobel and Sprintlink maintain relatively conservative growth, stabilizing at 12.6% and 11.3% respectively at window size 6. This confirms that larger sliding windows necessitate more temporary rules to resolve deadlocks, with hierarchical topologies (e.g., Ebone) demanding higher TCAM utilization.

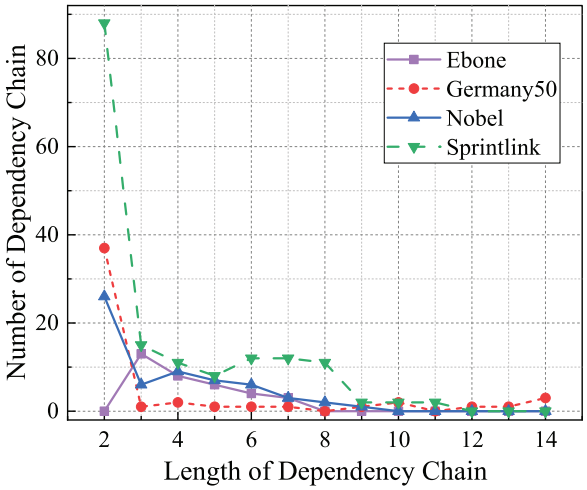
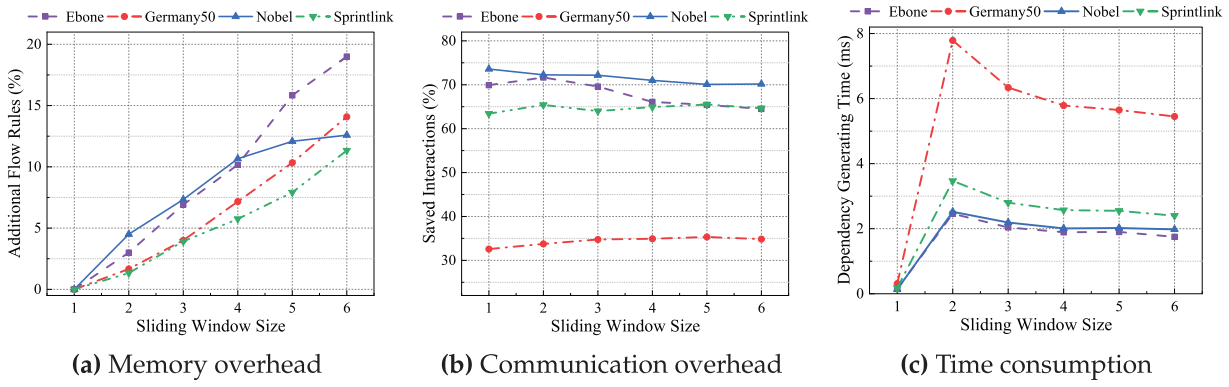


Figure 5: Experiment results of single-flow update



(a) Memory overhead **(b)** Communication overhead **(c)** Time consumption

Figure 6: Experiment results of multi-flow update

Communication overhead. Fig. 6b measures saved control-data plane interactions. Nobel consistently achieves the highest savings (70.1%–73.6%), demonstrating robust independence from window scaling. Ebone and Sprintlink maintain 63%–71% savings across all windows, with minor fluctuations linked to routing complexity. Germany50 shows the lowest but stable efficiency (32.6%–35.3%), indicating topology-dependent coordination costs. Notably, all topologies sustain 32.6% interaction savings—validating P4LoF's core advantage in minimizing control-plane overhead. For the communication overhead in the data plane, the deadlock-fix forwarding model requires the addition of *swTag* tags containing the deadlock node identifier in the packet header. According to our design of *swTag* in Section 3.3.2, the length of each tag is only 1 byte. Experiments in various topologies show that the length of a single dependency chain does not exceed 3 in 90% of cases, and the corresponding deadlock-fix forwarding model adds no more than 3 bytes to the length of each tag in the packet header. In general, the packet size in normal network communication

is not less than 64 bytes, then the additional bandwidth overhead added by the tag in the data plane is not more than 5% and will not affect the normal network communication.

Time consumption The execution time of the update algorithm should be considered, especially when the network has a low tolerance for update latency. Fig. 6c shows the relationship between window size and dependency generating time consumption. Germany50 exhibits the largest time consumption (<7.8 ms) among all topologies due to its high topology complexity. Other topologies maintain consistently low latency (1.7–3.4 ms) across all window sizes and stabilize below 2.58 ms when the window size exceeds 4. P4LoF uses a greedy algorithm to construct the shortest length dependency chain between update nodes, which simplifies the computational complexity of subsequent update sequences.

Table 3 presents the time consumption of the core algorithm (i.e., resolving SCS and update dependencies) and total processing time. For the Germany50 topology with the highest network complexity, P4LoF can give the update sequence in 14.93 s in 90% of the experiments, including time consumption for parsing topologies. Meanwhile, for other network topologies with relatively simple structures, P4LoF can compute the node update sequence in 1.87 s. We also found that the core algorithm of P4LoF operates very quickly, taking less than 149 ms. However, parsing the network topologies requires significantly more time, leading to an overall time consumption measured in seconds. As can be seen in Table 3, for 90% of the experiments performed on topology Germany50, the execution time of the core algorithm is within 100 ms. For other network topologies, the core algorithm is able to give the update sequence for the multi-flow update within 28 ms, which is only 1.50% of the total running time of the update algorithm.

Table 3: The execution time of P4LoF

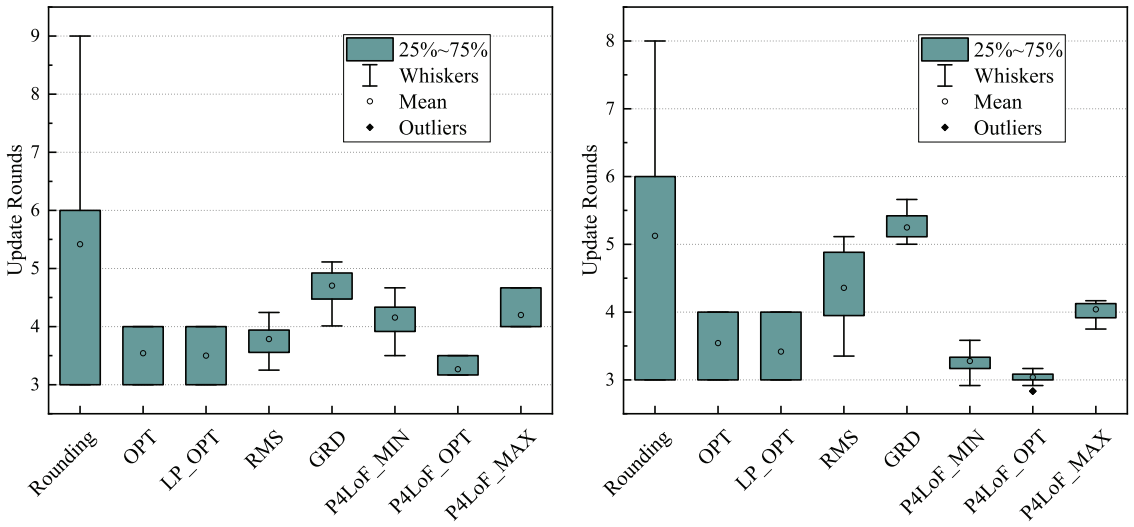
Topology	Core algorithm (ms)	Total (s)	Ratio
Germany50 (median)	26	6.98	0.37%
Germany50 (90%)	100	14.93	0.67%
Germany50 (max)	149	18.49	0.81%
others (median)	8	0.61	1.31%
others (90%)	20	1.38	1.45%
others (max)	28	1.87	1.50%

4.3 Update Rounds Comparison

To evaluate the update efficiency of P4LoF, we conduct comparative experiments with state-of-the-art methods across various topology scales. Fig. 7 displays the experimental results for the simplest topology (Nobel) and the most complex topology (Sprintlink), respectively.

Experiment results in Fig. 7a illustrate significant performance variations among the evaluated schemes in the simple topology. The Rounding heuristic shows the highest median update rounds at 5.5, accompanied by substantial data dispersion, as indicated by its elongated whiskers and the presence of outliers. This suggests an unpredictable interaction overhead. In contrast, the P4LoF variants (MIN, MAX, and OPT denote adopting min/max/optimal sliding windows) achieve notably lower median values of 4.5, with compact interquartile ranges and minimal data spread, thereby highlighting their stable and consistent performance. Other methods, including GRD, RMS, OPT, and LP_OPT, cluster around similar quartiles with medians ranging from 3.5 to 4.5. However, their slightly wider distributions suggest marginally less consistency compared to the P4LoF variants. Overall, P4LoF demonstrates a robust efficiency in minimizing interactions in the control-data plane, with its sliding-window optimizations yielding reliable outcomes.

For the complex topology (Fig. 7b), the advantages of P4LoF become even more pronounced. The Rounding method again demonstrates the poorest performance, with a higher median of approximately 6.3 and a broader data spread. In contrast, P4LoF_MIN and P4LoF_OPT achieve the most efficient results, consistently maintaining minimal mean interaction counts of around 3.8. All P4LoF variants exhibit remarkable data homogeneity, reinforcing the method's adaptability to intricate network structures. These patterns confirm that P4LoF's optimal sliding-window configuration and deadlock-fix model significantly reduce interaction overhead, even in demanding topologies, outperforming both heuristic methods (GRD, RMS) and theoretical baselines (LP_OPT).



(a) Update rounds in Nobel **(b)** Update rounds in Sprintlink

Figure 7: Update efficiency among varied topology scales

5 Related Work

For loop-free consistency updating in programmable networks, the proposed research can be classified into three categories: two-phase update, ordering-based update, and time-based update.

5.1 Two-Phase Update

Reitblatt et al. [5] first proposed a two-phase update (TP) scheme: By adding tags to each packet in a network, implement packets only according to the updated configuration before or after the flow rule configuration for forwarding, and will not appear at the same time, following the two configurations. This scheme makes programmers only need to focus on network status before and after the configuration updates, and have no need to worry about intermediate states that exist in violation of the consistency. The work in [7] further improved the scheme, eliminating the operation of adding labels to data packet headers, thus improving the network update efficiency. These update solutions can guarantee loop-free consistency in every intermediate state during network updating. However, such approaches and many subsequent studies based on them [18] have an inherent drawback: additional memory resource consumption (e.g., TCAM) is required to maintain both the old and new flow rules configuration at the same time [6]. Besides, since the old path is available in those approaches by default, they are not available for the update scenario with network link or node failure.

5.2 Ordering-Based Update

The ordering-based (OR) update solution converts the initial configuration of the data plane to the final configuration by calculating a specific update sequence of all the updating flow rules. Mahajan and Wattenhofer [26] first proposed a destination-based single-flow loop-freedom update scheme that emphasizes the inherent balance between the strength of consistency and the dependency of flow rules of different switching nodes. Unlike the TP updating plan, which adds flow rules to nodes, Ordering-based Update approaches [6,8,9] do not consume storage resources, but they also introduce a computational time bottleneck. Finding the fastest update solution was proved to be an NP-complete problem in [13]. Räcke et al. [8] propose a lazy cycle-breaking strategy which, by adding constraints lazily, dramatically improves update performance. Two strengths of loop-free updates are considered: strong loop-free consistency (the network is loop-free at any update stage) and relaxed loop-free consistency (only the forwarding path between source and destination nodes is required to be loop-free). It is shown that relaxed consistency can significantly improve the speed of network updates.

5.3 Time-Based Update

Time-based update approaches [10,11] provide the implementation flow consistency for the updating of new ideas. Such schemes utilize clock synchronization between switches to achieve a fine-grained arrangement of the updating process. These update plans can efficiently address the inconsistent update problems like forwarding loops, blackhole, and transient congestion [27]. However, this paradigm has extremely high requirements on the precision of clock synchronization between switches and is not applicable to actual deployment [28].

6 Conclusion

To address the forwarding loop inconsistencies during network configuration updates, this paper proposes an efficient loop-free update approach, P4LoF. It enables simultaneous rerouting of multiple flows with minimal control-data plane interactions. P4LoF constructs optimized update sequences by generating shortest per-flow dependency chains, merging them into a multi-flow update dependency graph, and solving it as an SCS problem. Furthermore, to address the multi-flow update deadlock dependencies, P4LoF builds a deadlock-fix model that provides consistent forwarding control based on the programmable data plane. Experimental results demonstrate that P4LoF can reduce the number of control-data plane interactions during updates by ≥32.6% with modest computation and memory overhead, significantly improving the update efficiency. Moreover, the sliding window mechanism of P4LoF is essentially a threshold-based solution whose effectiveness relies on thresholds setup. Further research could explore the mechanism of dynamic and adaptive thresholds configuration to improve scalability among different network conditions.

Acknowledgement: Not applicable.

Funding Statement: This work was supported by the National Key Research and Development Program of China under Grant 2022YFB2901501; and in part by the Science and Technology Innovation leading Talents Subsidy Project of Central Plains under Grant 244200510038.

Author Contributions: The authors confirm contribution to the paper as follows: Conceptualization, Jiqiang Xia and Jianhua Peng; methodology, Jiqiang Xia and Qi Zhan; formal analysis, Yuxiang Hu; investigation, Jianhua Peng; writing—original draft preparation, Jiqiang Xia; writing—review and editing, Qi Zhan and Le Tian; supervision, Yuxiang Hu and Jianhua Peng; funding acquisition, Yuxiang Hu and Le Tian. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: The data that support this study are available from authors.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest to report regarding the present study.

References

1. Le M, Huynh-The T, Do-Duy T, Vu TH, Hwang WJ, Pham QV. Applications of distributed machine learning for the internet-of-things: a comprehensive survey. IEEE Commun Surv Tutor. 2025;27(2):1053–100.
2. McKeown N, Anderson T, Balakrishnan H, Parulkar G, Peterson L, Rexford J, et al. OpenFlow: enabling innovation in campus networks. SIGCOMM Comput Commun Rev. 2008;38(2):69–74.
3. Su J, Wang W, Liu C. A survey of control consistency in software-defined networking. CCF Trans Netw. 2019;2(3):137–52. doi:10.1007/s42045-019-00022-w.
4. Deb R, Roy S. A comprehensive survey of vulnerability and information security in SDN. Comput Netw. 2022;206(11):108802. doi:10.1016/j.comnet.2022.108802.
5. Reitblatt M, Foster N, Rexford J, Schlesinger C, Walker D. Abstractions for network update. In: Proceedings of the ACM SIGCOMM, 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '12. New York, NY, USA: Association for Computing Machinery; 2012. p. 323–34.
6. Dolati M, Khonsari A, Ghaderi M. Minimizing update makespan in SDNs without TCAM overhead. IEEE Trans Netw Serv Manag. 2022;19(2):1598–613. doi:10.1109/tnsm.2022.3146971.
7. Namjoshi KS, Gheissi S, Sabnani K. Algorithms for in-place, consistent network update. In: Proceedings of the ACM SIGCOMM, 2024 Conference, ACM SIGCOMM '24. New York, NY, USA: Association for Computing Machinery; 2024. p. 244–57.
8. Räcke H, Schmid S, Vintan R. Fast algorithms for loop-free network updates using linear programming and local search. In: IEEE INFOCOM 2024—IEEE Conference on Computer Communications; 2024 May 20–23; Vancouver, BC, Canada. p. 1930–9.
9. Gao X, Majidi A, Gao Y, Wu G, Jahanbakhsh N, Kong L, et al. Nous: drop-freeness and duplicate-freeness for consistent updating in SDN multicast routing. IEEE/ACM Trans Netw. 2024;32(5):3685–98. doi:10.1109/tnet.2024.3404967.
10. Hasan KF, Feng Y, Tian YC. Precise GNSS time synchronization with experimental validation in vehicular networks. IEEE Trans Netw Serv Manag. 2023;20(3):3289–301. doi:10.1109/tnsm.2022.3228078.
11. He X, Zheng J, Dai H, Sun Y, Dou W, Chen G. Chronus+: minimizing switch buffer size during network updates in timed SDNs. In: IEEE 40th International Conference on Distributed Computing Systems; 2020 Jul 8–10; Singapore. p. 377–87.
12. Larsen KG, Mariegaard A, Schmid S, Srba J. AllSynth: a BDD-based approach for network update synthesis. Sci Comput Program. 2023;230(C):102992. doi:10.1016/j.scico.2023.102992.
13. Henzinger M, Paz A, Pourdamghani A, Schmid S. The augmentation-speed tradeoff for consistent network updates. In: Proceedings of the Symposium on SDN Research, SOSR '22. New York, NY, USA: Association for Computing Machinery; 2022. p. 67–80.
14. Naeem MA, Ud Din I, Meng Y, Almogren A, Rodrigues JJPC. Centrality-based on-path caching strategies in NDN-based internet of things: a survey. IEEE Commun Surv Tutor. 2025;27(4):2621–2657. doi:10.1109/COMST.2024.3493626.
15. He X, Zheng J, Dai H, Zhang C, Li G, Dou W, et al. Continuous network update with consistency guaranteed in software-defined networks. IEEE/ACM Trans Netw. 2022;30(3):1424–38. doi:10.1109/tnet.2022.3143700.
16. Györgyi C, Larsen KG, Schmid S, Srba J. SyPer: synthesis of perfectly resilient local fast re-routing rules for highly dependable networks. In: IEEE INFOCOM 2024—IEEE Conference on Computer Communications; 2024 May 20–23; Vancouver, BC, Canada. p. 2398–407.
17. Bosshart P, Daly D, Gibb G, Izzard M, McKeown N, Rexford J, et al. P4: programming protocol-independent packet processors. ACM SIGCOMM Comput Commun Rev. 2014;44(3):87–95.

18. Zhou Z, He M, Kellerer W, Blenk A, Foerster KT. P4Update: fast and locally verifiable consistent network updates in the P4 data plane. In: Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies, CoNEXT '21. New York, NY, USA: Association for Computing Machinery; 2021. p. 175–90.
19. Pang Z, Huang X, Li Z, Zhang S, Xu Y, Wan H, et al. Flow scheduling for conflict-free network updates in time-sensitive software-defined networks. IEEE Trans Ind Inf. 2021;17(3):1668–78. doi:10.1109/tii.2020.2998224.
20. Dudycz S, Ludwig A, Schmid S. Can't touch this: consistent network updates for multiple policies. In: 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks; 2016 Jun 28–Jul 1; Toulouse, France. p. 133–43.
21. Basta A, Blenk A, Dudycz S, Ludwig A, Schmid S. Efficient loop-free rerouting of multiple SDN flows. IEEE/ACM Trans Netw. 2018;26(2):948–61. doi:10.1109/tnet.2018.2810640.
22. P4-Language-Consortium. Behavioral model repository; 2014 [Internet]. [cited 2025 Jan 4]. Available from: https://github.com/p4lang/behavioral-model.
23. Spring N, Mahajan R, Wetherall D, Anderson T. Measuring ISP topologies with Rocketfuel. IEEE/ACM Trans Netw. 2004;12(1):2–16. doi:10.1109/tnet.2003.822655.
24. Orlowski S, Wessäly R, Pióro M, Tomaszewski A. SNDlib 1.0—survivable network design library. Networks. 2010;55(3):276–86. doi:10.1002/net.20371.
25. Amiri SA, Ludwig A, Marcinkowski J, Schmid S. Transiently consistent SDN updates: being greedy is hard. In: Structural information and communication complexity. Cham, Switzerland: Springer; 2016. p. 391–406. doi:10.1007/978-3-319-48314-6_25.
26. Mahajan R, Wattenhofer R. On consistent updates in software defined networks. In: Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks, HotNets-XII. New York, NY, USA: Association for Computing Machinery; 2013. p. 1–7.
27. Ceylan E, Chatterjee K, Schmid S, Svoboda J. Congestion-free rerouting of network flows: hardness and an FPT algorithm. In: 2024 IEEE Network Operations and Management Symposium; 2024 May 6–10; Seoul, Republic of Korea. p. 1–7.
28. Chefrour D. Evolution of network time synchronization towards nanoseconds accuracy: a survey. Comput Commun. 2022;191(1):26–35. doi:10.1016/j.comcom.2022.04.023.