Cupid: Congestion-free Consistent Data Plane Update In Software Defined Networks

Wen Wang*, Wenbo He*, Jinshu Su†, Yixin Chen*
*School of Computer Science, McGill University, Montreal, QC, Canada
†School of Computer, National University of Defense Technology, Changsha, Hunan, China
Email: wen.wang4@mail.mcgill.ca, wenbohe@cs.mcgill.ca, sjs@nudt.edu.cn, yixin.chen@mail.mcgill.ca

***Abstract*—With the popular applications of SDN in load balancing and failure recovery, the controller schedules affected flows to redundant paths to avoid network congestions and failures by updating flow tables in data plane. However, inconsistent flow table updating may lead to transient incorrect network behaviors or undesired performance degradation. Therefore, the consistency imposes dependencies among updates, so that the order of updates must be carefully considered to keep the consistency. To update flow tables consistently and efficiently, in this paper, we propose an update ordering approach – Cupid. To avoid high overhead in update ordering, we divide the global dependencies among updates into local restrictions by: 1) partitioning a new routing path into several independent segments, 2) identifying critical nodes controlling traffic shifting between the old path and new path, and 3) constructing a dependency graph among critical nodes for potential congested links. We then design a heuristic algorithm to resolve the dependency graph. To save the flow table space, a switch keeps only one flow entry with multiple ports for a flow during updating. Our simulation shows that Cupid schedules updates at least 2 times faster and has less throughput losses than the state-of-the-art approaches in both fat-tree and mesh networks.**

I. Introduction

Software defined networking has been widely applied for traffic engineering [1], [2] and failure recovery [3] with its global view. The SDN controller schedules affected flows to other available paths and updates flow tables in concerned switches for load balancing or failure recovery. Although plenty of researches [4], [5], [6], [7] have been carried out to compute optimized routing paths based on current network topology and traffic distribution to protect against failures and congestions, a common challenge faced in all centrally-controlled networks is updating the data plane consistently and efficiently [8].

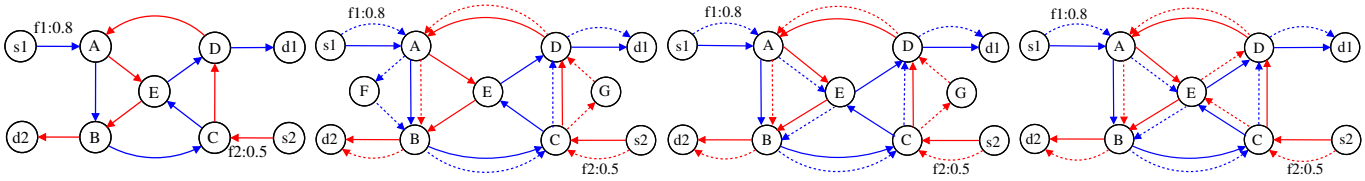
The consistency requires that flows are migrated to new routing paths seamlessly, never with loops nor black-holes during flow tables updating, which imposes dependencies among rules in flow tables along a routing path [9]. The missing or mismatch of forwarding rules in switches may interrupt a flow for a while. Therefore, the ordering of updating flow tables in concerned switches must be carefully considered. Moreover, in networks where communications usually have sensitive performance requirements, such as data centers, a stronger consistency instructs that traffic should not exceed link bandwidth capacity during updating, i.e., congestion-free consistency [10]. With limited bandwidth resource of links, even though the bandwidth demands of flows are satisfiable before and after data plane updating, flows may be rerouted to a link before offloading original flows on the link, which may result in congestions and throughput degradation during updating. Therefore, the congestion-free consistency further poses more dependencies among updates.

To update flows to new routing paths without any performance degradation, [10], [11] formulates the problem as LP (Linear Program) to find a transition sequence from the initial state to the target state. However, this approach would be quite slow and does not scale to large networks with a large number of flows. Heuristic approaches trying to find an updating order to resolve dependencies among updates, e.g., [8], [12], also suffer substantial overhead due to the high dependency complexity. Meanwhile, to ensure the loop free and black-hole free consistency of each flow, two-phase update [13] is proposed which forwards packets either with the new path or the old path, but never with a mixed path. Unfortunately, the atomic commit adds to the complexity of dependencies among updates. As congestions may occur on any hop of a new routing path during updating, to avoid throughput degradation, a flow only could migrate to the new path until all these hops have enough available bandwidth, which brings down updating efficiency. Moreover, a flow table may hold multiple entries of different versions for each flow in two-phase update. This may overload the limited flow table space. Attempts to reduce flow table space overhead during updating have been made in [14], [15], but they always have to make trade-offs between flow table space and updating efficiency.

In this paper, we present a congestion-free update ordering approach while maintaining the black-hole free, loop free consistency properties. We firstly find that the new routing path of each flow could be divided into several independent segments, and identify the critical nodes which control traffic shifting between the old path and new path. To avoid congestions during updating, instead of resolving a global dependency graph composed of updates and network resources in [8], we divide dependencies among updates into local dependencies among critical nodes, and then construct a dependency graph with potential congested links. The divided local dependencies improve update parallelism, and ensure the

The authors' research was supported by the fund FRQ-NT NC-182928.

978-1-4673-9953-1/16/$31.00 ©2016 IEEE



(a) Initial State (b) Target State 1 (c) Target State 2 (d) Target State 3

Fig. 1: Routing update of flows: (a) shows the current routing paths (solid lines) of flows f_1 (blue from s_1 to d_1) and f_2 (red from s_2 to d_2), and then f_1 and f_2 may be rerouted to new paths (dotted lines) in (b), (c), (d) respectively with three different routing schemes. The bandwidth capacity of each link is 1 unit, and throughputs of f_1 and f_2 are 0.8 unit and 0.5 unit.

efficiency and scalability of congestion-free updating. In this way, we successfully restrict the problem space while keeping the dependency relationships. We then design and implement a heuristic dependency resolution algorithm. Meanwhile, to reduce the flow table space overhead, we use multiple input and output ports with weights in each flow entry, so that a switch keeps only one rule for each flow during updating. The results of simulation show that Cupid schedules update ordering at least 2 times faster than Dionysus [8] and has less throughput losses in both fat-tree and mesh topologies.

The rest of the paper is organized as follows: Section II checks the challenges of congestion-free consistent updating. Section III describes critical nodes and segment partition for each flow. Section IV constructs the dependency graph with critical nodes. Section V presents the dependency resolution algorithm. In Section VI, we evaluate Cupid and compare with other approaches. Finally, Section VII concludes the paper.

II. Challenges And Related Work

A. Complexity of Congestion-free Updating

To achieve high network utilization during flow tables updating, [10], [11] formulates the updating problem as a LP problem to find a transition sequence from the initial state to the target state for inter-data center WANs and inside data center networks respectively. However, real-world networks are usually more complicated than the well organized inter- and intra- data center networks. A more complex network tends to involve stronger dependency among updates and even dependency deadlocks which complicates the update ordering. In Figure 1a, flows f_1 and f_2 are rerouted to other paths to release more available bandwidth for future flows on links $A \to B$, $C \to E$ and $E \to D$. With different rerouting schemes, the target routing paths (dotted lines) are different in Figure 1b, 1c, 1d, which results in differences in update ordering. Moreover, the three updating scenarios vary in dependency complexity.

In Figure 1b, when flow f_1 is rerouted from subpath $C \to E \to D$ to $C \to D$, it has to wait for the removal of f_2 on link $C \to D$ due to the bandwidth limitation. Meanwhile, the rerouting of f_2 from subpath $A \to E \to B$ to $A \to B$ depends on the remove of f_1 on link $A \to B$. As f_1 could update from $A \to B$ to $A \to E \to B$, and f_2 could also shift from $C \to D$ to $C \to G \to D$ freely without any bandwidth restriction, the dependencies are released. Thus, we can schedule a feasible update order to solve the dependencies.

In Figure 1c, when f_1 updates from subpath $A \to B$ to $A \to E \to B$, it requires moving f_2 from $A \to E \to B$ to $A \to B$ firstly. However, due to the bandwidth restriction, it is impossible to make the exchange without any performance reduction with single path forwarding. Therefore, [8], [10], [11] use multipath to migrate flows gradually to new paths, e.g., splitting f_1 into 0.5 unit and 0.3 unit on subpaths $A \to E \to B$ and $A \to B$ respectively, thus f_2 could shift to $A \to B$ completely with 0.5 unit, and then f_1 finally migrates 0.3 unit on the old subpath to $A \to E \to B$. Unfortunately, in this scenario, if we split f_1 into two subflows with $A \to B$ and $A \to E \to B$ before updating the rule in C from $C \to E$ to $C \to D$, we will get a loop $\{E, B, C\}$ on $f_1's$ routing path.

Despite of the multipath transition with $A \to B$ and $A \to E \to B$, Figure 1d also requires multipath transition with $C \to D$ and $C \to E \to D$ for f_1 and f_2. Furthermore, to avoid loop $\{E, B, C\}$, the removal of rule $C \to E$ for f_1 in switch C requires the multipath transition of f_1 and f_2 with $C \to D$ and $C \to E \to D$, which further requires the removal of rule $A \to E$ for f_2 in switch A to avoid loop $\{E, D, A\}$, so that a deadlock occurs between the two multipath transitions.

B. Efficiency Of Update Ordering

Although congestion-free updating is usually complicated in real-world networks, updates should react in time to routing changes to minimize the duration of performance degradation and network failures [16]. However, maintaining congestion-free consistency during updating usually requires global coordination due to complex dependencies among updates, which poses challenges for update ordering efficiency. LP [10], [11] would be too slow to find a feasible ordering with the complex dependency. Dionysus [8] proves that finding the fastest update scheduling is a NP-complete problem, and dynamically schedules a dependency graph among updates and network resources with a heuristic algorithm. However, with the complicated topology and strong dependencies in real-world networks, the global dependency graph coordination is of great overhead. [10], [17] also note that careful ordering of updates cannot always guarantee congestion freedom during updating. [16], [18] reduce the consistency problem to a model checking problem instead of designing a new ordering algorithm.

In spite of the low efficiency due to the global coordination overhead, the consistency in two-phase update [13] requires never forwarding packet with a mixed path, so that all the switches in a new path must be updated before shifting any traffic of a flow to the new path. Thus, the strong property of consistency would lead to a long delay for rerouting update. Moreover, we must make sure neither the new path nor the old path is congested, while any hop on a routing path may be exposed to congestions during updating. To improve the update efficiency, [9] notes that dependency structures are simpler for weaker consistency properties than stronger properties, and makes a trade-off between the strength of consistency property and dependencies.

III. Independent Segments of A Single Flow

A. Consistency With Mixed Path

Considering the low efficiency of the strong consistency, we note that the black-hole free and loop free consistency could still be preserved with mixed paths without the atomic committing in two-phase update, e.g., a packet of flow f_1 in Figure 1b going through $s_1 \to A \to B \to C \to D \to d_1$ or $s_1 \to A \to F \to B \to C \to E \to D \to d_1$ with mixed path. [16] notes that black-hole free and loop free consistency is a downstream-dependent property, updating from downstream to upstream switches along the routing path is sufficient to ensure it (Lemma 1).

Lemma 1. *The reverse order updating of a routing path is black-hole free and loop free.*

With Lemma 1, upstream switches always have to wait for the update completion of downstream switches in reverse order updating. Actually, some upstream switches are independent from downstream updates as long as connectivity maintains. In Figure 1b, switch A does not need to know the downstream path of C is $C \to E \to D \to d_1$ or $C \to D \to d_1$ for flow f_1 provided the connectivity between C and d_1. Similarly, C also does not care about the upstream path of flow f_1 is $s_1 \to A \to F \to B$ or $s_1 \to A \to B$ as long as packets arrive at C correctly. In other words, the updates of flow f_1 to subpaths $A \to F \to B$ and $C \to D$ are independent.

To understand how switches migrate a flow from its old routing path to a new routing path, we identify the critical nodes which control the routing path switching. The critical nodes $C(f)$ of a flow f are the common switches $\{n_f\}$ on both the old path $P_o(f)$ and the new path $P_n(f)$, but using different rules.

$$C(f) = \{n_f | n_f \in P_o(f) \cap P_n(f), rule(n_f, P_o(f)) \neq rule(n_f, P_n(f))\}$$

As critical nodes connect the new path with the old path, such as switches A, E, B on f_1's routing paths in Figure 1c, these critical nodes control the traffic shifting between the old path and new path. Therefore, the updates of critical nodes are of great importance during updating. Based on the modifications made to flow entries, critical nodes could be divided into three classes: $(inport, output)$ in which both the input and output ports require to be modified (e.g., E for f_1), $(*, output)$ changes the output port of flow entry while the input port stays unchanged (e.g., A for f_1), and $(inport, *)$ only changes the input port (e.g., B for f_1).

To save flow table space, each flow entry is equipped with multiple input or output ports during updating in critical nodes. Thus, there is only one flow entry kept for each flow in a switch. Especially for the critical nodes which require output port modifications, these nodes control how to forward packets through mix paths with multiple $outport$s. Each $outport$ is associated with a weight, so that these weights determine the amount of traffic on the new path and old path to avoid congestions during updating.

B. Segment Partition of A flow

To make the updating flexible and reduce the updating latency, we divide the new routing path of flow f into several segments $S(f)$ and each segment could be updated independently. Each segment $s \in S(f)$ consists of a minimum sequence of switches on the new path which starts and ends with critical nodes (or the source and destination switches for the first and last segments), so that the beginning and ending nodes control traffic switching between the old subpaths and new segments. Adjacent segments only share the ending or beginning nodes and never overlap on nodes inside segments. Thus, each segment is an atomic sequence.

$$\begin{aligned} S(f) = \{ & s \subseteq P_n(f) | \min\{s.length\}, \\ & s.start, s.end \in C(f) \cup \{P_n(f).start, P_n(f).end\}, \\ & \forall s' \in S(f), s \cap s' = \varnothing \ or \ = s.start \equiv s'.end \\ & \ or \ = s.end \equiv s'.start, \\ & lasthop(s) \notin Circle(P_o(f) \cup P_n(f))\} \end{aligned}$$

As the beginning and ending nodes of each segment are critical nodes, they always need to update either input port or output port. Even though a beginning node may require updating the input port, to make each segment independent, the updating of the beginning node in a segment only modifies the output port field $(*, outport)$ of the flow entry, while the modification of the input port $(inport, *)$ belongs to the ending node of last segment. Similarly, the updates of input and output ports in an ending node are also divided into two segments.

To avoid loops during updating, the last hop of a segment should not belong to any circle formed by the old path and new path. If there is a segment of flow f_1 in Figure 1c ending with $\to E \to B \to C$ which is involved in the circle $\{E, B, C\}$, packets will be circulated if we update the routing path to $E \to B \to C$ in the segment before removing the rule $C \to E$ in C which belongs to the next segment. Hence, the circle makes the updating of $E \to B \to C$ depends on updating f_1 to $C \to D$. Thus, $C \to D$ should be added to the end of the segment to break the circle, otherwise the updating of the segment depends on the next segment.

We design an algorithm to calculate segments set $S(f)$ for each flow f in Algorithm 1. Non-critical nodes and the nodes belonging to any circle are added to a segment

Algorithm 1 Segment Partition

1: $S(f) = \varnothing$
2: **for** $n : P_n(f)$ **do**
3: $s = n$
4: $m =$ successor node of n on $P_n(f)$
5: **while** $m \neq \varnothing \wedge (m \notin C(f)$ *or* $(n \rightarrow m) \in Circle(P_o(f) \cup P_n(f)))$ **do**
6: $s = s \rightarrow m$
7: $n = m,\ m =$ successor node of n on $P_n(f)$
8: **end while**
9: $s = s \rightarrow m$
10: $S(f) = S(f) \cup s$
11: $n = m$
12: **end for**

until meeting a critical node breaking the circle in lines 5-8. $Circle(P_o(f) \cup P_n(f))$ is the circle set formed by the old path and new path, such as circles $\{E, B, C\}$ formed by f_1's routing paths and $\{E, D, A\}$ formed by f_2's routing paths in Figure 1d. By traversing switches along a new path (line 2), any switch on the new path is assigned to its segment.

C. Properties of Segments

Theorem 1. *A segment $s \in S(f)$ is black-hole free and loop free with reverse order updating.*

Proof. As $s.start, s.end$ are the common nodes of the new and old paths, we construct a flow f' from $s.start$ to $s.end$ with the same header of f. The routing update of f' follows Lemma 1 with reverse order updating, thus the black-hole free and loop free properties preserve in s. □

Theorem 2. *Segments in $S(f)$ are independent from each other, which means updates in a segment do not depend on updates in other segments.*

Proof. If the updating of switch $s(i)$ in segment s depends on another segment $s' \in S(f)$, the updating of $s(i)$ before updates in s' will result in a loop l or black hole b.

Switch $s(i)(0 \leqslant i < s.length - 1)$ updates the rule $rule(s(i), P_o(f))$ to $rule(s(i), P_n(f))$ which establishes a path $s(i) \rightarrow s(i+1) \in s$. As $s(i) \notin s'$, $s(i) \rightarrow s(i+1) \notin s'$, which means no path changes in segment s'. Thus, $l, b \notin s'$. Meanwhile, segment s is black-hole free and loop free with reverse order updating (Theorem 1), which conflicts with l, b. Therefore, $s(i)$ does not depend on any node in s'. □

With Theorem 1 and 2, each segment acts as an independent routing path with reverse order updating. Thus, segments could be updated in parallel to improve updating efficiency. During updating of independent segments, packets of a flow f may be forwarded along a path composed of mixed nodes belonging to $P_o(f)$ or $P_n(f)$, while the connectivity always maintains.

According to Algorithm 1, the segments of flows in Figure 1b,1c,1d are showed in Table I. Although segments start and end with critical nodes, a lot of segments do not need any update due to the divided updating of $inport$ and $outport$ between beginning and ending nodes. For instance, the segment $s_1 \rightarrow A$ of flow f_1 ends with a critical node A in Figure 1b. Switch A only needs to update the $outport$ field for f_1 with the beginning node in the next segment $A \rightarrow F \rightarrow B$, so that no node requires updating in segment $s_1 \rightarrow A$. Therefore, segments requiring no update are shadowed in Table I, such that only partial segments require updates, which further reduces updating overhead.

TABLE I: Segments of flows

Figure	Segments of f_1	Segments of f_2
1b	$\{s_1 \rightarrow A, A \rightarrow F \rightarrow B, B \rightarrow C, C \rightarrow D, D \rightarrow d_1\}$	$\{s_2 \rightarrow C, C \rightarrow G \rightarrow D, D \rightarrow A, A \rightarrow B, B \rightarrow d_2\}$
1c	$\{s_1 \rightarrow A, A \rightarrow E, E \rightarrow B \rightarrow C \rightarrow D, D \rightarrow d_1\}$	$\{s_2 \rightarrow C, C \rightarrow G \rightarrow D, D \rightarrow A, A \rightarrow B, B \rightarrow d_2\}$
1d	$\{s_1 \rightarrow A, A \rightarrow E, E \rightarrow B \rightarrow C \rightarrow D, D \rightarrow d_1\}$	$\{s_2 \rightarrow C, C \rightarrow E, E \rightarrow D \rightarrow A \rightarrow B, B \rightarrow d_2\}$

IV. Congestion-free Updating of Multiple Flows

Topology changes and load balancing usually require rerouting multiple flows to other available paths in a short time. To ensure performance of flows during updating, we have to schedule a feasible congestion-free updating order.

A. Potential Congested Links During Updating

To avoid congestions during updating, we have to discover potential congestions at first. We define the criteria to identify a potential congested link l between switch u and v as follow:

Definition 1 (Potential Congested link $l = u \rightarrow v$)**.** *With flows desire to use the link $F_n(l) = \{f \mid \forall f, l \in P_n(f) - P_o(f)\}$, flows to be moved away from the link $F_o(l) = \{f \mid \forall f, l \in P_o(f) - P_n(f)\}$, and unchanged flows going through this link $F_u(l) = \{f \mid \forall f, l \in P_n(f) \cap P_o(f)\}$ during updating, the throughputs of flows $b(f)$ on a potential congested link l satisfy*

$$\sum_{f_i \in F_o(l) \cup F_u(l)} b(f_i) \leq c(l), \quad \sum_{f_i \in F_n(l) \cup F_u(l)} b(f_i) \leq c(l) \quad (1)$$

$$\sum_{f_i \in F_o(l)} b(f_i) + \sum_{f_i \in F_n(l)} b(f_i) + \sum_{f_i \in F_u(l)} b(f_i) > c(l) \quad (2)$$

The consumed bandwidth does not exceed the bandwidth capacity $c(l)$ in both the initial and final states (1). However, flows in $F_n(l)$ may be scheduled to link l at any time before moving some old flows in $F_o(l)$ away during updating, which exceeds the link bandwidth capacity (2). To avoid any throughput degradation, we must find all potential congestion combinations of $F_n(l)$ and $F_o(l)$ on link l. With Definition 1, we can find a set of potential congested links CL which contains all the links that may be congested during updating.

Proof. If link l is congested during updating, but $l \notin CL$, there must exists a subset of newly added flows $F'_n(l) \subseteq F_n(l)$ and old flows $F'_o(l) \subseteq F_o(l)$ on link l when the congestion occurs.

$$\sum_{f_i \in F_o(l)} b(f_i) + \sum_{f_i \in F_n(l)} b(f_i) + \sum_{f_i \in F_u(l)} b(f_i) \geq \sum_{f_i \in F'_o(l)} b(f_i) + \sum_{f_i \in F'_n(l)} b(f_i) + \sum_{f_i \in F_u(l)} b(f_i) > c(l)$$

Therefore, l must be a potential congested link. □

B. Dependency Graph For Congestion-free Updating

Intuitively, if flows in $F_o(l)$ are moved away from the potential congested link l before flows in $F_n(l)$ shifting to l, congestions will be avoided. As Figure 1b shows, the update of f_1 depends on the removal of f_2 on potential congested link $C \to D$, while f_2 depends on the removal of f_1 on potential congested link $A \to B$. The dependencies seem make a deadlock. With segments partition, the updating of f_1 to $C \to D$ falls into segment $(C \to D)_{f_1}$ while the removing of f_2 from $C \to D$ is in segment $(C \to G \to D)_{f_2}$. Similarly, the updating of f_2 to $A \to B$ is in segment $(A \to B)_{f_2}$, while the removing of f_1 falls into $(A \to F \to B)_{f_1}$. Thus, dependencies among flows could be divided into local dependencies among segments, e.g., $(A \to F \to B)_{f_1}$ depends on $(A \to B)_{f_2}$ and $(C \to G \to D)_{f_2}$ depends on $(C \to D)_{f_1}$.

To resolve local dependencies, we would like to find the exact local critical nodes controlling traffic on potential congested links. Thus, the updates of these nodes are critical to avoid congestions locally.

Lemma 2. *For flows $\forall f \in F_n(l) \cup F_o(l)$ on a potential congested link $l = u \to v \in P_n(f) - P_o(f)$ or $P_o(f) - P_n(f)$, there must exists at least a critical node $\exists n_f \in C(f)$ preceding u on $P_n(f)$ or $P_o(f)$ respectively.*

Proof. If $u \in C(f)$, $n_f = u$. Otherwise, $u \notin C(f)$,

for $\forall f \in F_o(l)$, $l \notin P_n(f)$. If we can not find a critical node from $P_o(f)[0]$ to u along $P_o(f)$, as the source node $P_o(f)[0] \in P_n(f) \cap P_o(f)$ and $P_o(f)[0] \notin C(f)$, link $P_o(f)[0] \to P_o(f)[1] \in P_n(f)$. Iteratively, $P_o(f)[1] \to P_o(f)[2], ..., u \to v \in P_n(f)$, which conflicts with $l \notin P_n(f)$. Therefore, there must $\exists n_f \in C(f)$ between $P_o(f)[0]$ and u.

for $\forall f \in F_n(l)$, $l \notin P_o(f)$. $u \in$ segment s, and the beginning node $s.start \in C(f) \cup \{P_n(f)[0]\}$. If $s.start \in C(f)$, $n_f = s.start$. Otherwise, similar with the proof of flows in $F_o(l)$, $\exists n_f \in C(f)$ between $P_n(f)[0]$ and u. □

With Lemma 2, we always could find a critical node for each flow to control the amount of traffic on a potential congested link. Moreover, during the multipath transition, the critical node splits traffic of a flow between the old subpath and new subpath, and shifts traffic gradually to the new path with different weights on the new path and old path. Therefore, the dependency among segments could be transformed into dependency among critical nodes controlling traffic on each potential congested link.

Definition 2 (Critical nodes dependency for potential congested link $l = u \to v$)**.** *The last critical node $n_f(l)$ of each flow $f \in F_n(l) \cup F_o(l)$ preceding u controls traffic on l. Thus, the critical node set of $F_n(l)$ depends on the critical node set of $F_o(l)$: $CN(F_n(l)) = \{n_{f_n}(l) | \forall f_n \in F_n(l)\} \rightharpoonup CN(F_o(l)) = \{n_{f_o}(l) | \forall f_o \in F_o(l)\}$.*

With local critical nodes dependencies for potential congested links, we get a dependency graph in which each potential congested link l matches to a directed edge from the critical node set $CN(F_n(l))$ to $CN(F_o(l))$. The critical nodes dependency of link $A \to B$ in Figure 1b is $A_{f_2} \rightharpoonup A_{f_1}$ in which A_{f_2} and A_{f_1} are critical nodes of f_2 and f_1 respectively for link $A \to B$. Likewise, the dependency graph for the three updating scenarios in Figure 1 are showed in Table II. For the dependency graph of Figure 1b, A_{f_1} and C_{f_2} should update before A_{f_2} and C_{f_1} to resolve the two dependencies independently. With independent segments in Table I, the updates in Figure 1b could be scheduled by updating segments $(A \to F \to B)_{f_1}$, $(C \to G \to D)_{f_2}$ before $(A \to B)_{f_2}$ and $(C \to D)_{f_1}$, so that A_{f_1} and C_{f_2} update before A_{f_2} and C_{f_1}.

TABLE II: Dependency Graph

Figure	Dependency Graph
1b	$(A_{f_2}) \rightharpoonup (A_{f_1})$ $(C_{f_1}) \rightharpoonup (C_{f_2})$
1c	$(A_{f_1}) \rightleftharpoons (A_{f_2}) \leftharpoonup (E_{f_1})$ $(C_{f_1}) \rightharpoonup (C_{f_2})$
1d	$(A_{f_1}) \rightleftharpoons (A_{f_2})$ $(C_{f_1}) \rightleftharpoons (C_{f_2})$ $(E_{f_1}) \rightleftharpoons (E_{f_2})$

V. Dependency Resolution

While the global dependency among updates is divided into local dependencies to reduce resolution complexity, switches should also follow reverse order updating in segments to preserve black-hole free and loop free consistency. In this section, we design a heuristic algorithm to resolve the dependency graph while considering update order in each segment.

A. Direct Dependency Resolution

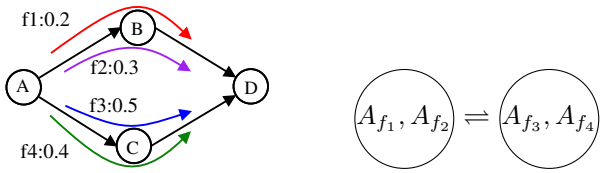
After constructing the dependency graph (lines 4-7) in Algorithm 2, the nodes which do not belong to the dependency graph could be updated with $UpdateSegment(s)$ (line 8) and added to the update sequence US, as long as the downstream nodes in the same segment have already been updated.

For the nodes involved in the dependency graph D, if the update of a critical node $n_f \in CN(F_o(l))$ does not depend on others, which means no other $CN(F_n(l'))$ contains n_f and the downstream nodes in the same segment have been updated ($CanUpdateInSegment(n_f)$), n_f could be updated immediately (lines 10-14). After the update of n_f, nodes in the same segment s previously blocked by n_f due to the reverse order updating are able to update with $UpdateSegment(s)$, and then n_f is removed from the dependency graph (line 13). Furthermore, if the updates of some nodes in $CN(F_o(l))$ relieve potential congestions on l (line 15), lines 16-21 update free critical nodes in $CN(F_n(l))$, and line 22 removes the dependency $d(l)$ from dependency graph D and deletes l from potential congested link set CL.

Although the dependency $C_{f_1} \rightharpoonup C_{f_2}$ in Figure 1c could be scheduled sequentially with lines 10-24 in Algorithm 2, A_{f_1} and A_{f_2} depend on each other which makes a deadlock. The deadlocks in Table II are clear, as each critical node set has only one node. However, the critical node sets of each link may contain several critical nodes. We define the deadlock as a circle in which $CN(F_o)$ and $CN(F_n)$ of adjacent critical node dependencies share at least one critical node.

Algorithm 2 Direct Dependency Resolution

1: $US = \varnothing$ #update sequence
2: $D = \varnothing$ #dependency of critical nodes
3: CL = potential congested links
4: **for** *each link* $l : CL$ **do**
5: $d(l) = CN(F_n(l)) \rightharpoonup CN(F_o(l))$
6: $D = D \cup d(l)$
7: **end for**
8: $US = US + UpdateSegment(s)$ $(\forall f, s \in S(f))$
9: **# Critical node update without deadlocks:**
10: **while** $(\exists d(l) \in D) \wedge (d(l).CN(F_o) = \varnothing$ or $\exists n_f \in d(l).CN(F_o) \wedge n_f \notin \forall d(l').CN(F_n))$ **do**
11: **if** $CanUpdateInSegment(n_f)$ **then**
12: $US = US + n_f + UpdateSegment(s)$ $(n_f \in s)$
13: remove n_f from all D
14: **end if**
15: **if** $IsPotentialCongested(l) == false$ **then**
16: **for** $n_f : d(l).CN(F_n)$ **do**
17: **if** $CanUpdateInSegment(n_f) \wedge n_f \notin \forall d(l').CN(F_n)$ **then**
18: $US = US + n_f + UpdateSegment(s)$ $(n_f \in s)$
19: remove n_f from D
20: **end if**
21: **end for**
22: $D = D - d(l),\ CL = CL - l$
23: **end if**
24: **end while**
25: **# Schedulable critical node update in deadlock:**
26: **while** $(\exists d(l) \in D) \wedge (\exists n_f \in d(l).C(F_0) \wedge InDeadlock(n_f) \wedge CanSchedule(n_f))$ **do**
27: **if** $CanUpdateInSegment(n_f)$ **then**
28: $US = US + n_f + UpdateSegment(s)$ $(n_f \in s)$
29: remove n_f from D
30: **end if**
31: **end while**



(a) Current State (b) Dependency Graph

Fig. 2: Schedulable flows in deadlock: the link bandwidth capacity is 1 unit, and throughputs of flows are marked.

Definition 3 (Deadlock L)**.** *For a set of potential congested links* $\{l_0, l_1, l_2, ..., l_k\}$, $CN(F_o(l_0)) \cap CN(F_n(l_1)) = cn_0 \neq \varnothing$, $CN(F_o(l_1)) \cap CN(F_n(l_2)) = cn_1 \neq \varnothing$, ..., $CN(F_o(l_k)) \cap CN(F_n(l_0)) = cn_k \neq \varnothing$, *the intersection critical node sets* $cn_0, cn_1, ..., cn_k$ *form a deadlock* L.

Actually, not all the nodes involved in a deadlock are blocked. As Figure 2 shows, there are 4 flows from A to D using $A \rightarrow B \rightarrow D$ and $A \rightarrow C \rightarrow D$ respectively. If we want to exchange the routing paths of f_1, f_2 with f_3, f_4 to release more available bandwidth for other flows on link $A \rightarrow C$, the exchanging is recognized as a deadlock in Figure 2b. Indeed, the exchanging could be serialized with the update sequence f_3, f_1, f_2, f_4, as the available bandwidth on new paths is large enough to reroute flows directly. Therefore, schedulable nodes in a deadlock may be updated directly with an appropriate update order. In Algorithm 2, line 26 checks whether nodes in deadlocks ($InDeadlock(n_f)$) could be scheduled with enough bandwidth ($CanSchedule(n_f)$), and then add schedulable nodes to the update sequence in lines 27-30.

Algorithm 3 Multipath Transition

1: **while** $L \neq \varnothing$ **do**
2: $\{n_f, ab\} = CanUpdateInSegment(n_f) \wedge \max_{n_f \in L}\{\min\{n_f.old, \min_{n_f \in \forall d(l').CN(F_n)} AvailBw(l')\}\}$
3: **if** $ab \leq 0$ **then**
4: $RateLimit(f)$ $(f = \max_{f' \in P_n(f)}(b(f')))$
5: **end if**
6: $n_f.old : n_f.new = (n_f.old - ab) : (n_f.new + ab)$
7: $US = US + n_f$
8: **if** $n_f.old == 0$ **then**
9: $US = US + UpdateSegment(s)$ $(n_f \in s)$
10: $L = L - n_f$, remove n_f from D
11: **end if**
12: **end while**

B. Updating With Multipath Transition

Although schedulable critical nodes could update directly to resolve the deadlock in Figure 2, there may be situations in which no node in a deadlock could update completely in a step due to the limitation of bandwidth capacity. For example, if throughputs of f_2 and f_3 are both 0.4 unit, none of these 4 flows could be updated directly. Therefore, we use multipath transition by spitting traffic of a flow between the old path and new path with critical nodes. For the deadlock $A_{f_1} \rightleftharpoons A_{f_2}$ in Figure 1c, we split both flows f_1 and f_2 with link $A \rightarrow B$ and $A \rightarrow E$ to shift traffic to their new paths gradually. To reduce the multipath transition overhead, we design a greedy algorithm to shift the largest amount of traffic to minimize multipath transition steps. For example, A_{f_1} in Figure 1c shifts 0.5 unit of f_1 to $A \rightarrow E$ firstly, as 0.5 unit on $A \rightarrow E$ available for f_1 is larger than 0.2 unit for f_2 on $A \rightarrow B$. In Algorithm 3, line 2 searches the non-blocked critical node with the largest available bandwidth ab in deadlock L. The available shifting bandwidth ab for n_f is the minimum value of traffic amount on the old path $n_f.old$ and minimum available bandwidth along the new subpath. If $ab > 0$, lines 6-7 reassign weights $n_f.old : n_f.new$ for the old path and new path, and add the node to update sequence. With the multipath migration, all the traffic on the old path is finally shifted to the new path (line 8), so that line 10 removes n_f from the dependency graph. After the completion of multipath transition in Figure 1c, E_{f_1} is able to update as the dependency on A_{f_2} has been resolved.

Unfortunately, multipath transition may be blocked by fully utilized links. For example, in Figure 2, if throughputs of the 4 flows are 0.5 unit, no flow could migrate as there is no available bandwidth at all. In this case, the maximum available bandwidth $ab \leq 0$ (line 3), so that we need to limit throughputs of some flows to release a small amount of available bandwidth for multipath transition. Thus, if we would like to shift f_1 with multipath in Figure 2, we should limit rate of flows f_3 or f_4 on the new path of f_1, e.g., reducing 0.2 unit of f_3, so that f_1 could split traffic with weights 0.2 : 0.3 on the new path and old path, and then multipath transition is able to carry out as normal. After the multipath transition, the throughputs of rate-limited flows are restored to reduce throughput losses.

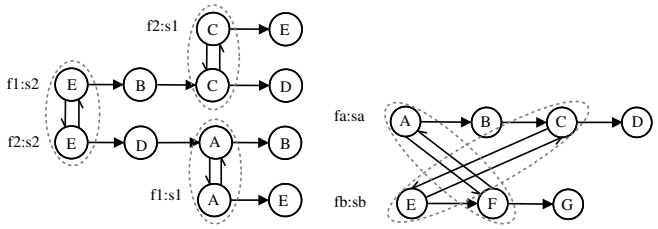
C. Dependency Resolution With Segments

For Figure 1d, the dependency graph requires multipath transitions for (A_{f_1}, A_{f_2}), (C_{f_1}, C_{f_2}) and (E_{f_1}, E_{f_2}). While a multipath transition updates multiple nodes in a deadlock simultaneously, nodes should follow reserve order updating in each segment (checked by $CanUpdateInSegment(n_f)$ in Algorithm 3). Combining the dependency graph and segments of Figure 1d together, Figure 3a shows that multipath transitions of (A_{f_1}, A_{f_2}) and (C_{f_1}, C_{f_2}) are independent from each other, so that these two deadlocks could be scheduled concurrently as long as their downstream nodes have been updated. After the resolution of $A_{f_1} \rightleftharpoons A_{f_2}$ and $C_{f_1} \rightleftharpoons C_{f_2}$, switch D in segment s_2 of f_2 and switch B in segment s_2 of f_1 are able to update, and then the multipath transition of (E_{f_1}, E_{f_2}) resolves the dependency graph finally.

Although dependency graph could be resolved following reverse order updating of each segment in Figure 3a, there are still situations in which dependency can not be solved due to conflicts between the dependency graph and segments. Figure 3b requires multipath transitions for (A_{f_a}, F_{f_b}) and (C_{f_a}, E_{f_b}). Meanwhile, the reserve order updating of segments s_a and s_b instructs that C_{f_a} updates before A_{f_a} and F_{f_b} updates before E_{f_b} respectively. Consequently, the multipath transition of (A_{f_a}, F_{f_b}) has to wait for the update completion of C_{f_a}, while the update of C_{f_a} in multipath transition of (C_{f_a}, E_{f_b}) requests F_{f_b} in (A_{f_a}, F_{f_b}) to update firstly. Thus, there is a conflict between dependency graph and segment. We detect this kind of conflicts by checking circles formed by the dependency graph and reserve order of segments. The circle $A \rightarrow B \rightarrow C \rightleftharpoons E \rightarrow F \rightleftharpoons A$ in Figure 3b is composed of critical nodes dependencies and sub-segments, while there is no circle formed in Figure 3a.

Theorem 3. *If no circle forms with the dependency graph and reverse order of segments, we could always find an update order with Algorithms 2 and 3.*

Algorithm 2 firstly updates free nodes and schedulable nodes in deadlocks, and then Algorithm 3 resolves deadlocks with multipath transition and rate-limit. After resolving deadlocks, nodes previously blocked by deadlocks are now relieved and could be scheduled by running Algorithm 2 again.



(a) Dependencies and segments in Figure 1d (b) Conflicts between dependency graph and segments

Fig. 3: Combining dependency graph with segments

VI. Evaluation

A. Evaluation Setup

We implement Cupid with 2000+ lines of Java code. Cupid sits between the routing modules (e.g., failure recovery, load balancing) and the control message communication module in the controller. In the architecture showed in Figure 4, all control messages to manipulate forwarding rules in flow tables are captured to schedule an appropriate updating order before applied in switches. The new routing path of each flow is partitioned into several independent segments with Algorithm 1. We identify potential congested links with Definition 1, and then generate a dependency graph among critical nodes for these links with Definition 2. Loops in routing paths and deadlocks in the dependency graph are recognized with strong connected components. Finally, with Algorithm 2 and 3, a feasible updating order is scheduled to update flow tables consistently.

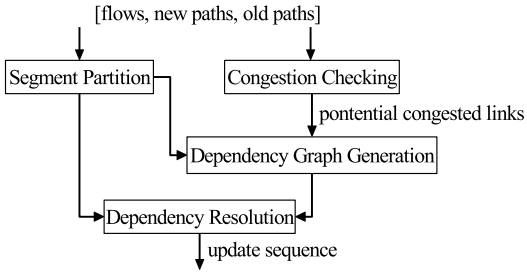
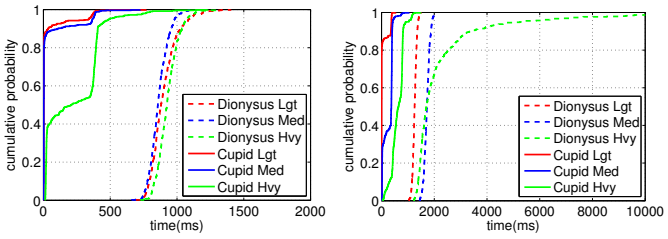


Fig. 4: Cupid Architecture

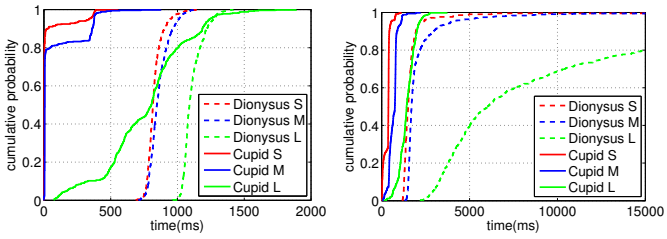
We evaluate Cupid with fat-tree and mesh networks. These two topologies have different path deployments and traffic distributions, so that the dependencies imposed by congestion-free consistency are quite different. Each network consists of 100 switches connected by 1Gbps links.

- Fat-tree: We use a three-layer fat-tree topology [19] with 12 edge switches, 22 aggregate switches and 66 core switches, so that link bandwidth capacities of aggregated layer and core layer are balanced. Traffic distribution in the fat-tree network follows the data center traffic characteristics in [20].
- Mesh: The diameter of the mesh topology is 18, and the degree of each node in mesh network is 4. The traffic in the mesh network is randomly generated and uniformly distributed among all node pairs.



(a) Fattree (b) Mesh

Fig. 5: Ordering latency at different traffic load



(a) Fattree (b) Mesh

Fig. 6: Ordering latency with different flow size

More than 10,000 flows are running simultaneously in each network. During simulation, we assign link failures in the network and reroute affected flows to other available paths, and also schedule flows to other less loaded paths for load balancing. We compare the update ordering efficiency of Cupid with Dionysus [8] and random update ordering.

B. Evaluation Results

1) Update Ordering Latency: Figure 5 shows the ordering latency of updating 1000 flows simultaneously at light (<40% network utilization), medium (40∼80% network utilization) and heavy (>80% network utilization) traffic load. The network utilization is measured by weighted link utilization. In both fat-tree and mesh networks, Cupid takes shorter time to schedule a feasible updating order than Dionysus. With divided independent segments and local dependencies among critical nodes, Cupid reduces dependency resolution complexity and ensures a faster update ordering. For the three layer fat-tree network, the longest routing path is only 4 hops and each switch has its own dedicated links to the higher and lower layer switches, so that flows have fewer chances to collide to congest links during updating. Thus, the dependency graph is relatively simple due to the short and dedicated routing paths in the fat-tree. Therefore, Cupid could finish ordering within 500ms in most cases while Dionysus solves the global dependency graph in 1000ms. On the other hand, the mesh network tends to encounter more congestions during updating compared with fat-tree, especially on links in the middle of network shared by a lot of flows. Meanwhile, the routing paths in the mesh topology are usually longer than 4 hops in the fat-tree, and there may be loops formed with new and old paths during updating, which adds to the complexity of dependency graph. For the light and medium traffic load, although Dionysus could schedule the ordering within 2000ms, the ordering time of Cupid is 500ms which is 4 times faster than Dionysus. The situation is much worse for Dionysus at heavy traffic load. As the highly complex dependency graph is more difficult to resolve due to the scarcity of available bandwidth, Dionysus takes even tens of seconds to find a feasible ordering, while Cupid is still able to schedule the ordering within 1000ms.

2) Dependency Resolution Analysis: Although a lot of researches [4], [5], [6] discover and reroute large flows to less congested path for load balancing, large flows are more likely to be stuck by limited bandwidth resource during updating. Thus, large flows tend to migrate to new paths with multipath transition, while small flows probably could be scheduled freely with a small amount of available bandwidth. To show this difference in updating, we classify flows into three classes according to flow size: small flow (<1M), medium flow (1∼10M) and large flow (>10M). Figure 6 shows the order scheduling time for 1000 flows with the three classes at heavy traffic load. In the fat-tree topology, the update ordering of small and medium flows in Cupid takes much shorter than Dionysus. However, the ordering of large flows updates in Cupid may be worse than Dionysus at times. Figure 7 shows most of flows in the fat-tree topology have only 1 segment, as the ingress and egress switches are the only common nodes in the new and old paths, so that the segment partition brings few benefits for fat-tree. Moreover, with large flows at heavy traffic load, Cupid identifies almost all the links as potential congested links and constructs dependencies for these links, so that the resolution of the large dependency graph takes longer. Nevertheless, the overall update ordering time of Cupid is much shorter than Dionysus in most cases (over 90%). Compared with the fat-tree topology, flows usually have longer routing paths in mesh network and also larger number of segments as Figure 7 shows. With the benefits of independent segments and local dependencies, Cupid always outperforms Dionysus in mesh network. Especially for large flows, Dionysus takes even tens of seconds to resolve the complicated global dependency graph, while Cupid is able to finish ordering within 2000ms.

To understand how dependencies are resolved, we divide the resolution process into 4 phases: non-deadlock, sched-deadlock, multipath and rate-limit, which correspond to the updates without deadlock, schedulable updates in deadlocks, multipath transition and rate-limit respectively in Algorithm 2 and 3. Figure 8 shows phases at which dependencies are resolved for different size of flows at heavy traffic load. As small flows almost could be freely scheduled in the fat-tree network, few flows are involved in dependency resolution, and the dependencies could be solved in the non-deadlock phase. Compared with the mesh topology, the update ordering in fat-tree could be scheduled in the first three phases, while updates in mesh network tend to be scheduled in the last three phases due to the higher dependency complexity. Especially for the large flows updating in the mesh network, over 2% orderings fall into the rate-limit phase which results in throughput losses.

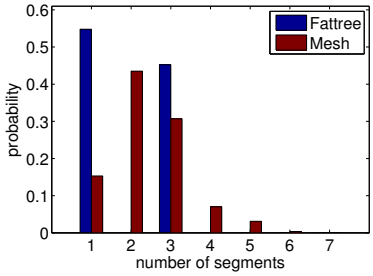


Fig. 7: Segments Count

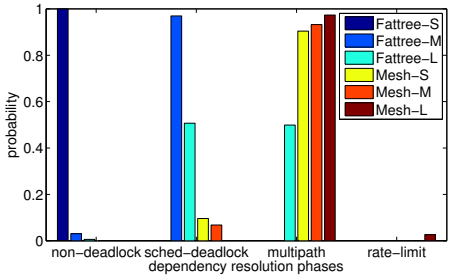


Fig. 8: Dependency Resolution Phases

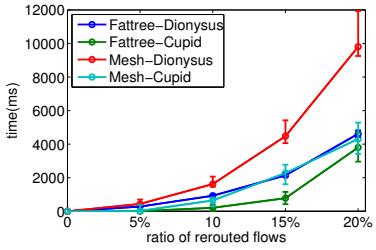


Fig. 9: Ordering latency with different flow counts (50th percentile with [10th, 90th])

TABLE III: Network utilization losses (90th percentile)

Topology	Approach	Small	Medium	Large
Fattree	Cupid	0	0	0
	Dionysus	0	1.59×10^{-16}	2.46×10^{-9}
	Random	0	0.06%	0.48%
Mesh	Cupid	0	0	0.16%
	Dionysus	0.12%	0.79%	4.71%
	Random	0.54%	1.41%	7.39%

3) Throughput Losses: Due to the limited link bandwidth, flows have to reduce their throughputs once fall into the rate-limit phase during updating. Cupid could schedule updates without any throughput loss in non-deadlock, sched-deadlock and multipath phases. Only a small percentage of orderings fall in the rate-limit phase as Figure 8 shows. We compare the network utilization losses of Cupid with Dionysus and random update ordering. In Table III, there are always less network utilization losses with Cupid than Dionysus and random update ordering in both fat-tree and mesh networks. Even though the losses in the fat-tree network are quite low using Dionysus, Cupid does not experience any loss. Moreover, for the large flows migration in the mesh network, the network utilization loss in Cupid is 0.16% while the losses of Dionysus and random update order are 10 times larger than Cupid.

4) Scalability: We further study how the update ordering latency scales with the number of involved flows at heavy traffic load, and find that Cupid always schedules faster than Dionysus when migrating 0∼20% of flows in the network to new paths in Figure 9. Especially for mesh topology, the ordering time in Dionysus rises steeply with the increasing number of involved flows, as more flows adds to the global dependency graph size and complexity. Cupid is able to schedule the ordering within 4000ms, while Dionysus takes more than 10 seconds for updating 20% flows in mesh network.

VII. Conclusion

With increasing SDN applications scheduling flows for load balancing and failure recovery, in this paper, we focus on updating flow tables in data plane consistently and efficiently while preserving throughputs of flows. To reduce the overhead of finding a feasible updating order, we firstly partition the rerouted path into independent segments, and then divide the global dependency among updates into local dependencies among critical nodes. We then design and implement a heuristic dependency resolution algorithm with the dependency graph and reverse order updating within each segment. To reduce the flow table space overhead, we use multiple ports with weights in each flow entry during updating, so that there is only one rule kept for each flow in a switch. The results of simulation show that Cupid is able to schedule update ordering at least 2 times faster than Dionysus and has less throughput losses in both fat-tree and mesh topologies.

References

[1] S. Jain, A. Kumar, S. Mandal, J. Ong *et al.*, "B4: Experience with a globally-deployed software defined wan," in *SIGCOMM*, 2013.

[2] S. Agarwal, M. Kodialam, and T. Lakshman, "Traffic engineering in software defined networks," in *INFOCOM*, 2013.

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

[4] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang *et al.*, "Hedera: Dynamic flow scheduling for data center networks." in *NSDI*, 2010.

[5] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula *et al.*, "Devoflow: Scaling flow management for high-performance networks," in *SIGCOMM*, 2011.

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

[7] M. Suchara, D. Xu, R. Doverspike, D. Johnson *et al.*, "Network architecture for joint failure recovery and traffic engineering," in *SIGMETRICS*, 2011.

[8] X. Jin, H. H. Liu, R. Gandhi, S. Kandula *et al.*, "Dynamic scheduling of network updates," in *SIGCOMM*, 2014.

[9] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *HotNets*, 2013.

[10] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang *et al.*, "Achieving high utilization with software-driven wan," in *SIGCOMM*, 2013.

[11] H. H. Liu, X. Wu, M. Zhang, L. Yuan *et al.*, "zupdate: Updating data center networks with zero loss," in *SIGCOMM*, 2013.

[12] S. Ghorbani and M. Caesar, "Walk the line: consistent network updates with bandwidth guarantees," in *HotSDN*, 2012.

[13] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger *et al.*, "Abstractions for network update," in *SIGCOMM*, 2012.

[14] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *HotSDN*, 2013.

[15] T. Mizrahi, O. Rottenstreich, and Y. Moses, "Timeflip: Scheduling network updates with timestamp-based tcam ranges," in *INFOCOM*, 2015.

[16] W. Zhou, D. Jin, J. Croft, M. Caesar *et al.*, "Enforcing customizable consistency properties in software-defined networks," in *NSDI*, 2015.

[17] L. Shi, J. Fu, and X. Fu, "Loop-free forwarding table updates with minimal link overflow," in *ICC*, 2009.

[18] J. McClurg, H. Hojjat, and N. Foster, "Efficient synthesis of network updates," in *PLDI*, 2015.

[19] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM*, 2008.

[20] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," in *SIGCOMM*, 2010.