Dynamic Scheduling of Network Updates (Extended version)

Xin Jin[†] Hongqiang Harry Liu[⋆] Rohan Gandhi[∧] Srikanth Kandula[◦]
Ratul Mahajan[◦] Ming Zhang[◦] Jennifer Rexford[†] Roger Wattenhofer[×]

Microsoft Research[◦] Princeton University[†] Yale University[⋆] Purdue University[∧] ETH Zurich[×]

Abstract— We present Dionysus, a system for fast, consistent network updates in software-defined networks. Dionysus encodes as a graph the consistency-related dependencies among updates at individual switches, and it then dynamically schedules these updates based on runtime differences in the update speeds of different switches. This dynamic scheduling is the key to its speed; prior update methods are slow because they pre-determine a schedule, which does not adapt to runtime conditions. Testbed experiments and data-driven simulations show that Dionysus improves the median update speed by 53–88% in both wide area and data center networks compared to prior methods.

1. INTRODUCTION

Many researchers have shown the value of centrally controlling networks. This approach can prevent oscillations due to distributed route computation [1]; ensure that network paths are policy compliant [2, 3]; reduce energy consumption [4]; and increase throughput [5, 6, 7, 8, 9]. Independent of their goal, such systems operate by frequently updating the data plane state of the network, either periodically or based on triggers such as failures. This state consists of a set of rules that determine how switches forward packets.

A common challenge faced in all centrally-controlled networks is consistently and quickly updating the data plane. Consistency implies that certain properties should not be violated during network updates, for instance, packets should not loop (*loop freedom*) and traffic arriving at a link should not exceed its capacity (*congestion freedom*). Consistency requirements impose dependencies on the order in which rules can be updated at switches. For instance, for congestion freedom, a rule update that brings a new flow to a link must occur after an update that removes an existing flow if the link cannot support both flows simultaneously. Not obeying update ordering requirements can lead to inconsistencies such as loops, blackholes, and congestion.

Current methods for consistent network updates are slow because they are based on *static* ordering of rule updates [9, 10, 11, 12]. They pre-compute an order in which rules must be updated, and this order does not adapt to runtime differences in the time it takes for individual switches to apply updates. These differences inevitably arise because of disparities in switches' hardware and CPU load and the variabilities in the time it takes the centralized controller to make remote procedure calls (RPC) to switches. In B4, a centrally-controlled wide area network, the ratio of the 99th percentile to the median delay to change a rule at a switch was found to be over five (5 versus 1 second) [8]. Further, some switches can "straggle," taking substantially more time than average (e.g., 10-100x) to apply an update. Current methods can stall in the face of straggling switches.

The speed of network updates is important because it determines the agility of the control loop. If the network is being updated in response to a failure, slower updates imply a longer period during which congestion or packet loss occurs. Further, many systems update the network based on current workload, both in the wide area [8, 9] and the data center [5, 6, 7], and their effectiveness is tied to how quickly they adapt to changing workloads. For example, recent works [8, 9] argue for frequent traffic engineering (e.g., every 5 minutes) to achieve high network utilization; slower network updates would lower network utilization.

We develop a new approach for consistent network updates. It is based on the observations that $i)$ there exist multiple valid rule orderings that lead to consistent updates; and $ii)$ dynamically selecting an ordering based on update speeds of switches can lead to fast network updates. Our approach is general and can be applied to many consistency properties, including all the ones that have been explored by prior work [9, 10, 11, 12, 13].

We face two main challenges in practically realizing our approach. The first is devising a compact way to represent multiple valid orderings of rule updates; there can be exponentially many such orderings. We address this challenge using a dependency graph in which nodes correspond to rule updates and network resources, such as link bandwidth and switch rule memory capacity, and (directed) edges denote dependencies among rule updates and network resources. Scheduling updates in any order, while respecting dependencies, guarantees consistent updates.

The second challenge is scheduling updates based on dynamic behavior of switches. This problem is NP-complete in the general case, and making matters worse, the dependency graph can also have cycles. To schedule efficiently, we develop greedy heuristics based on preferring critical paths and strongly connected components in the dependency graph [14].

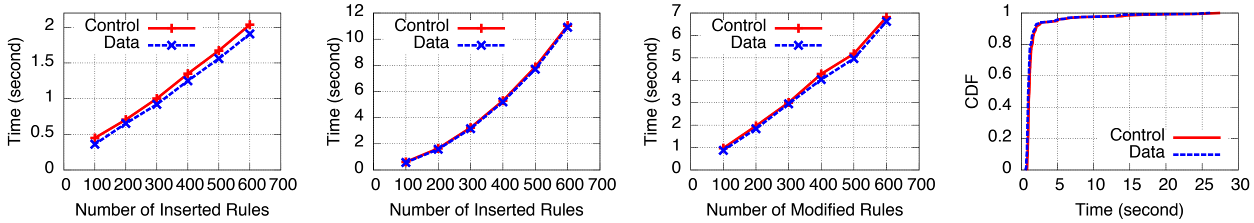
We instantiate our approach in a system called Dionysus and evaluate it using experiments on a modest-sized testbed and large-scale simulations. Our simulations are based on topology and traffic data from two real networks, one wide-area network and one data center network. We show that Dionysus improves the median network update speed by 53–88%. We also show that its faster updates lower congestion and packet loss by over 40%.

2. MOTIVATION

Our work is motivated by the observations that the time to update switch rules varies widely and that not accounting for this variation leads to slow network updates. We illustrate these observations using measurements from commodity switches and simple examples.

2.1 Variability in update time

Several factors lead to variable end-to-end rule update times, including switch hardware capabilities, control load on the switch, the nature of the updates, RPC delays (which include network path delays), etc. [7, 8, 15, 16]. To illustrate this variability, we perform



(a) (b) (c) (d)

Figure 1: Rule update times on a commodity switch. (a) Inserting single-priority rules. (b) Inserting random-priority rules. (c) Modifying rules in a switch with 600 single-priority rules. (d) Modifying 100 rules in a switch with concurrent control plane load.

controlled experiments on commodity switches. In these experiments, RPC delays are negligible and identical switch hardware and software are used, yet significant variability is evident.

The experiments explore the impact of four factors: i) the number of rules to be updated; ii) the priorities of the rules; iii) the types of rule updates (e.g., insertion vs. modification); and iv) control load on the switch. We measure switches from two different vendors and observe similar results. Figure 1 shows results for one switch vendor. We build a customized switch agent on the switch and obtain confirmation of rule updates in both the control and data planes. The control plane confirmation is based on the switch agent verifying that the update is installed in the switch's TCAM (ternary content addressable memory), and the data plane confirmation is based on observing the impact of the update in the switch's forwarding behavior (e.g., changes in which interface a packet is sent out on).

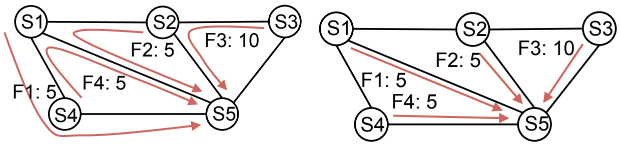
Figure 1(a) shows the impact of the number of rules by plotting the time to add different numbers of rules. Here, the switch has no control load besides rule updates, the switch starts with an empty TCAM, and all rule updates correspond to adding new rules with the same priority. We see that, as one might expect, that the update time grows linearly with the number of rules being updated, with the per-rule update time being 3.3 ms.

Figure 1(b) shows the impact of priorities. As above, the switch has no load and starts with an empty TCAM. The difference is that the inserted rules are assigned random priorities. We see that the per-rule update time is significantly higher than before. The slope of the line increases as the number of rules increase, and the per-rule update time reaches 18 ms when inserting 600 rules.

This variability stems from the fact that TCAM packing algorithms do different amounts of work, depending on the TCAM's current content and the type of operation performed. For instance, the TCAM itself does not encode any rule priority information. The rules are stored from top to bottom in decreasing priority and when multiple rules match a packet, the one with the highest place is chosen. Thus, when a new rule is inserted, it may cause existing rules to move in the table. Although the specific packing algorithms are proprietary and vary across vendors, the intrinsic design of a TCAM makes the update time variable.

Figure 1(c) shows the impact of the type of rule update. Rather than inserting rules into an empty TCAM, we start with 600 rules of the same priority and measure the time for rule modifications. We modify only match fields or actions, not rule priorities. The graph is nearly linear, with a per-rule modification latency of 11 ms. This latency is larger than the per-rule insertion latency because a rule modification requires two operations in the measured switch: inserting the new rule and deleting the old rule.

Finally, Figure 1(d) shows the impact of control load, by engaging the switch in different control activities while updates are performed. Here, the switch starts with the 600 same-priority rules and we modify 100 of them. Control activities performed include reading packet and byte counters on rules with OpenFlow protocol, querying SNMP counters, reading switch information with CLI commands, and running BGP protocol (which SDN systems use as backup [8]). We see that despite the fact that update operations are identical (100 new rules), the time to update highly varies, with the 99th percentile 10 times larger than the median. Significant rule update time variations are also reported in [8, 16].



(a) Current State (b) Target State

Figure 2: A network update example. Each link has 10 units of capacity; flows are labeled with their sizes.

In summary, we find that even in controlled conditions, switch update time varies significantly. While some sources of this variability can be accounted for statically by update algorithms (e.g., number of rule updates), others are inherently dynamic in nature (e.g., control plane load and RPC delays). Accounting for these dynamic factors ahead of time is difficult. Our work thus focuses on adapting to them at runtime.

2.2 Consistent updates amid variability

We illustrate the downside of static ordering of rule updates with the example of Figure 2. Each link has a capacity of 10 units and each flow's size is marked. The controller wants to update the network configuration from Figure 2(a) to 2(b). Assume for simplicity that the network uses tunnel-based routing and all necessary tunnels have already been established. So, moving a flow requires updating only the ingress switch.

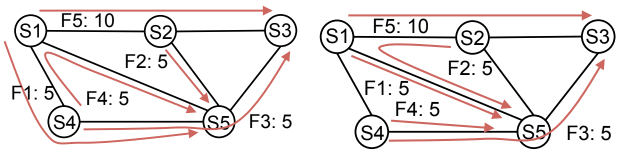
If we want a congestion-free network update, we cannot update all the switches in "one shot" (i.e., send all update commands simultaneously). Since different switches will apply the updates at different times, such a strategy may cause congestion at some links. For instance, if $S1$ applies the update for moving $F1$ before $S2$ moves $F2$ and $S4$ moves $F4$, link $S1$-$S5$ will be congested.

Ensuring that no link is congested requires us to carefully order the updates. Two valid orderings are:

Plan A: $[F3 \rightarrow F2]$ $[F4 \rightarrow F1]$
Plan B: $[F4]$ $[F3 \rightarrow F2 \rightarrow F1]$

Plan A mandates that $F2$ be done after $F3$ and $F1$ be done after $F4$. Plan B mandates that $F1$ be done after $F2$ and that $F2$ be done after $F3$. In both plans, $F3$ and $F4$ have no pre-requisites and can be done anytime and in parallel.[1]

[1] Some consistent update methods [9] use stages, a more rigid version of static ordering. They divide updates into multiple stages,



(a) Current State (b) Target State

Figure 3: An example in which a completely opportunistic approach to scheduling updates leads to a deadlock. Each link has 10 units of capacity; flows are labeled with their sizes. If $F2$ is moved first, $F1$ and $F4$ get stuck.

Which plan is faster? In the absence of update time variability, if all updates take unit time, Plan A will take 2 time units and Plan B will take 3. However, with update time variability, no plan is a clear winner. For instance, if $S4$ takes 3 time units to move $F4$, and other switches take 1, Plan A will take 4 time units and Plan B will take 3. On the other hand, if $S2$ is slow and takes 3 time units to move $F2$, while other switches take 1, Plan A will take 4 time units and Plan B will take 5.

Now consider a dynamic plan that first issues updates for $F3$ and $F4$, issues an update for $F2$ as soon as $F3$ finishes, and issues an update for $F1$ as soon as $F2$ *or* $F4$ finishes. This plan dynamically selects between the two static plans above and will thus equal or beat those two plans regardless of which switches are slow to update. Practically implementing such plans for arbitrary network topologies and updates is the goal of our work.

3. DIONYSUS OVERVIEW

We achieve fast, consistent network updates through dynamic scheduling of rule updates. As in the example above, there can be multiple valid rule orderings that lead to consistent updates. Instead of statically selecting an order, we implement on-the-fly ordering based on the realtime behavior of the network and the switches.

Our focus is on flow-based traffic management applications for the network core (e.g., ElasticTree, MicroTE, B4, SWAN [4, 6, 8, 9]). As is the case for these applications, we assume that any forwarding rule at a switch matches at most one flow, where a flow is (a subset of) traffic between ingress and egress switches that uses either single or multiple paths. This assumption does not hold in networks that use wild-card rules or longest prefix matching. Increasingly, such rules are being moved to the network edge or even hosts [17, 18, 19], keeping the core simple with exact match rules.

The primary challenge is to tractably explore valid orderings. One difficulty is that there are combinatorially many such orderings. Conceivably, one may formulate the problem as an ILP (Integer Linear Program). But this approach would be too slow and does not scale to large networks with a lot of flows. Also it is static and not incrementally computable; one has to rerun the ILP every time the switch behaviors change. Another difficulty is that the extreme approach of being completely opportunistic about rule ordering does not always work. In such an approach, the controller will immediately issue any updates that are not gated (per consistency requirements) on any other update. While this approach works for the simple example in the previous section, in general, it can result in deadlocks (that are otherwise resolvable). Figure 3 shows an example. Since $F2$ can be moved without waiting for any other flow movement, an opportunistic approach might make that move.

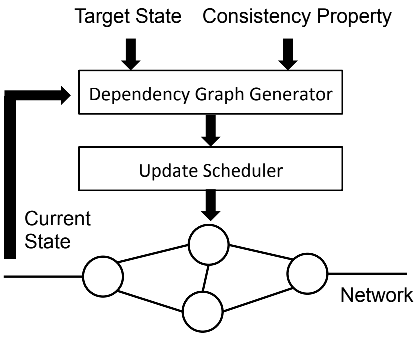
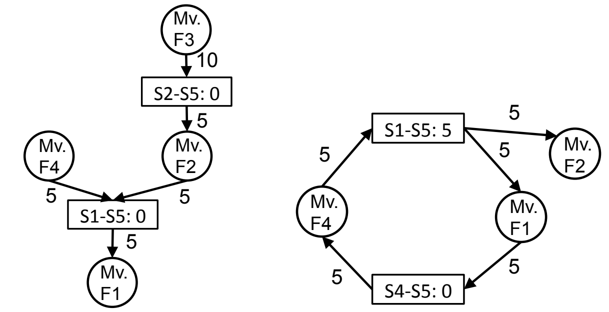


Figure 4: Our approach.



(a) Dependency graph for Figure 2 (b) Dependency graph for Figure 3

Figure 5: Example dependency graphs.

But at this point, we are stuck, because no flow can be moved to its destination without overloading at least some link. This is avoidable if we move other flows first. It is because of such possibilities that current approaches carefully plan transitions, but they err on the side of not allowing any runtime flexibility in rule orderings.

We balance planning and opportunism using a two-stage approach, shown in Figure 4. In the first stage, we generate a *dependency graph* that compactly describes many valid orderings. In the second stage, we schedule updates based on the constraints imposed by the dependency graph. Our approach is general in that it can maintain any consistency property that can be described using a dependency graph, which includes all properties used in prior work [8, 9, 11]. The scheduler is independent of the consistency property.

Figure 5(a) shows a simplified view of the dependency graph for the example of Figure 2. In the graph, circular nodes denote update operations, and rectangular nodes represent link capacity resources. The numbers within rectangles indicate the current free capacity of resources. A label on an edge from an operation to a resource node shows the amount of resource that will be released when the operation completes. For example, link $S2$-$S5$ has 0 free capacity, and moving $F3$ will release a capacity of 10 to it. Labels on edges from resource to operation nodes show the amount of free resource needed to conduct these operations. As moving $F1$ requires 5 free capacity on link $S1$-$S5$, $F1$ cannot move until $F2$ or $F4$ finishes.

Given the dependency graph in Figure 5(a), we can dynamically generate good schedules. First, we observe that $F3$ and $F4$ don't depend on other updates, so they can be scheduled immediately. After $F3$ finishes, we can schedule $F2$. Finally, we schedule $F1$ once *one* of $F2$ or $F4$ finishes. From this example, we see that the dependency graph captures dependencies but still leaves scheduling flexibility, which we leverage at runtime to implement fast updates.

There are two challenges in dynamically scheduling updates. The first is to resolve cycles in the dependency graph. These arise due to complex dependencies between rules. For example, Fig-

and all updates in the previous stage must finish before any update in the next stage can begin. In this terminology, Plan A is a two-stage solution in which the first stage will update $F3$ and $F4$ and the second will update $F2$ and $F1$. Plan B is a three-stage solution. Since SWAN minimizes the number of stages, it will prefer Plan A.

Index	Operation
A	Add p_3 at S1
B	Add p_3 at S4
C	Add p_3 at S5
D	Change weight at S1
E	Delete p_2 at S1
F	Delete p_2 at S2
G	Delete p_2 at S5

Table 1: Operations to update f with tunnel-based rules.

Index	Operation
X	Add weights with new version at S2
Y	Change weights, assign new version at S1
Z	Delete weights with old version at S2

Table 2: Operations to update f in WCMP forwarding.

ure 5(b) shows that there are cycles in the dependency graph for the example of Figure 3. Second, at any given time, multiple subsets of rule updates can be issued, and we need to decide which ones to issue first. As described later, the greedy heuristics we use for these challenges are based on critical-path scheduling and the concept of SCC (strongly connected component) in graph theory.

4. NETWORK STATE MODEL

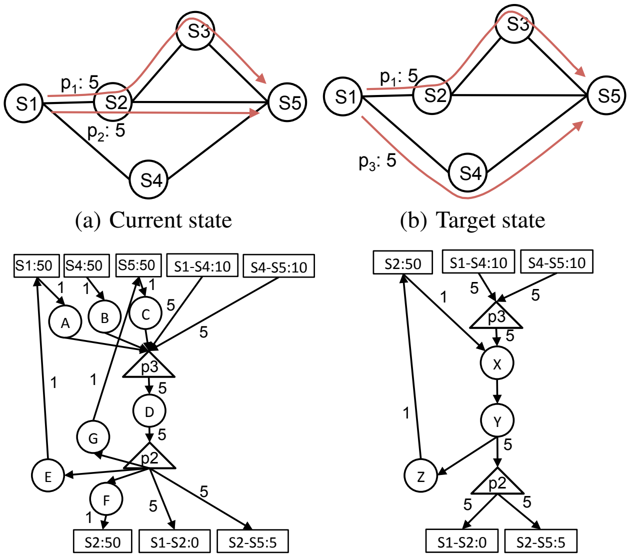
This section describes the model of network forwarding state that we use in Dionysus. The following sections describe dependency graph generation and scheduling in detail.

The network G consists of a set of switches S and a set of directed links L. A flow f is from an ingress switch s_i to an egress switch s_j with traffic volume t_f, and its traffic is carried over a set of paths P_f. The forwarding state of f is defined as $R_f = \{r_{f,p} | p \in P_f\}$ where $r_{f,p}$ is the traffic load of f on path p. The network state NS is then the combined state of all flows, i.e., $NS = \{R_f | f \in F\}$. For example, consider the network in Figure 6(a) that is forwarding a flow across two paths, with 5 units of traffic along each. Here, $t_f = 10$, $P_f = \{p_1 = S_1S_2S_3S_5, p_2 = S_1S_2S_5\}$, and $R_f = \{r_{f,p_1} = 5, r_{f,p_2} = 5\}$.

The state model above captures both tunnel-based forwarding that is prevalent in WANs and also WCMP (weighted cost multi path) forwarding that is prevalent in data center networks. In tunnel-based forwarding, a flow is forwarded along one or more tunnels. The ingress switch matches incoming traffic to the flow, based on packet headers, and splits it across the tunnels based on configured weights. Before forwarding a packet along a tunnel, the ingress switch tags the packet with the tunnel identifier. Subsequent switches only match on tunnel tags and forward packets, and the egress switch removes the tunnel identifier. Representing tunnel-based forwarding in our state model is straightforward. P_f is the set of tunnels and the weight of a tunnel is $r_{f,p}/t_f$.

In WCMP forwarding, switches at *every hop* match on packet headers and split flows over multiple next hops with configured weights. Shortest-path and ECMP (equal cost multipath) forwarding are special cases of WCMP forwarding. To represent WCMP routing in our state model, we first calculate the flow rate on link l as $r_f^l = \sum_{l \in p, p \in P_f} r_{f,p}$. Then at switch s_i, the weight for next-hop s_j is: $w_{i,j} = r_f^{l_{ij}} / \sum_{l \in L_i} r_f^l$ where l_{ij} is the link from s_i to s_j and L_i is the set of links starting at s_i. For instance, in Figure 6(a), $w_{1,2} = 1, w_{1,4} = 0, w_{2,3} = 0.5, w_{2,5} = 0.5$.

5. DEPENDENCY GRAPH GENERATION



(a) Current state

(b) Target state

(c) Dependency graph using tunnel-based rules (Table 1)

(d) Dependency graph using WCMP-based rules (Table 2)

Figure 6: Example of building dependency graph for updating flow f from current state (a) to target state (b).

As shown in Figure 4, the dependency graph generator takes as input the current state NS_c, the target state NS_t, and the consistency property. The network states includes the flow rate, and as in current systems [4, 6, 8, 9], we assume that flows obey this rate as a result of rate limiting or robust estimation. A static input to Dionysus is the rule capacity of each switch, relevant in settings where this resource is limited. Since Dionysus manages all rule additions and removals, it then knows how much rule capacity is available on each switch at any given time. This information is used such that rule capacity is not exceeded at any switch.

Given NS_c and NS_t, it is straightforward to compute the set of operations that would update the network from NS_c to NS_t. The goal of dependency graph generation is to inter-link these operations based on the consistency property. Our dependency graph has three types of nodes: *operation nodes*, *resource nodes*, and *path nodes*. Operation nodes represent addition, deletion, or modification of a forwarding rule at a switch, and resource nodes correspond to resources such as link capacity and switch memory and are labeled with the amount of resource currently *available*. An edge between two operation nodes captures an *operation dependency* and implies that the parent operation must be done before the child. An edge between a resource and an operation node captures a *resource dependency*. An edge from a resource to an operation node is labeled with the amount of resource that must be available before the operation can occur. An edge from an operation to a resource node is labeled with the amount of the resource that will be freed by that operation. There are no edges between resource nodes.

Path nodes help group operations and link capacity resources on a path. Path nodes can connect to operation nodes as well as to resource nodes. An edge between an operation and a path node can be either an operation dependency (un-weighted) or a resource dependency (weighted). The various types of links connecting different types of nodes are detailed in Figure 7.

During scheduling, each path node that frees link resources has a label *committed* that denotes the amount of traffic that is moving away from the path; when the movement finishes, we use *committed* to update the free resource of its child resource nodes. We do not

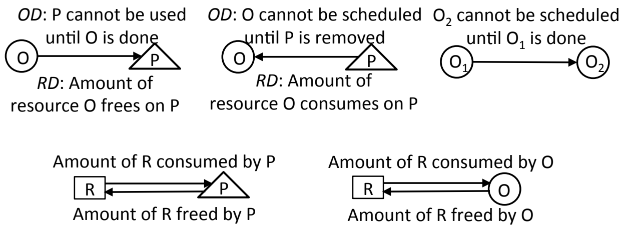

OD: P cannot be used until O is done
O
P
RD: Amount of resource O frees on P
OD: O cannot be scheduled until P is removed
O
P
RD: Amount of resource O consumes on P
O2 cannot be scheduled until O1 is done
O1
O2
Amount of R consumed by P
R
P
Amount of R freed by P
Amount of R consumed by O
R
O
Amount of R freed by O

Figure 7: Links and relationships among path, operation, and resource nodes; *RD* indicates a resource dependency and *OD* indicates an operation dependency.

need to keep *committed* for path nodes that require resource, because we always reduce free capacity on its parent resource nodes first before we move traffic into the path.

In this paper, we focus on four consistency properties from prior work [13] and show how our dependency graphs capture them. The properties are $i)$ *blackhole-freedom*: no packet should be dropped at a switch (e.g., due to a missing rule); $ii)$ *loop-freedom*: no packet should loop in the network; $iii)$ *packet coherence*: no packet should see a mix of old and new rules in the network; and $iv)$ *congestion-freedom*: traffic arriving at a link should be below its capacity. We believe that our dependency graphs are general enough to describe other properties as well, which may be proposed in the future.

We now describe dependency graph generation. We first focus on tunnel-based forwarding without resource limits and then discuss WCMP forwarding and resource constraints.

Tunnel-based forwarding: Tunnel-based forwarding offers loop freedom and packet coherence by design; it is not possible for packets to loop or to see a mix of old and new rules during updates. We defer discussion of congestion freedom until we discuss resource constraints. The remaining property, blackhole freedom, is guaranteed as long as we ensure that $i)$ a tunnel is fully established before the ingress switch puts any traffic on it, and $ii)$ all traffic is removed from the tunnel before the tunnel is deleted.

A dependency graph that encodes these constraints can be built as follows. For each flow f, using NS_c and NS_t, we first calculate the tunnels to be added and deleted and generate a path node for each. Then, we generate an operation node for every hop, adding an edge from each of them to the path node (or from the path node to each of them), denoting adding (or deleting) this tunnel at the switch. Then, we generate an operation node that changes the tunnel weights to those in NS_t at the ingress switch. To ensure blackhole freedom, we add an edge from each path node that adds new tunnels to the operation node that changes tunnel weights, and an edge from the operation node that changes tunnel weights to each path node that deletes old tunnels.

We use the example in Figure 6 to illustrate the steps above. Initially, we set the tunnel weights on p_1 and p_2 with 0.5 and 0.5 respectively. In the target state, we add tunnel p_3, delete tunnel p_2, and change the tunnel weights to 0.5 on p_1 and 0.5 on p_3. To generate the dependency graph for this transition, we first generate path nodes for $p2$ and $p3$ and the related switch operations as in Table 1. Then we add edges from the tunnel-addition operations (A, B and C) to the corresponding path node ($p3$), and edges to the tunnel-deletion operations (E, F and G) from the corresponding path node ($p2$). Finally, we add an edge from the path node of the added path ($p3$) to the weight-changing operation (D) and from D to the path node for the path to be deleted ($p2$). The resulting graph is shown in Figure 6(c). The resource nodes in this graph are discussed later.

WCMP forwarding: With NS_c and NS_t, we calculate for each flow the weight change operations that update the network from NS_c to NS_t. We then create dependency edges between these operations based on the consistency property. Algorithm 1 shows how to do that for packet-coherence, using version numbers [10, 11]. In this approach, the ingress switch tags each packet with a version number and downstream switches handle packets based on the embedded version number. This tagging ensures that each packet either uses the old configuration or the new configuration, and never a mix of the two. The algorithm generates three types of operations: $i)$ the ingress switch tags packets with the new version number and uses new weights (Line 3); $ii)$ downstream switches have rules for handling the packets with the new version number and new weights (Line 6); and $iii)$ downstream switches delete rules for the old version number (Line 7). Packet coherence is guaranteed if Type i operation occurs after Type ii (Line 8) and Type iii operations occur after Type i (Line 9). Line 5 is an optimization; no changes are needed at switches that have only one next hop for the flow in both the old and new configurations.

Algorithm 1 Dependency graph for packet coherence in a WCMP network

```
   – $v_0$: old version number
   – $v_1$: new version number
1: for each flow $f$ do
2:     $s^* = GetIngressSwitch(f)$
3:     $o^* = GenRuleModifyOp(s^*, v_1)$
4:     for $s_i \in GetAllSwitches(f) - s^*$ do
5:         if $s_i$ has multiple next-hops then
6:             $o_1 = GenRuleInsertOp(s_i, v_1)$
7:             $o_2 = GenRuleDeleteOp(s_i, v_0)$
8:             Add edge from $o_1$ to $o^*$
9:             Add edge from $o^*$ to $o_2$
```

We use the example in Figure 6 again to illustrate the algorithm above. For flow f, we need to update the flow weights at $S1$ from $[(S2, 1), (S4, 0)]$ to $[(S2, 0.5), (S4, 0.5)]$, and weights at $S2$ from $[(S3, 0.5), (S5, 0.5)]$ to $[(S3, 1), (S5, 0)]$. This translates to three operations (Table 2): add new weights with new version numbers at $S2$ (X), change to new weights and new version numbers at $S1$ (Y), and delete old weights at $S2$ (Z). We connect X to Y and Y to Z as shown in Figure 6(d).

Blackhole-freedom and loop-freedom do not require version numbers. For the former, we must ensure that every switch that may receive a packet from a flows always has a rule for it. For the latter, we must ensure that downstream switches (per new configuration) are updated before updating a switch to new rules [13]. These conditions are easy to encode in a dependency graph. For space constraints, we omit detailed description of graph construction.

Resource constraints: We introduce resource nodes to the graph corresponding to resources of interest, including link bandwidth and switch memory. These nodes are labeled with their current free amount or with infinity if that resource can never be a bottleneck.

We connect link bandwidth nodes with other nodes as follows. For each path node and bandwidth node for links along the path: if the traffic on the path increases, we add an edge from the bandwidth node to the path node with a label indicating the amount of traffic increase; if the traffic decreases, we add edges in the other direction. For a tunnel-based network, we add an edge from each path node on which traffic increases to the operation node that changes weight at the ingress switch with a label indicating the amount of traffic increase; similarly, we add an edge in the other direction if the traffic decreases. For a WCMP network, we add an edge from each path node on which traffic increases to each operation node that *adds weights with new versions* with a label indicating the amount of increase; similarly, we add an edge from the oper-

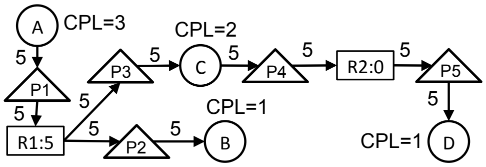


Figure 8: Critical-path scheduling. C has larger CPL than B, and is scheduled.

ation node that *changes weight at the ingress switch* to each path node on which traffic decreases with a label indicating the amount of decrease. This difference is due to that tunnels offer packet coherence by design, while WCMP networks need version numbers.

Connecting switch memory resource nodes with other nodes is straightforward. We add an edge from a resource node to an operation node if the operation consumes that switch memory with an weight indicating the amount of consumption; we add an edge from an operation node to a resource node if the operation releases that switch memory with an weight indicating the amount of release.

For example, in Figure 6(c) node D, which changes tunnel weights at $S1$, increases 5 units of traffic on p_3 which includes link $S1$-$S4$ and $S4$-$S5$, and decreases 5 units of traffic on p_2 which includes link $S1$-$S2$ and $S2$-$S5$. Node A that adds tunnel p_3 consumes 1 rule at $S1$. In Figure 6(d), we link p_3 to X and link X to Y. X and Y essentially takes the same effect as D in Figure 6(d).

Post-processing: After generating the dependency graph, we reduce it by deleting edges from non-bottlenecked resources. For each resource node R_i, we check the edges to its child nodes N_j. If the free resource $R_i.free$ is no smaller than $\sum_j l_{ij}$ where l_{ij} is the edge weight, we delete all the edges from R_i to its children and decrease the free capacity by $\sum_j l_{ij}$. The idea is that R_i has enough free resource to accommodate all operations that need it, so it's not a bottleneck resource and the scheduling will not consider it. For example, if $S1$-$S4$ has over 5 units of free capacity, we can delete the edge from $S1$-$S4$ to p_3 in Figures 6(c) and 6(d).

6. DIONYSUS SCHEDULING

We now describe how updates are scheduled in Dionysus. First, we discuss the hardness of the scheduling problem, which guided our approach. Then, we describe scheduling algorithm for the special case where the dependency graph is a DAG (directed acyclic graph). Finally, we extend this algorithm to handle cycles.

6.1 The hardness of the scheduling problem

Scheduling is a resource allocation problem, that is, how to allocate available resources to operations to minimize the update time. For example, resource node $R1$ in Figure 8 has 5 units of free resource. It cannot cover both B and C. We must decide to schedule $i)$ B, $ii)$ C, or $iii)$ part of B and C. Every time we make a scheduling decision, we decide how to allocate a resource to its child operations and which parent operation to execute to obtain a resource. Additional constraints on scheduling are placed by dependencies between operations.

We can prove the following about network update scheduling.

THEOREM 1. *In the presence of both link capacity and switch memory constraints, finding a feasible update schedule is NP-complete.*

PROOF. See Appendix A. □

The hardness stems from the fact that memory constraints involve integers and memory cannot be allocated fractionally. Scheduling is simpler if we only have link capacity constraints, but finding the fastest schedule is still hard because of the huge search space.

Symbol	Description
O_i	Operation node i
R_j	Resource node j
$R_j.free$	Free capacity of R_j
P_k	Path node k
$P_k.committed$	Traffic that is moving away from path k
l_{ij}	Edge weight from node i to j

Table 3: Key notation in our algorithms.

Algorithm 2 ScheduleGraph(G)

```
1: while true do
2:     $UpdateGraph(G)$
3:     Calculate $CPL$ for every node
4:     Sort nodes by $CPL$ in decreasing order
5:     for unscheduled operation node $O_i \in G$ do
6:         if $CanScheduleOperation(O_i)$ then
7:             Schedule $O_i$
8:     Wait for time $t$ or for all scheduled operations to finish
```

THEOREM 2. *In the presence of link capacity constraints, but no switch memory constraints, finding the fastest update schedule is NP-complete.*

PROOF. See Appendix A. □

6.2 Scheduling DAGs

We first consider the special case of a DAG. Scheduling a DAG is, expectedly, simpler:

LEMMA 1. *If the dependency graph is a DAG, finding a feasible update schedule is in P.*

While it is easy to find a feasible solution for a DAG, we want to find a fast one. Different scheduling orders lead to different finishing times. For example, if all operations take the same amount of time Figure 8, scheduling C before B will be faster.

We use *critical-path scheduling*. The intuition is that the critical path decides the completion time, and we thus want to schedule operations on the critical path first. Since resource nodes and path nodes in the dependency graph are only used to express constraints, we assign weight w=0 to them when calculating critical paths; for operation nodes, we assign weight w=1. With this, we calculate a critical-path length CPL for each node i as:

$$CPL_i = w_i + \max_{j \in children(i)} CPL_j \qquad (1)$$

To calculate CPL for all the nodes in the graph, we first topologically sort all the nodes and then iterate over them to calculate CPL with Equation 1 in the reverse topological order. In Figure 8, for example, CPL_D=1, CPL_C=2, CPL_B=1, CPL_A=3. The CPL for each node can be computed efficiently in linear time.

Algorithm 2 shows how Dionysus uses CPL to schedule updates, with key notations summarized in Table 3. Each time we enter the scheduling phase, we first update the graph with finished operations and delete edges from unbottlenecked resources (line 2). Then, we calculate CPL for every node (Line 3) and sort nodes in decreasing order of CPL (Line 4). Then, we iterate over operation nodes and schedule them if their operation dependency and resource dependency are satisfied (Lines 6, 7). Finally, the scheduler waits for some time for all scheduled operations to finish before starting the next round (Line 10).

To simplify presentation, we first show the related pseudo code of $CanScheduleOperation(O_i)$ and $UpdateGraph(G)$ for tunnel-based networks and describe them below. Then, we briefly describe how the WCMP case differs.

Algorithm 3 CanScheduleOperation(O_i)

```
    // Add tunnel operation node
 1: if $O_i.isAddTunnelOp()$ then
 2:     if $O_i.hasNoParents()$ then
 3:         return true
 4:     $R_j \leftarrow parent(O_i)$ // AddTunnelOp only has 1 parent
 5:     if $R_j.free \geq l_{ji}$ then
 6:         $R_j.free \leftarrow R_j.free - l_{ji}$
 7:         Delete edge $R_j \rightarrow O_i$
 8:         return true
 9:     return false
    // Delete tunnel operation node
10: if $O_i.isDelTunnelOp()$ then
11:     if $O_i.hasNoParents()$ then
12:         return true
13:     return false
    // Change weight operation node
14: $total \leftarrow 0$
15: $canSchedule \leftarrow false$
16: for path node $P_j \in parents(O_i)$ do
17:     $available \leftarrow l_{ji}$
18:     if $P_j.hasOpParents()$ then
19:         $available \leftarrow 0$
20:     else
21:         for resource node $R_k \in parents(P_j)$ do
22:             $available \leftarrow min(available, l_{kj}, R_k.free)$
23:         for resource node $R_k \in parents(P_j)$ do
24:             $l_{kj} \leftarrow l_{kj} - available$
25:             $R_k.free \leftarrow R_k.free - available$
26:     $total \leftarrow total + available$
27:     $l_{ji} \leftarrow l_{ji} - available$
28: if $total > 0$ then
29:     $canSchedule \leftarrow true$
30: for path node $P_j \in children(O_i)$ do
31:     $P_j.committed \leftarrow min(l_{ij}, total)$
32:     $l_{ij} \leftarrow l_{ij} - P_j.committed$
33:     $total \leftarrow total - P_j.committed$
34: return $canSchedule$
```

Algorithm 4 UpdateGraph(G)

```
 1: for finished operation node $O_i \in G$ do
        // Finish add tunnel operation node
 2:     if $O_i.isAddTunneOp()$ then
 3:         Delete $O_i$ and all its edges
        // Finish delete tunnel operation node
 4:     else if $O_i.isDelTunnelOp()$ then
 5:         $R_j \leftarrow child(O_i)$
 6:         $R_j.free \leftarrow R_j.free + l_{ij}$
 7:         Delete $O_i$ and all its edges // DelTunnelOp only has 1 child
        // Finish change weight operation node
 8:     else
 9:         for path node $P_j \in children(O_i)$ do
10:             for resource node $R_k \in children(P_j)$ do
11:                 $l_{jk} \leftarrow l_{jk} - P_j.committed$
12:                 $R_k.free \leftarrow R_k.free + P_j.committed$
13:                 if $l_{jk} = 0$ then
14:                     Delete edge $P_j \rightarrow R_k$
15:             $P_j.committed \leftarrow 0$
16:             if $l_{ij} = 0$ then
17:                 Delete $P_j$ and its edges
18:         for path node $P_j \in parents(O_i)$ do
19:             if $l_{ji} = 0$ then
20:                 Delete $P_j$ and its edges
21:         if $O_i.hasNoParents()$ then
22:             Delete $O_i$ and its edges
23: for resource node $R_i \in G$ do
24:     if $R_i.free \geq \sum_j l_{ij}$ then
25:         $R_i.free \leftarrow R_i.free - \sum_j l_{ij}$
26:         Delete all edges from $R_i$
```

CanScheduleOperation (Algorithm 3): This function decides if an operation O_i is ready to be scheduled and updates the resource levels for resource and path nodes accordingly. If O_i is a tunnel addition operation, we can schedule it either if it has no parents (Lines 2, 3) or its parent resource node has enough free resource (Lines 4–8). If O_i is a tunnel deletion operation, we can schedule it if it has no parents (Lines 11–12); tunnel deletion operations do not have resource nodes as parents because they always release (memory) resources. If O_i is a weight change operation, we gather all free capacities on the paths where traffic increases and moves traffic to them (line 14-34). We iterate over each parent path node and obtain the available capacity ($available$) of the path (Lines 16–27). This capacity limits the amount of traffic that we can move to this path. We sum them up to $total$, which is the total traffic we can move for this flow (Line 26). Then, we iterate over child path nodes (Lines 30–33). Finally, we decrease $P_j.committed$ traffic on path represented by P_j (Line 31).

UpdateGraph (Algorithm 4): This function updates the graph before scheduling based on operations that successfully finished in the last round. We get all such operations and update related nodes in the graph (Lines 1–22). If the operation node adds a tunnel, we delete the node and its edges (Lines 2, 3). If the operation node deletes a tunnel, it frees rule space. So, we update the resource node (Lines 5, 6) and delete it (Line 7). If the operation node changes weight, for each child path node, we release resources to links on it (Lines 11–12) and delete the edge if all resources are released (Lines 13, 14). We reset the amount of traffic that is moving away from this path, $P_j.committed$, to 0 (Line 15). If we have moved all the traffic away from this path, we delete this path node (Lines 16, 17). Similarly, we check all the parent path nodes (Lines 18–20). If we have moved all the traffic into a path, we delete the path node (Lines 19, 20). Finally, if all parent path nodes are removed, the weight change for this flow finishes; we remove it from the graph (Line 22). After updating the graph with finished operations, we check all resource nodes (Lines 23–26). We delete edges from unbottlenecked resources (Lines 24–26).

WCMP network: Algorithms 3 and 4 for WCMP-based networks differ in two respects. First, WCMP networks do not have tunnel add or delete operations. Second, unlike tunnel-based networks that can simply change the weights at the ingress switches, WCMP networks perform a two-phase commit using version numbers to maintain packet coherence (node X and Y in Figure 6(d)). The code related to the weight change operation in the two algorithms has minor difference accordingly. The details can be found in Appendix B.

6.3 Handling cycles

Cycles in the dependency graph pose a challenge because inappropriate scheduling can lead to deadlocks where no progress can be made, as we saw for Figure 5(b) if $F2$ is moved first. Further, many cycles may intertwine together, which makes the problem even more complicated. For instance, A, B and C are involved in several cycles in Figure 9.

We handle dependency graphs with cycles by first transforming them into a virtual DAG and then using the DAG scheduling algorithm above. We use the concept of a strongly connected component (SCC), a subgraph where every node has a path to every other node [14]. One can think of an SCC as a set of intertwined cycles. If we view each SCC as a virtual node in the graph, then the graph becomes a virtual DAG, which is called the component graph in

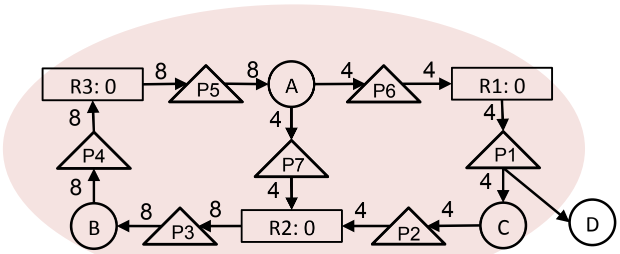


Figure 9: A deadlock example where the target state is valid but no feasible solution exists.

Algorithm 5 RateLimit(SCC, k^*)

```
1: $O^* \leftarrow$ weight change nodes $\in SCC$
2: for $i$=0 ; $i$<$k^*$ && $O^* \neq \emptyset$ ; $i$++ do
3:     $O_i \leftarrow O^*.pop()$
4:     for path node $P_j \in children(O_i)$ do
           // $f_i$ is the corresponding flow of $O_i$
5:         Rate limit flow $f_i$ by $l_{ij}$ on path $P_j$
6:         for resource node $R_k \in children(P_j)$ do
7:             $R_k.free \leftarrow R_k.free + l_{ij}$
8:         Delete $P_j$ and its edges
```

graph theory. We use Tarjan's algorithm [20] to efficiently find all SCCs in the dependency graph. Its time complexity is $O(|V|+|E|)$ where $|V|$ and $|E|$ are the number of nodes and edges.

With each SCC being a virtual node, we can use critical-path scheduling on the component graph. While calculating $CPLs$, we use the number of operation nodes in an SCC as the weight of the corresponding virtual node, which makes the scheduler prefer paths with larger SCCs.

We make two modifications to the scheduling algorithm to incorporate SCCs. The first is that the *for* loop at Line 5 in Algorithm 2 iterates over all nodes in the virtual graph. When a node is selected, if it is a single node, we directly call $CanScheduleOperation(O_i)$. If it is a virtual node, we iterate over the operation nodes in its SCC and call the functions accordingly. We use centrality [21] to decide the order of the iteration; the intuition is that a central node of an SCC is on many cycles, and if we can schedule this node early, many cycles will disappear and we can finish the SCC quickly. We use the popular outdegree-based definition of centrality, but other definitions may also be used. The second modification is that when path nodes consume link resources or tunnel add operations consume switch resources, they can only consume resources from nodes that either are in the same SCC or are independent nodes (not in any SCC). This heuristic prevents deadlocks caused by allocating resources to nodes outside the SCC ("Mv. F2") before nodes in the SCC are satisfied as in Figure 5(b).

Deadlocks: The scheduling algorithm resolves most cycles without deadlocks (§9). However, we may still encounter deadlocks in which no operations in the SCC can make any progress even if the SCC have obtained all resources from outside nodes. This can happen because (1) given the hardness of the problem, our scheduling algorithm, which is basically an informed heuristic, doesn't find the feasible solution among the combinatorially many orderings and gets stuck, or (2) there does not exist a feasible solution even if the target state is valid, like the example in Figure 9. One should note that deadlocks stem from the need for consistent network updates. Previous solutions face the same challenge but are much slower and cause more congestion than Dionysus (§9.4).

Our strategy for resolving deadlocks is to reduce flow rates (e.g., by informing rate limiters). Reducing flow rate frees up link capacity; and reducing it to zero on a path allows removal of the tunnel, which in turn frees up switch memory. Freeing up these resources allows some of the operations that were earlier blocked on resources to go through. In the extreme case, if we rate limit all the flows involved in the deadlocked SCC, the deadlock can be resolved in one step. However, this extreme remedy leads to excessive throughput loss. It is also unnecessary because often rate limiting a few strategically selected flows suffices.

We thus rate limit a few flows to begin with, which enables some operations in the SCC to be scheduled. If that does not fully resolve the SCC, we rate limit a few more, until the SCC is fully resolved. The parameter k^* determines the maximum number of flows that we rate limit each time, and it controls the tradeoff between the time to resolve the deadlock and the amount of throughput loss. Algorithm 5 shows the procedure to resolve deadlocks for tunnel-based networks. It iterates over up to k^* weight change nodes in the SCC, each of which corresponds to a flow (Lines 2–8). The order of iteration is based on centrality value as above.

We use Figure 9 to illustrate deadlock resolution. Let k^*=1. The procedure first selects node A. It reduces 4 units of traffic on path $P6$ and 4 units on $P7$, which releases 4 units of free capacity to $R1$ and 4 units to $R2$, and deletes $P6$ and $P7$. At this point, node A has no children and thus does not belong to the SCC any more. After this, we call $ScheduleGraph(G)$ to continue the update. It schedules C, and partially schedules B (i.e., moves 4 units of traffic from path $P3$ to $P4$). After C finishes, it schedules the remainded of operation B and finishes the update. Finally, for node A and its corresponding flow f_A, we increase its rate on $P5$ as long as $R3$ receives free capacity released by $P4$.

We have the following theorem to prove that as long as the target state is valid (i.e., no resource is oversubscribed), we can fully resolve a deadlock using the procedure above.

THEOREM 3. *If the target state is valid, a deadlock can be always resolved by calling $RateLimit$ a finite number of steps.*

PROOF. Each time we call $RateLimit(SCC, k^*)$, the deadlock reduces by at least k^* number of operations. Let O^* be the number of operations in the deadlock. We can resolve the deadlock by at most $\lceil O^*/k^* \rceil$ iterations of $RateLimit$. □

We find experimentally that often the number of steps needed is a lot fewer than the bound above.

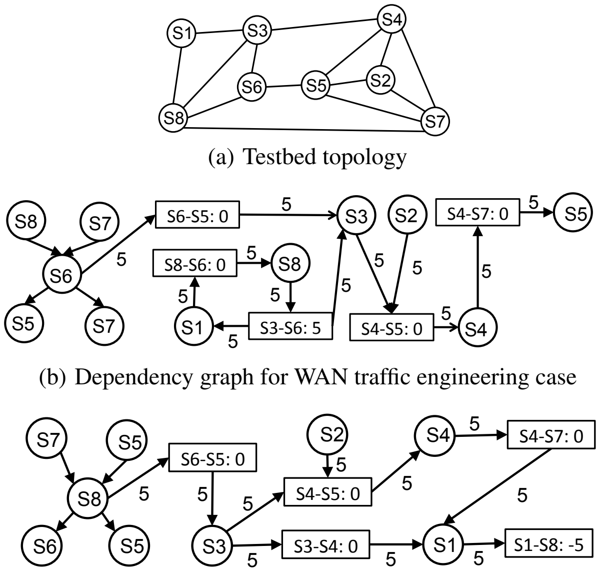
7. IMPLEMENTATION

We have implemented a prototype of Dionysus with 5,000+ lines of C# code. It receives current state from the network and target state from applications as input, generates a dependency graph, and schedules rule updates. We implemented dependency graph generators for both tunnel-based and WCMP networks and all the scheduling algorithms discussed above. For accurate control plane confirmations of rule updates (not available in most OpenFlow agents today), we run a custom software agent on our switches.

8. TESTBED EVALUATION

We evaluate Dionysus using testbed experiments in this section and using large-scale simulations in the next section. We use two update cases, a WAN TE case and a WAN failure recovery case. To show its benefits, we compare Dionysus against SWAN [9], a static solution.

Methodology: Our testbed consists of 8 Arista 7050T switches as shown in Figure 10(a). It emulates a WAN scenario. The switches



(a) Testbed topology

(b) Dependency graph for WAN traffic engineering case

(c) Dependency graph for WAN failure recovery case

Figure 10: Testbed setup. Path nodes are removed from the dependency graphs ((b) and (c)) for brevity.

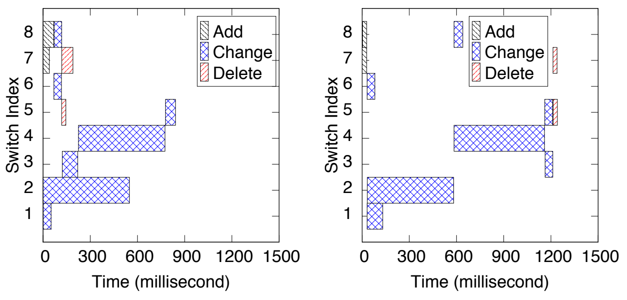
are connected by 10 Gbps links. With the help of our switch agents, we log the time of sending updates and receiving confirmation. We use VLAN tags to implement tunnels and use prefix-splitting to implement weights when a flow uses multiple tunnels. We let $S2$ and $S4$ be straggler switches and inject 500 ms latency for rule updates on them. The remaining switches update at their own pace.

WAN TE case: In this experiment, the update is triggered by a traffic matrix change. TE calculates a new allocation for the new matrix, and we update the network accordingly. A simplified dependency graph for this update is shown in Figure 10(b). Numbers in the circles correspond to the switch to which the rule update is sent. For example, the operation node with annotation "S8" means a rule update at switch $S8$. The graph contains a cycle that includes nodes "S8", "S3-S6", "S1" and "S8-S6". Careless scheduling, e.g., one that schedules node "S3" before "S1" may cause a deadlock. There are also operation dependencies for this update: to move a flow at $S6$, we have to install a new tunnel at $S8$ and $S7$; after the movement finishes, we delete an old tunnel at $S5$ and $S7$.

Figure 11 shows the time series of this experiment. The x-axis is the time, and the y-axis is the switch index. A rectangle represents a rule update on a switch (y-axis) for some period (x-axis). Different rectangular patterns show different rule update operations (add rule, change rule, or delete rule). Rule updates on straggler switches, $S2$ and $S4$, take longer than those on other switches. But even on non-straggler switches, the rule update time varies—the lengths of the rectangles are not identical—between 20 and 100 ms.

Dionysus dynamically performs the update as shown in Figure 11(a). First it finds the SCC and schedules node "S1". It also schedules "S2", "S8" and "S7" as they don't have any parents. After they finish, Dionysus schedules "S6" and "S8", then "S3", "S5" and "S7". Rather than waiting for "S2," which is a straggler, Dionysus schedules "S4" after "S3" finishes—"S3" releases enough capacity for it. Finally Dionysus schedules "S5". The update finishes in 842 ms.

SWAN uses a static, multi-step solution to perform the update (Figure 11(b)). It first installs the new tunnel (node "S8" and "S7"). Then, it adjusts tunnel weights with a congestion-free plan with the minimal number of steps, as follows:



(a) Dionysus

(b) SWAN

Figure 11: Time series for testbed experiment of WAN TE.

Step 1: "S1", "S6", "S2"
Step 2: "S4", "S8"
Step 3: "S3", "S5"

Due to stragglers $S2$ and $S4$, SWAN takes a long time on both Steps 1 and 2. Finally, SWAN deletes the old tunnel (node "S5" and "S7"). It does not start the tunnel addition and deletion steps with the weight change steps. The whole update takes 1241 ms, 47% longer than Dionysus.

WAN failure recovery case: In this experiment, the network update is triggered by a topology change. Link S3-S8 fails; flows that use this link rescale their traffic to other tunnels. This causes link S1-S8 to get overloaded by 50%. To address this problem, TE calculates a new traffic allocation that eliminates the link overload. The simplified dependency graph for this network update is shown in Figure 10(c). To eliminate the overload on link S1-S8, a flow at $S1$ is to be moved away, which depends on several other rule updates. Doing all the rule updates in one shot is undesirable as it may cause more link overloads and affect more flows. For example, if "S1" finishes faster than "S3" and "S4", then it causes 50% link overload on link S3-S4 and S4-S7 and unnecessarily brings congestions to flows on these links. We present extensive results in §9.3 to show that one-shot updates can cause excessive congestion.

Figure 12(a) shows the time series of the update performed by Dionysus. It first schedules nodes "S7", "S5" and "S2". After "S7" and "S5" finish, a new tunnel is established and it safely schedules "S8". Then it schedules "S3", "S5" and "S6". Although "S2" is on a straggler switch and is delayed, Dionysus dynamically schedules "S4" once "S3" finishes. Finally, it schedules "S1". It finishes the update in 808 ms, which eliminates the overload on S1-S8, as shown in Figure 12(c).

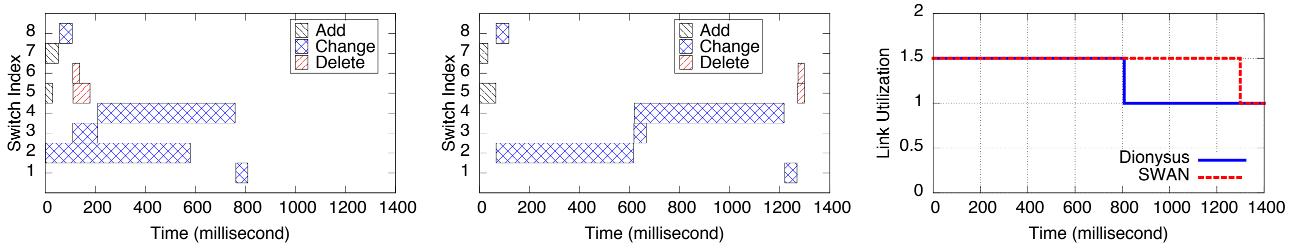
Figure 12(b) shows the time series of the update performed by SWAN. It first installs the new tunnel (node "S7" and "S5"), then calculates an update plan with minimal steps as follows.

Step 1: node "S2", node "S8"
Step 2: node "S3", node "S4"
Step 3: node "S1"

This static plan does not adapt, and it is delayed by straggler switches at both Steps 1 and 2. It misses the opportunity to dynamically reorder rule updates. It takes 1299 ms to finish the update and eliminate the link overload, 61% longer than Dionysus.

9. LARGE-SCALE SIMULATIONS

We now conduct large-scale simulations to show that Dionysus can significantly improve update speed, reduce congestion, and effectively handle cycles in dependency graphs. We focus on congestion freedom as the consistency property, a particularly challenging property and most relevant for the networks we study.



(a) Dionysus (b) SWAN (c) Link Utilization on Link $S1$-$S8$

Figure 12: Time series for testbed experiment of WAN failure recovery.

9.1 Datasets and methodology

Wide area network: This dataset is from a large WAN that interconnects $O(50)$ sites. Inter-site links have tens to hundreds of Gbps capacity. We collect traffic logs on routers and aggregate them into site-to-site flows over 5-minute intervals. The flows are classified into 3 priorities: interactive, elastic and background [9]. We obtain 288 traffic matrices on a typical working day, where each traffic matrix consists of all the site-to-site flows in one interval.

The network uses tunnel-based routing, and we implement the TE algorithm of SWAN [9] which maximizes network throughput and approximates max-min fairness among flows of the same priorities. The TE algorithm produces the network configuration for successive intervals and we compute the time to update the network from one interval to the next.

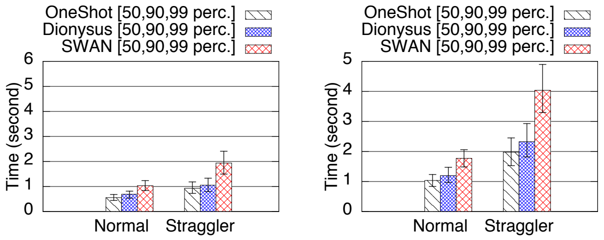
Data center network: This dataset is from a large data center network with several hundred switches. The topoogy has 3 layers: ToR (Top-of-Rack), Agg (Aggeration), and Core. Links between switches are 10 Gbps. We collect traffic traces by logging the socket events on all servers and aggregate them into ToR-to-ToR flows over 5-minute intervals. As for the WAN, we obtain 288 traffic matrices for a typical working day.

Due to the large scale, we do elephant-flow routing [5, 6, 7]. We choose the 1500 largest flows, which account for 40–60% of all traffic. We use an LP to calculate their traffic allocation and use ECMP for other flows. This method improves the total throughput by up to 30% as compared to using ECMP for all flows. We run TE and update WCMP weights for elephant flows every interval. Since mice flows use default ECMP entries, nothing is updated for them.

For both settings, we leave 10% scratch capacity on links to aid transitions [9], and we use 1500 as switch rule memory size. This memory size means that the memory slack (i.e., unused capacity) is at least 50% in our experiments in §9.2 and §9.3. In §9.4, we study the impact of memory limitation by reducing memory size.

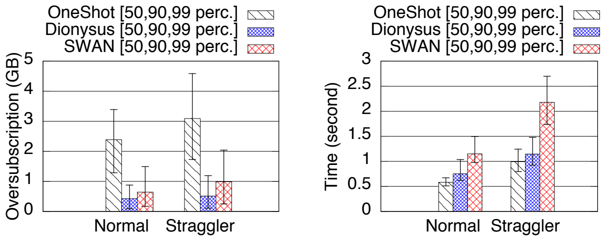
Alternative approaches: We compare Dionysus with two alternative approaches. First, *OneShot* sends all updates in one shot. It does not maintain any consistency, but serves as the lower bound for update time. Second, *SWAN* is the state-of-the-art approach in maintaining congestion freedom [9]. It uses a heuristic to divide the update into multiple phases based on memory constraints so that each intermediate phase can fit all rules in switches. SWAN may rate limit flows in intermediate phase as the paths in the network cannot carry all the traffic. Between consecutive phases, it uses a linear program to calculate a congestion-free multi-step plan based on capacity constraints.

Rule update time: The rule update time at switches is based on switch measurement results (§2). We show results in both *normal* setting and *straggler* setting. In the former case, we use the median rule update time in §2; in the latter case, we draw rule update time from the CDF in §2. We use 50 ms as RTT in WAN scenario.



(a) WAN TE (b) Data center TE

Figure 13: Dionysus **is faster than SWAN and close to OneShot.**



(a) Link Oversubscription (b) Update time

Figure 14: In WAN failure recovery, Dionysus **significantly reduces oversubscription and update time as compared to SWAN. OneShot, while fast, incurs huge oversubscription.**

9.2 Update time

WAN TE: Figure 13(a) shows the 50^{th}, 90^{th}, 99^{th} percentile update time across all intervals for the WAN TE scenario. Dionysus outperforms SWAN in both normal and straggler settings. In the normal setting, Dionysus is 57%, 49%, and 52% faster than SWAN in the 50^{th}, 90^{th}, 99^{th} percentile, respectively. The gain is mainly from pipelining: in every step, different switches receive different number of rules to update and thus takes different amount of time to finish. While SWAN has to wait for the switch with the most number of rules to finish, Dionysus begins to issue new operations as soon as some switches finish.

In the straggler setting, Dionysus reduces update time even more. It is 88%, 84%, and 81% faster than SWAN in the 50^{th}, 90^{th}, 99^{th} percentile, respectively. This advantage is because stragglers provide more opportunities for dynamic scheduling which SWAN cannot leverage. Dionysus also performs close to OneShot. It is only 25% and 13% slower than OneShot in the 90^{th} percentile in normal and straggler settings, respectively.

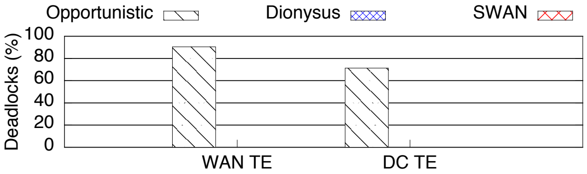


Figure 15: Opportunistic scheduling frequently deadlocks. Dionysus and SWAN have no deadlocks.

Data center TE: Figure 13(b) shows results for the data center TE scenario. Again, Dionysus significantly outperforms SWAN. In the normal setting, it is 53%, 48%, and 40% faster than SWAN in the 50^{th}, 90^{th}, and 99^{th} percentile; in the straggler setting, it is 81%, 74%, and 67% faster. Data center TE takes more time because it involves a two-phase commit across multiple switches for each flow; WAN TE only needs to update the ingress switch if all tunnels are established.

9.3 Link oversubscription

We use a WAN failure recovery scenario to show that Dionysus can reduce link oversubscription and shorten recovery time. We use the same topology and traffic matrices as in the WAN TE case. For each traffic matrix, we first use TE to calculate a state NS_0. Then we fail a randomly selected link, which causes the ingress switches to move traffic away from the failed tunnels to the remaining ones. For example, if flow f originally uses tunnels T_1, T_2 and T_3 with weights w_1, w_2 and w_3 and the failed link causes T_1 to break, then f carries its traffic using T_2 and T_3 with weights $w_2/(w_2+w_3)$ and $w_3/(w_2+w_3)$. We denote the network state that emerges after the failure and rescaling as NS_1. Since rescaling is a local action, NS_1 may have overloaded links. The TE calculates a new state NS_2 to eliminate congestion. The network update that we study is the update from NS_1 to NS_2.

If the initial state NS_1 already has congestion, there will be no congestion-free update plan. For Dionysus and SWAN, we generate plans in which, during updates, no oversubscribed link carries more load than its current load. In such plans, the capacity of congested links is virtually increased to its current load, to make each link appear non-congested. For Dionysus, we increase the weight of overloaded links to the overloaded amount in CPL calculation (Equation 1). Then, Dionysus will prefer operations that move traffic away from overloaded links. For SWAN, we use the linear program to compute the plan such that total oversubscription across all links is minimized at each step. Of all possible static plans, this modification makes SWAN prefer one that minimizes congestion quickly. OneShot operates as before because it does not care about congestion.

Figure 14 shows the update time and link oversubscription—the amount of data above capacity arriving at a link. Dionysus has the least oversubscription among the three. OneShot, while quick, has huge oversubscription. SWAN incurs 1.49 GB and 2.04 GB oversubscription in the 99^{th} percentile in normal and straggler settings, respectively. As even high-end switches today only have hundreds of MB buffer [22], such oversubscription will cause heavy packet loss. Dionysus reduces oversubscription to 0.88 GB and 1.19 GB, which are 41% and 42% less than SWAN. For update time, Dionysus is 45% and 82% faster than SWAN in the 99^{th} percentile in normal and straggler setting, respectively.

9.4 Deadlocks

We now study the effectiveness of Dionysus in handling circular dependencies, which can lead to deadlocks. First, we show that, as mentioned in §3, completely opportunistic scheduling can lead to frequent deadlocks even in a setting that is not resource-constrained. Then, we show the effectiveness of Dionysus in handling resource-constrained settings.

Figure 15 shows the percentage of network updates finished by Dionysus, SWAN, and an opportunistic approach without deadlocks, that is, without having to reduce flow rates during updates. The opportunistic approach immediately issues any updates that do not violate consistency (§3), instead of planning using a dependency graph. The data in the figure corresponds to the WAN and data center TE scenarios in §9.2, where the memory slack was over 50%. We do not show results for OneShot; it does not deadlock by design as it does not worry about consistency.

We see that planning-based approaches, Dionysus and SWAN, lead to no deadlocks, but the opportunistic approach deadlocks 90% of the time for WAN TE and 70% of the time for data center TE. It performs worse for WAN TE because the WAN topology is less regular than the data center topology, which leads to more complex dependencies.

We now evaluate Dionysus and SWAN in resource-constrained settings. To emulate such a setting, instead of using 1500 as memory size, we vary switch memory slack; 10% memory slack means we set the memory size as 1100 when the switch is loaded with 1000 rules. We show three metrics in the WAN TE setup: (1) the percentage of cases that deadlock and use rate limiting to finish the update, (2) the throughput loss caused by rate limiting (i.e., the product of the limited rate and the rate limited time), and (3) the update time. We set k^*=5 in Algorithm 5 for Dionysus.

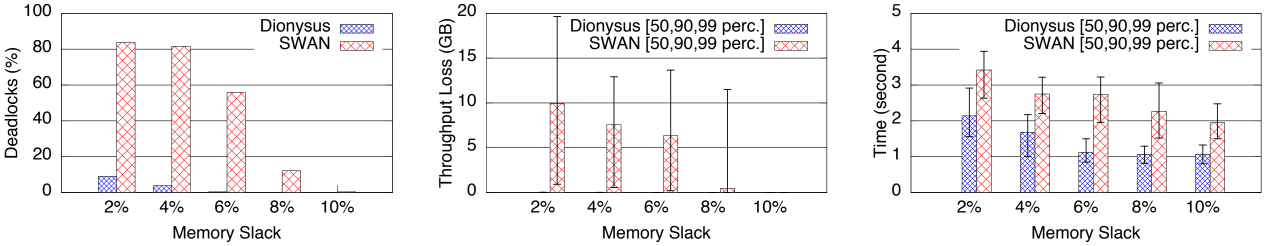
Figure 16 shows the results for the straggler setting. The results with the normal setting are similar. Figure 16(a) shows the percentage of cases that use rate limiting under different levels of memory slack. Dionysus only occasionally runs into deadlocks and uses resorts to rate limiting more sparingly than SWAN. Even with only 2% memory slack, Dionysus uses rate limiting in fewer than 10% cases. SWAN, on the other hand, uses rate limiting in more than 80% of the cases. This difference is because the heuristics in Dionysus strategically account for dependencies during scheduling. SWAN uses simplistic metrics, such as the amount of traffic that a tunnel carries and the number of hops of the tunnel, to decide which tunnel to add or delete.

Figure 16(b) shows the throughput loss. The throughput loss with SWAN can be as high as 20 GB, while that with Dionysus is only tens of MB. Finally 16(c) shows the update time. Dionysus is 60%, 145%, and 84% faster than SWAN in the 90^{th} percentile under 2%, 6% and 10% memory slack respectively.

10. RELATED WORK

In the domain of distributed protocols, there is a lot of work on avoiding transient misbehavior during network updates. Much focuses on maintain properties like loop-freedom for specific protocols or scenarios. For example, Francois *et al.* [23], John *et al.* [24] and Kushman *et al.* [25] focus on BGP, Francois *et al.* [26, 27] and Raza *et al.* [28] focus on link-state protocols, and Vanbever *et al.* [29] focus on migration from one routing protocol to another.

With the advent of SDN, many recent works propose solutions to maintain different consistency properties during network updates. Reitblatt *et al.* [11] provide a theoretical foundation and propose a two-phase commit protocol to maintain packet coherence. Katta *et al.* [12] and McGeer *et al.* [30] propose solutions to reduce the memory requirements to maintain packet coherence. SWAN [9], zUpdate [10] and Ghorbani and Caesar [31] provide solutions for congestion-free updates. Noyes *et al.* [32] propose a model checking based approach to generate update orderings that maintain in-



(a) Percentage of rate limited cases (b) Throughput Loss (c) Update Time

Figure 16: Dionysus only occasionally runs into deadlocks and uses rate limiting, and experiences little throughput loss. It also consistently outperforms SWAN in update time.

variants specified by the operator. Mahajan and Wattenhofer [13] present an efficient solution for maintaining loop freedom. As mentioned earlier, unlike these works, the key characteristic of our approach is dynamic scheduling, which leads to faster updates.

Mahajan and Wattenhofer [13] also analyze the nature of dependencies among switches induced by different consistency properties and outline a general architecture for consistent updates. We build on their work by developing a concrete system.

Some works develop approaches that spread traffic such that the network stays congestion-free after a class of common failures [33, 34], and thus no network-wide updates are needed to react to these failures. These approaches are complementary to our work. They help reduce the number of network updates needed. But network updates are still be needed to adjust to changing traffic demands and reacting to failures that are not handled by these approaches. Dionysus ensures that these updates will be fast and consistent.

11. CONCLUSION

Dionysus enables fast, consistent network updates in SDNs. The key to its speed is dynamic scheduling of updates at individual switches based on runtime differences in their update speeds. We showed using testbed experiments and data-driven simulations that Dionysus improves the median network update speed by 53%-88% over static scheduling. These faster updates translates to a more nimble network that reacts faster to events like failures and changes in traffic demand.

Acknowledgements We thank Srinivas Narayana, Meg Walraed-Sullivan, our shepherd Brighten Godfrey, and the anonymous SIGCOMM reviewers for their feedback on earlier versions of this paper. Xin Jin and Jennifer Rexford were partially supported by NSF grant TC-1111520 and DARPA grant MRC-007692-001.

12. REFERENCES

[1] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe, "Design and implementation of a routing control platform," in *USENIX NSDI*, 2005.

[2] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4D approach to network control and management," *SIGCOMM CCR*, 2005.

[3] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker, "Rethinking enterprise network control," *IEEE/ACM Trans. Networking*, vol. 17, no. 4, 2009.

[4] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "ElasticTree: Saving energy in data center networks.," in *USENIX NSDI*, 2010.

[5] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks.," in *USENIX NSDI*, 2010.

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

[7] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," in *ACM SIGCOMM*, 2011.

[8] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, *et al.*, "B4: Experience with a globally-deployed software defined WAN," in *ACM SIGCOMM*, 2013.

[9] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *ACM SIGCOMM*, 2013.

[10] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. A. Maltz, "zUpdate: Updating data center networks with zero loss," in *ACM SIGCOMM*, 2013.

[11] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *ACM SIGCOMM*, 2012.

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

[13] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *ACM SIGCOMM HotNets Workshop*, 2013.

[14] C. E. Leiserson, R. L. Rivest, C. Stein, and T. H. Cormen, *Introduction to Algorithms*. The MIT press, 2001.

[15] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An open framework for OpenFlow switch evaluation," in *Passive and Active Measurement Conference*, 2012.

[16] A. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Participatory networking: An API for application control of SDNs," in *ACM SIGCOMM*, 2013.

[17] Nicira, "Network virtualization for cloud data centers." `http://tinyurl.com/c9jbkuu`.

[18] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian, "Fabric: A retrospective on evolving SDN," in *ACM SIGCOMM HotSDN Workshop*, 2012.

[19] B. Raghavan, M. Casado, T. Koponen, S. Ratnasamy, A. Ghodsi, and S. Shenker, "Software-defined Internet architecture: Decoupling architecture from infrastructure," in *ACM SIGCOMM HotNets Workshop*, 2012.

[20] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, 1972.

[21] M. Newman, *Networks: An Introduction*. Oxford University Press, 2009.

[22] Arista, "Arista 7500 series technical specifications." `http://tinyurl.com/lene8sw`.

[23] P. Francois, O. Bonaventure, B. Decraene, and P.-A. Coste, "Avoiding disruptions during maintenance operations on BGP sessions," *Network and Service Management, IEEE Transactions on*, vol. 4, no. 3, 2007.

[24] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani, "Consensus routing: The Internet as a distributed system," in *USENIX NSDI*, 2008.

[25] N. Kushman, S. Kandula, D. Katabi, and B. M. Maggs, "R-BGP: Staying connected in a connected world," in *USENIX NSDI*, 2007.

[26] P. Francois, M. Shand, and O. Bonaventure, "Disruption free topology reconfiguration in OSPF networks," in *INFOCOM*, 2007.

[27] P. Francois and O. Bonaventure, “Avoiding transient loops during the convergence of link-state routing protocols,” *IEEE/ACM Trans. Networking*, vol. 15, no. 6, 2007.

[28] S. Raza, Y. Zhu, and C.-N. Chuah, “Graceful network state migrations,” *IEEE/ACM Trans. Networking*, vol. 19, no. 4, 2011.

[29] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure, “Lossless migrations of link-state IGPs,” *IEEE/ACM Trans. Networking*, vol. 20, no. 6, 2012.

[30] R. McGeer, “A safe, efficient update protocol for OpenFlow networks,” in *ACM SIGCOMM HotSDN Workshop*, 2012.

[31] S. Ghorbani and M. Caesar, “Walk the line: Consistent network updates with bandwidth guarantees,” in *ACM SIGCOMM HotSDN Workshop*, 2012.

[32] A. Noyes, T. Warszawski, and N. Foster, “Toward synthesis of network updates,” in *Workshop on Synthesis (SYNT)*, 2013.

[33] Y. Wang, H. Wang, A. Mahimkar, R. Alimi, Y. Zhang, L. Qiu, and Y. R. Yang, “R3:resilient routing reconfiguration,” in *ACM SIGCOMM*, 2010.

[34] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter, “Traffic engineering with forward fault correction,” in *ACM SIGCOMM*, 2014.

APPENDIX

A. PROOFS OF THEOREMS

Proof of Theorem 1: Given a network, where all flow demands from a set of sources to a set of destinations must go through either switch u or v. Each switch has a memory limit for k rules (flows), and each switch has a bandwidth capacity limit of c. We have $2k-1$ flows, one big flow with capacity $c/2$, $k-1$ flows with capacity ϵ (think of $\epsilon = 0$), plus a set S of $k-1$ flows, all with integer capacity, in total c. Currently, the set S goes through switch v, all other flows (the big one and the tiny ones) go through switch u. The target state is to swap the switches of all flows, i.e. the big and the tiny flows should go through switch v, the set S through switch u. Note that both current and target solution are feasible regarding both capacity and memory. Initially, we cannot move any flow from v to u, not even partially, because the rule limit on switch u is already maxed out. So we can only (partially) move a single flow from u to v. If we move the big flow, we need a new rule on switch v, which will also max out the rule limit on switch v, at which point we are stuck as both memory limits are maxed out. However, we can move an ϵ flow from u to v, first creating an additional rule at v, then moving the flow, then removing one rule at u. At this stage have used all rules on v, but we have one spare rule at u, which gives us the possibility to (partially) move a flow from v to u. Again, partially moving a flow is not a good idea as we are maxing out regarding rules on both switches. However, there is enough spare capacity on switch u to completely move one of the flows in S. We do that, as it is the only thing we can do. We continuing moving ϵ-flows from u to v and then S S-flows from v to u. However, since we cannot move flows partially, we always must move complete flows, and at some point, capacity on u will become a problem. In order to be able to move the big flow from u to v, we must have moved a subset S' of S from v to u such that this subset has exactly a total capacity $c/2$. In order to figure out the set S', we need to *partition* the flows into two equal-capacity sets. This is equivalent to the so-called partition problem, an NP-complete problem that must partition of set of n integers into two sets with the same sum.

Proof of Theorem 2: We use the same network as above, i.e. all flow demands from a set of sources to a set of destinations must go through either switch u or v. Each switch has a bandwidth capacity limit of c. We have k flows, one with capacity $c/2$, plus a set S of $k-1$ flows, all with integer capacity, in total c. The big flow initially goes through switch u, the set S through switch v. Again, as above, we want to swap all flows. If we could solve partition, we would in a first step move a set S', subset of S with total capacity of $c/2$ from v to u, then the big flow from u to v, and finally all the other flows ($S \setminus S'$) from v to u. All flows are properly moved and touched only once. If we cannot solve partition, at least one flow must first be split (some part of the flow going through switch u while the other part going through switch v). Eventually this flow is properly moved as well, but in addition to touching each flow once, we need to touch at least one flow at least twice, which costs time.

Algorithm 6 CanScheduleOperation(O_i) — WCMP Network

```
1: if !$O_i$.isChangeWeightOp() then
2:     return false
3: $canSchedule \leftarrow false$
   // Check link capacity resource
4: $total \leftarrow 0$
5: $O_{i_0} \leftarrow parents(O_i)[0]$
6: for path node $P_j \in parents(O_{i_0})$ do
7:     $P_j.available \leftarrow l_{ji_0}$
8:     for resource node $R_k \in parents(P_j)$ do
9:         $P_j.available \leftarrow min(available, l_{kj}, R_k.free)$
10:    $total \leftarrow total + P_j.available$
11: if $total > 0$ then
12:    $canSchedule \leftarrow true$
   // Check switch memory resource
13: for operation node $O_j \in parents(O_i)$ do
14:    $R_k \leftarrow resourceParent(O_j)$
15:    if $R_k \neq null$ && $R_k.free < l_{kj}$ then
16:        $canSchedule \leftarrow false$
17: if $canSchedule$ then
       // Update link capacity resource
18:    $O_{i_0} \leftarrow parents(O_i)[0]$
19:    for path node $P_j \in parents(O_{i_0})$ do
20:        for resource node $R_k \in parents(P_j)$ do
21:            $l_{kj} \leftarrow l_{kj} - P_j.available$
22:            $R_k.free \leftarrow R_k.free - P_j.available$
23:        for operation node $O_k \in children(P_j)$ do
24:            $l_{jk} \leftarrow l_{jk} - P_j.available$
       // Update switch memory resource
25:    for operation node $O_j \in parents(O_i)$ do
26:        $R_k \leftarrow resourceParent(O_j)$
27:        if $R_k \neq null$ then
28:            $R_k.free \leftarrow R_k.free - l_{kj}$
29: return $canSchedule$
```

B. SCHEDULING ALGORITHMS FOR WCMP NETWORK

Algorithm 6 and 7 show the pseudo code of $CanScheduleOperation(O_i)$ and $UpdateGraph(G)$ for WCMP networks.

CanScheduleOperation (Algorithm 6): Different from tunnel-based networks, WCMP networks don’t have tunnel add or delete operation. Instead, every hop have weights to split a flow among multiple next-hops. To update a flow, all switches of this flow have to be touched to implement a two-phase update. Therefore, this function checks on a per-flow basis by examining the change weight operation at the ingress switch for every flow, e.g., Y in Figure 2 (Lines 1, 2). Similar to tunnel-based networks, it gathers all free capacities on the paths where traffic increases (Lines 4-12). It iterates over path nodes and obtain the available capacity ($P_j.available$) of the path (Lines 6-10). This capacity limits the amount of traffic that we can move to this path. Note that these path nodes are the parents of O_i’s parents (e.g., parents of X rather

Algorithm 7 UpdateGraph(G) — WCMP Network

```
1:  for finished operation node $O_i \in G$ do
2:      if $O_i.isDelOldVerOp()$ then
3:          $R_j \leftarrow child(O_i)$
4:          $R_j.free \leftarrow R_j.free + l_{ij}$
5:      else if $O_i.isChangeWeightOp()$ then
6:          for path node $P_j \in children(O_i)$ do
7:              for resource node $R_k \in children(P_j)$ do
8:                  $l_{jk} \leftarrow l_{jk} - P_j.committed$
9:                  $R_k.free \leftarrow R_k.free + P_j.committed$
10:                 if $l_{jk} = 0$ then
11:                     Delete edge $P_j \rightarrow R_k$
12:             $P_j.committed \leftarrow 0$
13:             if $l_{ij} = 0$ then
14:                 Delete $P_j$ and its edges
15:         if $O_i.hasNoChildren()$ then
16:             Delete $O_i$, related nodes and edges
17: for resource node $R_i \in G$ do
18:     if $R_i.free \geq \sum_j l_{ij}$ then
19:         $R_i.free \leftarrow R_i.free - \sum_j l_{ij}$
20:         Delete all edges from $R_i$
```

than parents of Y in Figure 2) since O_i is the change weight operation at the ingress switch (e.g., Y in Figure 2). This flow can only be scheduled if there is any free capacity on these paths (Lines 11, 12). Then we check all the switches to see if they have free memory to accommodate the operations that add weights with new version, e.g., S2 in Figure 2 (Lines 13-16). If we have both link and switch resource, we can schedule update to this flow (Lines 17-28). We update link resource (Lines 18-24) and switch resource (Lines 25-28) accordingly.

The function finally returns $canSchedule$ denoting whether the flow can be scheduled. In the schedule part (Line 7 in Algorithm 2), different from tunnel-based networks, we do a two-phase update, where we first add weights with new version (e.g., X in Figure 2), change weights and assign new version at ingress switch (e.g., Y in Figure 2) then delete weights with old version (e.g., Z in Figure 2).

UpdateGraph (Algorithm 7): This function updates the dependency graph based on finished operations in the last round. We iterate over all finished operations (Lines 1-15). If the operation deletes weights with old version, we free rule space (Lines 2-4). If the operation changes weights with new version, for each child path node, we release resources to links on the path (Lines 8, 9) and delete the edge if all resources are released (Lines 10, 11). We reset $P_j.committed$ to 0 (Line 12) and delete it if all traffic to be moved has been moved (Lines 13-14). After this, we check whether O_i has any children left. If so, we keep these nodes in order to move the remaining traffic. Otherwise, it means all traffic has been moved, and we delete O_i and all the related two-phase commit nodes and edges (e.g., X, Y, Z , the related path nodes and edges in Figure 2). Finally, we iterate over all resource nodes and remove edges from unbottlenecked resources (Lines 17-20).