Max Progressive Network Update

Yang Chen and Jie Wu
Temple University, USA
Email: {yang.chen, jiewu}@temple.edu

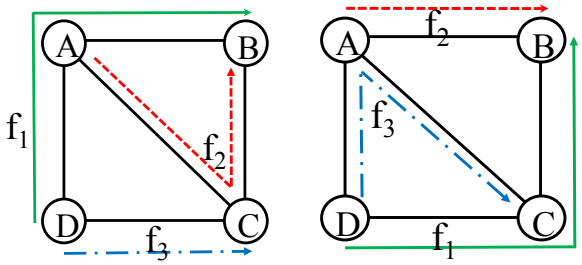
***Abstract*—Datacenter network must constantly be updated to adapt to frequent traffic changes and achieve high network utilization. However, the quality of service can be easily reduced by unexpected transient link congestion, which may be caused by variable link transmission latency during flow migration. Furthermore, when network administrators reallocate bandwidth resource without taking flow path information into consideration, deadlocks among flows waiting for link resources unavoidably block the update process. In this paper, we prove finding a feasible update plan with the minimum number of rate-limiting flows when deadlocks occur is NP-hard. We propose a buffer-assisted schedule, Max Progressive Updating Method (MAPUM), to update the network in a consistent and efficient way. Extensive simulations prove that MAPUM accomplishes network update effectively and without loss.**

***Index Terms*—Network updates, SDN, consistent, congestion.**

I. Introduction

Temporary network update disruptions may occur with many unexpected events like packet loss or blackholes. The reasons for disruptions include straggling of switch rule update, controller disconnection, and imperfection in clock synchronization. To prevent anomalies, researchers pay extensive attention to the network update problem in SDNs. There are three key challenges for network update issues: *optimality, consistency*, and *swiftness* [1]. An optimal migration requires that the final routing configuration after the update be the same as the given target state, which is derived by solving the multi-commodity flow problem [2]. In SDNs, a centralized controller with a global view can guarantee optimality by directly establishing the targeted network configuration. A consistent migration of flows satisfy the properties of congestion-free, loop freedom and no temporary demand reduction. A swift update prevents a new routing setup from becoming obsolete due to frequently changing network conditions. Swiftness is more critical in data centers, where network update serves as usual maintenance.

Current update schedules suffer from different downsides. We illustrate the unfeasibility of current methods in Fig. 1. There are three flows f_1, f_2, and f_3, whose capacities are 0.7 unit, 0.8 unit, and 1 unit, respectively. Each link has a capacity of 1 unit. We need to migrate the flows' paths from the initial state in 1a to the final state in 1b. Only when a flow's final path has enough available bandwidth, can it be consistently migrated. Flows are unsplittable. However, it is obvious that none of these flows can be updated to their final paths because other flows' initial paths are occupying their resources. When flows' initial paths occupy other flows' final state paths, none of them can be updated; we call this a deadlock. Flows f_1, f_2 and f_3 form a deadlock. Current works either neglect handling this situation or randomly stub out flows to vacate competing links. For example, SWAN [3] can only accomplish an update when leisure capacity exists along all links of the flow's path. In this example, all the links with flows going through them are saturated and have no free bandwidth. SWAN will come up with no solution. Dionysus [4] generates a dependency graph and finds all SCCs (strong connected components). When deadlocks exist, it tries to randomly rate limit a few flows until all deadlocks break. This solution causes packet loss, which reduces the quality of service. Additionally, the link capacity reservation method [3] has so little flexibility that bandwidth utilization is always low. The intermediate state involvement method [5] increases the number of update steps to avoid link congestion due to flow mixture in different network states.



(a) Initial state (b) Final state

Fig. 1: A network update example.

In this paper, we focus primarily on how to generate a consistent update plan that breaks all of the deadlocks with the help of the switch buffer, a resource ignored in previous research. We present a heuristic algorithm, Max Progressive Updating Method (MAPUM), to address this problem. With the help of an adequate amount of buffer, we can break the deadlocks where interwoven flows compete for bandwidth and update the network in a lossless manner. Deliberately deploying buffers in switches further shortens the time that it takes to migrate the flows. We also investigate the complexity of the deadlock-breaking problems mentioned above, and we demonstrate that they are NP-hard.

The main contributions of our work are in the following:

1) We summarize current network update methods and analyze their advantages and disadvantages.
2) We propose a buffer-assisted strategy in order to update network in a consistent way, even when there are

complex deadlocks. We also prove that it is NP-hard to allocate buffers' sizes and locations to resolve all deadlocks with a minimum effect on the network. Then we introduce a heuristic solution to efficiently arrange buffers in the switches.

3) We demonstrate the significant advantages of our approach compared to current works in simulations.

The remainder of this paper is organized as follows. Section II surveys related works. Section III describes the model and the problem formulation. Section IV talks about generating a resource dependency graph. Section V analyzes the problem and proposes our MAPUM solution. Section VI includes the experiments. Finally, Section VII concludes the paper.

II. Related Work

When it comes to the routing reconfiguration problem, there are two basic mainstream methods: ordering [6, 7] and two-phase [8–10]. The ordering method updates the forwarding table in the switches one-by-one in a specified order. The order is carefully calculated to preserve certain required properties, for example, loop-free, or blackhole-free. However, this order might not guarantee both forwarding and policy demand. The two-phase update scheme installs both the initial and final rules on all switches, and it tags packets to signal which rule should be applied. This method ensures the success of an update, but it doubles the number of rules on every switch, wasting expensive and power-hungry ternary content-addressable memory (TCAM) resource. In this paper, we perform the two-phase commit using version numbers for update rules to maintain packet coherence.

Due to constantly changing traffic demands, data center network updates occur frequently whether triggered by the operators, applications, or sometimes, even by failures. The inherent asynchrony will lead to over-utilization of links, inducing congestion and packet loss. One might just ignore these effects, hoping things will get better on their own, but delays and loss of data are not desirable, especially in real-time applications . Therefore, recent developments have considered consistent migration, i.e., congestion-free and without temporary demand reduction [1, 3, 5, 11]. One paper [12] does not strictly require that the update process be lossless. Rather, it aims to obtain time-efficiency and minimize the loss of packets.

We can roughly classify update strategies into three categories: link reservation [3], intermediate state involvement [5], and time-awareness [1, 13–15]. The approach of SWAN [3] has two parts: First, if a fraction s of capacity (*slack*) is guaranteed to be free on each link for both the old and new flow, the network can be updated in $\lceil 1/s \rceil - 1$ steps. Second, in order to solve the problem optimally, SWAN uses linear programming to check if a solution with x steps exists. However, when there is no slack on some edges, the algorithm is unlikely to halt in certain steps with a high computation complexity. ZUpdate [5] attempts to compute and execute a sequence of steps to progressively meet the end requirements from an initial traffic matrix and a traffic distribution. To safely migrate flows, these representative update scenarios need to be guaranteed that there will be no congestion. However, applying intermediate states will stretch the update time, and the traffic chaos caused by the migration will last longer. Time-awareness consistent update strategy [13] utilizes time-triggered network updates to achieve consistency by either the ordering or the two-phase commit method. This scheme asks too much of time synchronization. With one straggling switch, the whole process can create serious congestion or other problems.

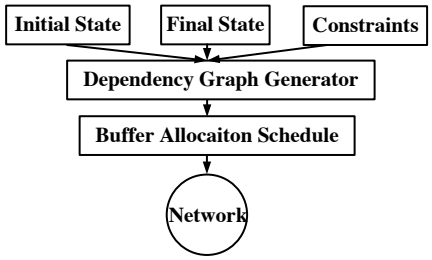


Fig. 2: Scheduling process.

III. Model and Formulation

Our scheduling process is shown in Fig. 2. From the network administrator, we learn the initial and final network states, the consistency and what constraints must be preserved. Assume the initial and final states are both valid and the flows are unsplittable. Consistency requires that the update be loop-free, drop-free, and congestion-free. We need to find a feasible and efficient solution to consistently migrate flows with the help of switch buffer, even when the network has a lot of deadlocks among flows in different states. Deadlocks are detected using a dependency graph generator. We propose an efficient approach, MAPUM, which focuses on handling deadlocks with the help of the proper allocation of buffer location and size.

A. A Motivating Example and Some Insights

Consider the example in Fig. 1. As discussed above, a consistent update plan does not exist. But with the proper allocation of buffer and careful selection of the flows that will be temporarily buffered in switches, we can update the network to its final state consistently. Now let's think of the most trivial plan. In this paper, we assume that migration is flow-based, which means that a flow is allowed to update only when the links along its path are all available. The flows can only be rate limited in their source switches' buffer, and the rate of the flows will be reduced to 0. We list all the feasible update plans in Table. I. We measure time in slots and buffer size in units. Take rate-limiting flow f_1 as an example. At time slot 0, we rate limit f_1. After 2 slots, f_2 can migrate because of the available bandwidth of AB. At slot 4, f_3 can move to its final path. At slot 5, f_1 migrates and the update is finished. The buffer size is 5. There are, in total, five methods to consistently break all the deadlocks. The minimum number of rate-limiting flows is only one; the minimum time is 4 and the minimum buffer size is 5, which demonstrates that

TABLE I: Rate-limiting plan

Combination of flows	f_1	$f_1$and f_2	$f_1$and f_3	$f_2$and f_3	f_1,$f_2$and f_3
# rate-limiting flows	1	2	2	2	3
Update time	5	4	5	7	3
Buffer size	3.5	4.4	7.5	10	5.3

different scheduling plans matter. Intuitively, we should buffer flows with an initial path that includes many congested link resources, and a final path that consists of few bottleneck links. It is also better to limit flows that are engaged in a higher number of deadlocks, because it is more beneficial for resolving all deadlocks. Furthermore, we can utilize buffer to accelerate the update. Consider the following situation. If flow f_1's initial path occupies f_2's final path resources, f_2 occupies f_3's,..., and f_{n-1} occupies f_n's. Therefore, we need to first update f_1, then f_2,..., and finally, f_n. This process means that the update will take a long time. With the help of buffer, we can additionally select a flow in the middle to be buffered so that two parts of the update process can be done concurrently, significantly shortening the time needed. With more parallel processes like these, the time can be reduced further. In an extreme case, if we buffer every flow, the network can be updated in the shortest time.

B. Network Model

The network G consists of a set of switches S and a set of directed links L. A flow f is from a source switch s_i to a destination switch s_j with traffic volume t_f. Links along the path of f are defined as $L_f = \{l_{i,j} | l_{i,j} \in p_f\}$, where p_f is the forwarding path of flow f. The network state N is the combined state of all flows, i.e., $N = \{f | f \in F\}$. The network state includes each flow's path and its allocated bandwidth. We use two-phase commit to update each switch's routing table information. At every hop, the switch checks the packet's header to decide its outgoing port. The controller needs to manage the time to tag the packets with the new version number.

Network, graph, capacity: We define a network as a simple directed graph with edge capacities.

***Definition** 1:* Let $G = (V, E)$ be a simple connected directed graph with $n = |V|$ nodes representing the switches, and $m = |E|$ edges representing the links. We denote the set of outgoing edges (u, v) of a node $v \in V$ by $out(v)$ and the set of incoming edges (u, v) by $in(v)$. A network N is a pair (G, c), with $c : E \to \mathbb{R}^+$ being a function assigning each edge $e \in E$ a capacity of $c(e)$.

Flow: Next, we define an unsplittable flow according to the buffer-involved flow constraints, i.e., demand satisfaction, flow conservation, and capacity constraints. As is common in this context, we only consider self cycle-free flows in this paper.

***Definition** 2:* Let $N = (G, c)$ be a network. A map $f : E \to \mathbb{R}^+$ is called an unsplittable flow from s to t if it is cycle-free and satisfies the buffer-involved flow constraints, i.

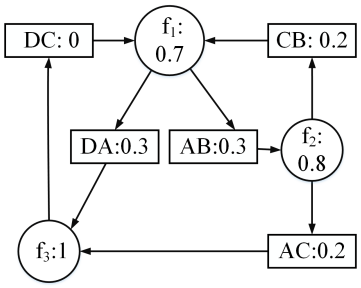


Fig. 3: Resource dependency graph.

e., $\forall v \in V \backslash \{s, t\}$:

$$f(e)_{e \in in(v)} = f(e)_{e \in out(v)} + buffer(v, f), \quad (1)$$

$$f(e)_{out} = d_f = f(e)_{in}, \quad (2)$$

$$\forall e \in E : \sum_f f(e) \leq c(e) \quad (3)$$

with d_f being the size of f and the edges with $f(e) > 0$ forming the path of f from s to t. All these unsplittable flows form the set $F = \{f_i, i = 1, 2, ..., m\}$ if there are m flows.

Initial and final network state: In this paper, we only consider network updates for flows, i.e., given a set of old forwarding rules for some flow f, we want to change it into a set of new forwarding rules for another flow f'.

***Definition** 3:* Let N be a network, and let F and F' be flows in N. A network update is a triple (N, F, F').

Atomicity of network updates: We assume that the change from a flow f to a flow f', both from s to t, is performed as an atomic operation on the source router s. In practice, this can be achieved by a two-phase protocol [9]. All forwarding rules for F' are installed by the SDN controller first. When these installations are confirmed, the ingress router s will start tagging all packets from F to F'. Note that with this method, if the latency from s to some node in the network is the same for F and F', the flow F' will arrive after F has already departed.

IV. Dependency Graph Generation

The primary challenge is to tractably explore relationships among flows and resources in the initial and final states. In this paper, we leverage a dependency graph to describe the update relationships. As shown in Fig. 2, the dependency graph generator takes the initial state NS_i, the final state NS_f, and the required constraints as input. There are two types of nodes in our dependency graph: link nodes and flow nodes. Link nodes represent link resources and are labelled with the amount of current residue capacity. Flow nodes correspond to different flows marked with demands. An edge from a link node to a flow node means that the flow needs this link resource to update, and the weight of the edge shows the delay after the resource becomes available. The link transmission delay is taken into consideration.

To simplify, we measure the delay in the same units as the update step. $late_{l_j, f_i} = m$ if the link l_j is the m_{th} link that flow f_i passes in the final state. $late_{f_i, l_j} = m$ if the link l_j is the m_{th} link that flow f_i passes in the final state.

Algorithm 1 Dependency Graph Generator

Input: The initial and final network State N_{ini} and N_{fin}, each flow i's demand dem_{f_i}, and each link's capacity $c_j \in C$;

Output: Dependency Graph G;

1: **for** each link l_j in the network **do**
2: add a new node with its residue capacity $res_{l_j} = c_{l_j} - \sum dem_{f_i}$ in the initial state N_{ini}, , p_{f_i} passes link l_j;
3: **for** each flow $f_i \in F$ **do**
4: add a new flow node with its demand dem_{f_i};
5: **for** each link along the initial (and final) path of flow f_i **do**
6: add a directed edge from the flow node to the link node (in the final path, the direction is reversed);
7: label the edge with the transmission latency $late_{f_i,l_j}$ (or $late_{l_j,f_i}$ in final st) in units of step;
8: **return** Dependency Graph G;

Algorithm 1 shows the details of generating a dependency graph. Take Fig. 3, the dependency graph of Fig. 1, as an example. There are three deadlocks in the dependency graph, shown in Fig. 4. We can clearly see that f_1 involves all three deadlocks, which means that rate limiting f_1 will make the update feasible. Therefore, the minimum number of flows to be rate limited is 1. However, this does not demonstrate that this method is the fastest plan or that it needs the least amount of buffer.

V. Consistent Updating Strategy with Buffer Assistance

A. The Hardness of the Buffer-assisted Schedule

To ensure a consistent update plan, we introduce buffer to strategically break the deadlocks. Buffer scheduling is a resource allocation problem: how can we allocate fewer buffer resources to accomplish a consistent network update.

***Theorem** 1:* Without limitations on buffer size or location, a consistent network update can be accomplished in one step.

Proof: First, we block all the flows and wait for a sufficiently long period of time. When there is no residue traffic in the network, we can migrate flows to their new paths with adequate bandwidths. Because the initial and final states are valid, there will not be any inconsistencies like loops, blackholes, or congestions. □

Buffer is expensive and has limited size. The above situation is an extreme condition. it is possible to strategically migrate flows in a specific order, or to limit a few of them to achieve a consistent update. As a result, we are required to choose only certain flows to be rate-limited from all the flows that need to be migrated. We also decide where to begin buffering and what percentage of the flow to buffer to meet the Quality of Service (QoS) requirement. After making a scheduling decision, we must continue selecting flows until all the deadlocks have been resolved. This is a complex resource allocation problem. We can prove the complexity of handling deadlocks in network update with the help of buffer.

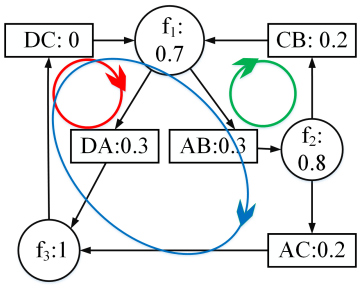


Fig. 4: Deadlocks in the dependency graph.

***Theorem** 2:* Even in the absence of partial flow limiting, finding a feasible update schedule with the minimum number of rate-limited flows with deadlocks is NP-hard.

Proof: Given a network's initial and final states, there exists several deadlocks among links in a dependency graph. Each deadlock is a cycle consisting of multiple flows. Each flow can be involved in several deadlocks because it must wait for multiple link resources. Suppose there are n deadlocks in the graph G and the general collection of deadlocks is U. In total, m flows become involved in G and the i^{th} flow ($i = 1, ..., m$) is able to break the set $S_i \subseteq U$. The goal is to break all the deadlocks by cutting off the fewest number of flows using buffer. So we reduce the original problem to the so-called set cover problem, an NP-hard problem that covers all elements using the fewest number of sets.

B. Rate-Limiting Flow Selection

Because of the NP-complete nature of the problem, we propose a heuristic algorithm to find a consistent update plan that selectively limits flows as seldom as possible.

1) Flow Priority: In order to limit fewer flows, we are required to carefully sort the flows engaged in deadlocks and to pick up "more beneficial" flows to be buffered first. From the example above, we can draw a conclusion that the priority of a flow is related to the out-degree and in-degree of the simplified dependency graph, the deadlocks it involves, and the whole latency of its engaged deadlocks. We formulate a flow's priority in Eq. 4 to evaluate the importance of breaking deadlocks and quickly update.

$$pri_{f_i} = \frac{d_(out)}{d(in)} * \max\ delay(cycle_k),\ k = 1, ..., cyc_{f_i} \quad (4)$$

cyc_{f_i} is the number of deadlocks where flow i is involved in the dependency graph. In the example's resource dependency graph, we can see that f_2's in-degree and out-degree difference is the largest. However, f_1 appears in more deadlocks, and the sum of its deadlocks' delays is biggest. Using Eq. 4, we sort the importance of these flows , from most important to least, as $f_1,\ f_2,\ and\ f_3$. Table I verifies the efficiency of our formulation.

2) MAPUM Scheduling: The specific update process of deadlocks is exhibited in Alg. 2, where time is measured in units of steps. After constructing the dependency graph G

Algorithm 2 MAPUM

Input: The dependency graph G and the deadlock set L;
Output: Updating plan of flow and buffer allocation;

```
for every new time step do
    In order, update every flow in the next link of its initial and final state;
    if there is still a deadlock in L then
        Select the current highest priority flow f_{highest-pri};
        if f_{highest-pri} can break down any deadlock in L then
            Rate-limit f_{highest-pri} and add buffer at the first blocked link's import switch along its path;
            Delete all deadlocks that limiting the flow f_{highest-pri} can solve in the deadlock set L;
    else
        The update arrangement is finished;
        Break out of the for-circulation;
return ;
```

using the above strategy, we use Dionysus [4] to consistently and efficiently update the flows that are not involved in any deadlock. We simplify the dependency graph G by deleting the non-blocked flows and their occupied resources. Next, [16] is utilized to find all the elementary circuits in the current G. Each flow's priority is calculated by Eq. 4 and arranged in a descending order. In every time step, we rate limit the highest priority flow and wait for $T_{interval}$ to make sure that no chaotic situations arise. We repeat the process until there is no deadlock in the G.

It takes some time to transfer a packet along a link; this is known as the link latency. By using the time step to reserve enough interval, we can efficiently avoid accidental congestions. Moreover, due to the difference of each link's latency and the impossibility of time synchronization, it is difficult for the controller to decide the exact update time to send messages to update switches. It also takes different amounts of time for switches to enforce the update plans, or for the same switch to handle different kinds of modifications (such as rule table insertion or deletion). We reserve enough time in each time step to make sure that the interval is long enough for even the slowest switch to accomplish the controller's update demand. The interval can be estimated as follows:

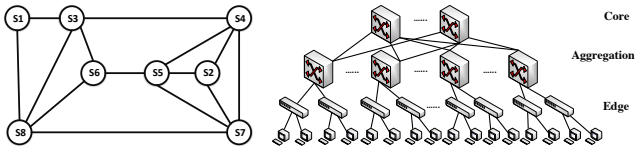
$$T_{interval} = \max\{\tau(s_i), i = 1, 2, ..., n\} \quad (5)$$

$$\tau(s_i) = \max\{t_i(\gamma), \gamma \in \Gamma\} \quad (6)$$

Γ is the general set of all kinds of update demands from the controller. We yield to the longest time to guarantee the congestion-free property.

C. Schedule Improvemet

We propose a heuristic algorithm to solve the consistent network update problem. We also find some improvements to our algorithm that help achieve a better performance. Every link has its transmission latency, so it takes some time for the flow to pass the link. As a result, we can begin to migrate the flow from its source switch when the flow can obtain every link resource. This saves the update time, but will have a more complicated scheduling strategy. Furthermore, if the update is slow and some buffer is released, reusing the free buffer is a flexible process that improves the utilization of switch buffer. We also observe that if a deadlock consists of many flows and has a large update delay, it is necessary to rate limit more than one flow. The divided parts of the deadlock can update in parallel, which saves significant time. The selection of rate-limiting flows should be balanced to achieve the min max update time of each branch.



(a) The WAN topology (b) The fat-tree topology

Fig. 5: Topology.

VI. Evaluation

Simulated experiments are conducted to evaluate the performances of the proposed algorithms. After presenting the network and flow settings, the results are shown from different perspectives to provide insightful conclusions.

A. Network and Flow Setting

We do simulations in two realistic topologies. The first is Microsoft's inter-data center WAN topology [4], consisting of 8 switches connected as shown in Fig. 5a. Each link is two-way and has a capacity of 10-Gbps. The second is a fat-tree topology [17] for the data center network scenario shown in Fig. 5b. There are 10 core switches, 20 aggregation switches, and 40 edge switches in this network. Each edge switch connects 2 hosts. Each switch has 4 10-Gbps ports, resulting in a full bisection bandwidth network. There are several flows generated randomly with different capacities, and we change flow numbers to simulate traffic variations. We assume the initial and final states of the network and flows are all valid, meaning that the path has no loop and the load on every link is within the capacity.

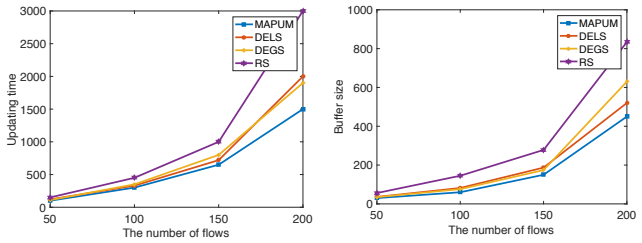
B. Benchmark Schemes

We compare our MAPUM with three schemes: RS (random flow update order), DELS (delay-consideration), DEGS (degree-consideration).

C. Performance

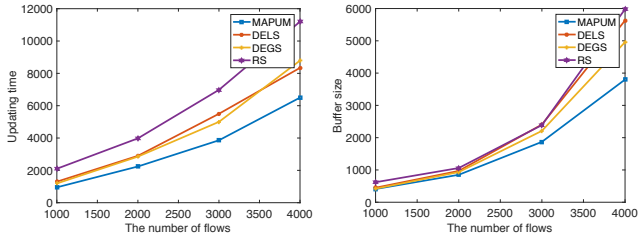
We study the update time and buffer size needed to finish the flow migration using four methods.

WAN: Fig. 6a shows the update time under different numbers of flows. As more flows are added, the update time increases more quickly. When the traffic load is heavy, the



(a) The WAN topology (b) The fat-tree topology

Fig. 6: WAN simulation results.



(a) The WAN topology (b) The fat-tree topology

Fig. 7: Data center simulation results.

network is likely to have more deadlocks. Then, it becomes harder to solve deadlocks, and the advantage of our scheme is more obvious (shown as the purple line in the picture). DELS's and DEGS's performances are roughly the same, but are not as good as MAPUM's, because they only consider a part of the elements influencing the flow deadlocks. RS is used as a baseline. Fig. 6b shows the buffer size needed to accomplish the update process. In order to migrate flows in a lossless way, we store the upcoming flows in the switch buffer to vacate their rooms to other flows of the final state. However, buffer is expensive and rare, so we would like to use it as seldom as possible. MAPUM not only achieves the best performance in update time, but also uses the least buffer.

Fat-tree: We also do large-scale simulations in a fat-tree data center topology. Though there are more switches, deadlocks are less likely to occur under this topology. The flow paths are configured neatly, so the traffic load balances much better than in the WAN. Resource in the data centers is expensive. Leveraging the network in a high-utilization way is highly recommended. Even a small congestion that lasts for only a short time will lead to a great loss, and it should be avoided to the best of the network administrators' abilities. Figs. 7a and 7b show that, of the four methods, MAPUM performs the best. It migrates flows with the shortest update time and the least buffer. The difference shows more clearly when the traffic load is heavier.

VII. Conclusion

We propose MAPUM, a method that can achieve the consistent and swift network update. We tactfully select several switches to limit bypass flows' rates with careful allocation of properly sized buffer. We prove that the buffer deployment problem is NP-hard. Furthermore, we propose a heuristic greedy algorithm to efficiently choose rate-limiting flows to accomplish different system constraints. We also make some improvements to our algorithm. Our simulation results evaluate the correctness, feasibility, and effectiveness of our approach.

VIII. Acknowledgment

This research was supported in part by NSF grants CNS1629746, CNS 1564128, CNS 1449860, CNS 1461932, CNS 1460971, CNS 1439672, CNS 1301774, and ECCS 1231461.

References

[1] N. W. S. Tseng, C. Lim and A. Tang, "Time-aware congestion-free routing reconfiguration," in *The IFIP Networking 2016 Conference (NETWORKING 2016)*.

[2] L. Fratta, M. Gerla, and L. Kleinrock, "The flow deviation method: An approach to store-and-forward communication network design," *Networks 1980*.

[3] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," ser. SIGCOMM '13.

[4] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," ser. SIGCOMM 2014.

[5] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zupdate: Updating data center networks with zero loss," *SIGCOMM 2013 Comput. Commun. Rev.*

[6] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure, "Seamless network-wide igp migrations," ser. SIGCOMM '11.

[7] J. McClurg, H. Hojjat, P. Černý, and N. Foster, "Efficient synthesis of network updates," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2015.

[8] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent updates for software-defined networks: Change you can believe in!" in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, ser. HotNets-X 2011.

[9] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," ser. SIGCOMM 2012.

[10] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," ser. HotSDN '13.

[11] R. W. S. Brandt, K. Foerster, "On consistent migration of flows in sdns," in *IEEE International Conference on Computer Communications (INFOCOM 2016)*.

[12] J. Zheng, H. Xu, G. Chen, and H. Dai, "Minimizing transient congestion during network update in data centers," in *2015 IEEE 23rd International Conference on Network Protocols (ICNP)*.

[13] T. Mizrahi, E. Saat, and Y. Moses, "Timed consistent network updates in software-defined networks," *IEEE/ACM Transactions on Networking*, 2016.

[14] T. Mizrahi, O. Rottenstreich, and Y. Moses, "Timeflip: Scheduling network updates with timestamp-based tcam ranges," in *2015 IEEE Conference on Computer Communications (INFOCOM)*.

[15] J. Zheng, H. Xu, G. Chen, H. Dai, and J. Wu, "Congestion-minimizing network update data centers."

[16] D. B. Johnson, "Finding all the elementary circuits of a directed graph," *SIAM 2006 Journal on Computing*.

[17] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *SIGCOMM 2008 Comput. Commun. Rev.*