# Scheduling Multi-flow Network Updates in Software-Defined NFV Systems

Yujie Liu*, Yong Li*, Marco Canini†, Yue Wang*, Jian Yuan*
* Tsinghua National Laboratory for Information Science and Technology
Department of Electronic Engineering, Tsinghua University, Beijing 100084, China
† ICTEAM, Université catholique de Louvain, Belgium

***Abstract*—Combining Network Functions Virtualization (NFV) with Software-Defined Networking (SDN) is an emerging solution to provide fine-grained control over scalable and elastic packet processing functions. Due to changes in network policy, traffic characteristics, or physical topology in Software-Defined NFV (SDNFV) systems, the controller needs to carry out network updates frequently, *i.e.*, change the data plane configuration from one state to another. In order to adapt to a newly desired network state quickly, the network update process is expected to be completed in the shortest time possible. However, the update scheduling schemes need to address resource constraints including flow table sizes, CPU capacities of Virtualized Network Functions (VNFs) and link bandwidths, which are closely coupled. Thus, the problem is difficult to solve, especially when multiple flows are involved in the network update. In this work we investigate the multi-flow update problem in SDNFV systems, and formulate it as a mixed integer programming problem, which is NP-complete. We propose an approximation algorithm via linear relaxation. By extensive simulations, we demonstrate that our algorithm approaches the optimal solution, while requiring 10x-100x less computing time.**

## I. Introduction

Network functions (NFs), or middleboxes, are widely deployed in enterprise, carrier and data center networks, and play an important role in enforcing security, boosting performance, and enabling novel functionality. Recently, Network Function Virtualization (NFV) [1] has gained increasing interest in academia and industry as it enables dynamic middlebox deployment in Virtual Machines (VMs) running on generic hardware rather than in expensive and dedicated devices. As such, NFV brings the advantages of reducing capital and operational expenses and improving the efficiency and flexibility of implementing service chains. In this context, Software-Defined Networking (SDN) provides a flexible mechanism to steer network traffic through a desired set of NFs. A growing number of studies [4]–[6] focus on combining NFV with SDN to achieve scalable, elastic, and on-demand network control.

In Software-Defined NFV (SDNFV) systems, a network update is defined as changing the network forwarding state from an initial configuration to the expected configuration. Network updates need to be implemented frequently for various reasons. When changes occur in traffic distribution or network topology, it is desired to dynamically adjust the network forwarding policy to meet traffic demands, perform planned maintenance and deal with network failures. Moreover, VNFs need to be migrated from one physical server to another to guarantee performance or enforce new policies. For example, when the CPU usage of an VNF (hereinafter referred to as "NF") is high, we need to offload a part of the traversing traffic to another NF to avoid degrading its processing performance.

However, considering the resource constraints, how to schedule network updates in SDNFV systems is still an open question. It is especially challenging when the forwarding states of multiple flows need to be updated simultaneously. This is because forwarding the traffic of a flow consumes three kinds of resources, which restrict the update process.

First, the *flow table sizes of SDN switches are limited*. Different from traditional networks, to perform pattern matching, the commonly used flow table is made of TCAM (Ternary Content Addressable Memory), which is expensive and power hungry. If during the update, a large amount of flow table entries are required to be added to a switch in excess to its TCAM capacity, the switch will refuse to install more rules or even drop them silently, resulting in an incorrect network configuration. Thus, from the perspective of switches, the flow table constraint should be considered.

Second, the *CPU resource allocated to an NF is limited*. If the CPU of an NF is highly utilized, its processing latency can increase, degrading the network performance. We should orchestrate the update of the flows, and restrict the traffic processed by NFs to avoid overloading the CPU.

Third, the *link bandwidth is another limited resource*. If a large flow is moved to a busy link during the update, the link's utilization could get significantly higher than that in the initial state. In order to guarantee network performance, congestion-freedom should be ensured during the update.

Making the problem even more challenging, we observe that these three constraints are coupled. Updating the flows' forwarding path will bring changes in the utilization of these three kinds of resources.

Network update schemes should not only satisfy these three constraints, but also be finished as quickly as possible. On one hand, the network needs to adjust to the new state quickly to adapt to changes in topology or traffic. On the other hand, network updates introduce additional overhead in terms of flow table memory [2] or controller's memory and communication bandwidth. Minimizing the update time can efficiently reduce the overhead.

Recent works have focused on network updates in SDN [11]–[18]. However, none of these prior works has considered

NF deployment in the system models, and did not analyze how NFs' CPU resources impact on the network update process. A growing number of studies [3], [19] investigated the problem of NF migration, and focused on preserving consistency of NF state during the migration. We study the problem from a larger perspective by considering NF migration problem in the scenarios where multiple flows' forwarding states are affected at the same time and analyzing how the network resources impact on the update process. These NF state update mechanisms [3], [19] can be incorporated into our scheme to implement NF state migration and ensure consistency.

In this paper, we investigate the problem of multi-flow network update in SDNFV systems. We make the following contributions:

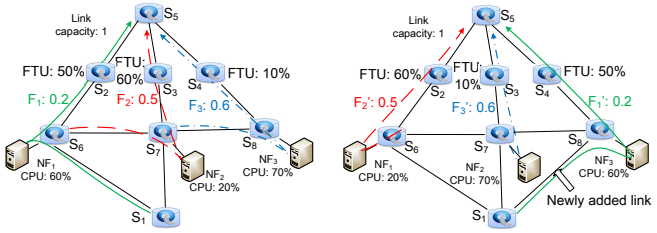
- We study the multi-flow update problem in SDNFV systems with a multi-step strategy, which to the best of our knowledge, is the first study of this problem. Aiming to minimize the update time, we formulate it as a mixed integer programming (MIP) problem, which is NP-complete.
- We propose a linear programming based algorithm via linear approximation to solve the multi-flow update problem in polynomial time.
- By extensive simulations with real traffic traces, we demonstrate that the proposed algorithm performs close to the optimal solution while completing with only 10x-100x less computing time.

## II. Overview

We consider an SDNFV system consisting of a logically centralized controller, SDN switches and NFs running on commodity servers of an NFV platform. SDN switches forward packets and their flow tables are configured by the SDN controller via an SDN southbound protocol such as OpenFlow. NFs run on generic compute resources via virtualization technology. Through interactions with SDN switches and NFs, we posit that the SDNFV controller (hereinafter referred to as "the controller") obtains a global view of the system. In this context, we aim to propose a network update scheduling mechanism that runs at the controller. When a network update is required, the controller computes an update solution. Then, to implement the update solution, it needs to reconfigure the forwarding path of the flows, which include updating not only the switches' flow tables but also the related flow state of the NFs.

We start by illustrating an example to introduce the network update problem in SDNFV systems, and then discuss the challenges of the problem that we need to address.

**Example.** We now illustrate the details of the network update process with an example shown in Fig. 1. For simplicity, the controller is not shown. Fig. 1 (a) depicts the initial network state. There are three active flows destined to $S_5$: $F_1$, $F_2$, and $F_3$, which originate from $S_1$, $S_6$, and $S_7$, respectively. Their traffic rates are $20\%$, $50\%$ and $60\%$ of link capacity, respectively. The flows traverse different types of NFs and consume $60\%$, $20\%$ and $70\%$ of CPU resources, respectively.




(a) Initial network state (b) Final network state

Fig. 1. A network update scenario caused by adding a new link.

We assume that $S_2$, $S_3$ and $S_4$ have the same flow table capacity, which is smaller than that of other switches. Each flow is a large traffic aggregate that occupies multiple flow table entries, and forwarding $F_1$, $F_2$ and $F_3$ brings $50\%$, $60\%$ and $10\%$ Flow Table Utilization (FTU) in $S_2$, $S_3$, and $S_4$ respectively.

When $S_1$ and $S_8$ are connected by a newly added link as shown in Fig. 1 (b), the controller decides to carry out a network update to reduce the average number of hops traversed by flows. Then, the paths of three flows $F_1$, $F_2$ and $F_3$ and their NFs, need to change from the initial network state shown in Fig. 1 (a) to the state shown in (b).

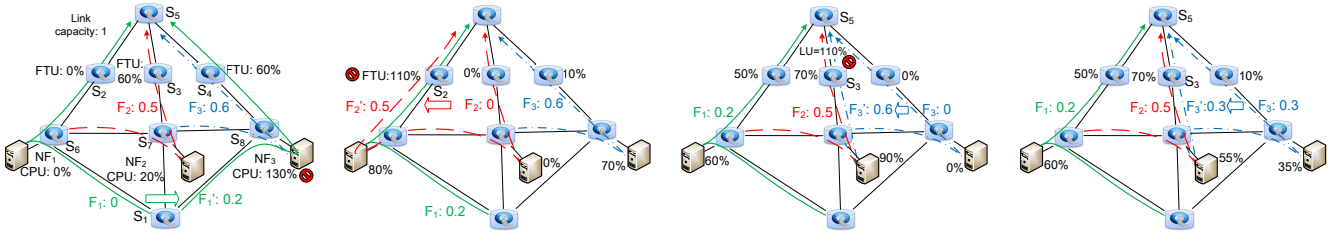
However, note that during the update certain links may experience higher utilization compared to their initial or final state. The same applies to CPU cycles at NFs and flow tables, which could be occupied more than what it is necessary in either state. For example, let us consider updating $F_1$, $F_2$, and $F_3$ simultaneously. Since it is inherently difficult to synchronize the update of multiple flows across devices, in case that $F_3$'s update is started earlier than $F_2$, the utilization of the link between $S_3$ and $S_5$ may exceed $100\%$ in the intermediate state.

In fact, it is generally impossible to complete the update of several flows in one step due to these resource constraints. Thus, we propose a multi-step strategy to solve the network update problem. However, addressing this problem is not straightforward as we face several challenges as follows.

**Challenge 1: How to finish network updates as quickly as possible?** To avoid service disruptions and adjust the network to the new intended state, we should complete the update process as quickly as possible. To speed up the update process, we expect to update several flows concurrently in each step. For example, if the network resources are doubled in Fig. 1, any update solution is feasible. Among these solutions, we prefer to update all flows in a single step, as it is the most efficient approach. Thus, in a multi-step strategy, we aim to reduce the total update time by minimizing the number of update steps.

**Challenge 2: How to meet resource constraints during network update?** The network update process is restricted by network resources. To analyze the constraints that need to be considered, we present four update plans in Fig. 2 that solve the problem shown in Fig. 1.

**CPU constraint:** NFs' CPU capacity is a constraint. The




(a) If $F_1$ is updated first, $\mathrm{NF}_3$'s CPU is overloaded.

(b) If $F_2$ is updated first, $S_2$'s flow table is over utilized.

(c) If $F_3$ is updated first, link congestion occurs.

(d) A feasible update plan is obtained considering these constraints.

Fig. 2. The first step of four possible update plans: (a) an update plan without considering the CPU load of NFs, (b) an update plan without considering the constraint of flow table, (c) an update plan without considering the constraint of link bandwidth and (d) a feasible update plan.

performances of NFs are closely correlated with their CPU usage. If an NF is overloaded, the packets traversing it will experience increased processing delay or be dropped, leading to severe service degradation. For example, if we move $F_1$ to its new path first, the packets of $F_3$ and $F_1'$ will both be sent to $\mathrm{NF}_3$, and the CPU load of $\mathrm{NF}_3$ will be above $100\%$ as presented in Fig. 2 (a). Thus, it is important to ensure that NFs' CPU utilizations are below a certain level.

**Flow table constraint:** A switch's flow table size is another constraint. A flow table contains entries that specify the forwarding policies for different flows, and its capacity is limited. For example, as depicted in Fig. 2 (b), if $F_2$ is moved to its new path first, $S_2$ has to steer the packets of $F_1$ and $F_2'$. The FTU of $S_2$ will become $50\% + 60\% = 110\%$, which means that $S_2$ cannot support this move and the plan will fail in practice. Therefore, when planning the multi-flow update, it is essential to guarantee that flow table constraints are not violated.

**Link bandwidth constraint:** A link's bandwidth is also a constraint. During the update, several flows may traverse the same links. If the link utilization is too high, congestion occurs and packet loss rate increases. For example, assuming that at the first step $F_3$ is moved to its new path, as presented in Fig. 2 (c), the link between $S_3$ and $S_5$ will carry the traffic of $F_2$ and $F_3'$ and its utilization becomes $110\%$. Thus, the link utilization should be below a certain threshold during the update in order to guarantee congestion-freedom.

The first step of a feasible update plan is shown in Fig. 2 (d), where we split $F_3$ into two subflows each with half of the traffic, and move one subflow to the new path at the first step. Then the CPU load on $\mathrm{NF}_2$ and the link utilization on the link between $S_3$ and $S_5$ will increase by half of that consumed by $F_3$, and the FTU of $S_3$ reaches $70\%$. This is because the rules need to be conservatively added to setup a new path, no matter how much traffic is redirected on it. Then, the update is successfully completed by moving $F_1$, $F_2$ and the remaining subflow of $F_3$, without violating the constraints of NFs, switches and links.

## III. System Model and Problem Formulation

In this section, we present the mathematical model of the SDNFV system and formulate the multi-step network update scheduling as an optimization problem.

### A. System Model

We now formally model the SDNFV system. In such a system, there are $M$ NFs denoted by set $\mathbb{M}$ and $S$ switches denoted by set $\mathbb{S}$, which are connected by $L$ directed links denoted by set $\mathbb{L}$. $K$ active flows need to update their path, and each flow should be processed by several NFs. We define a "flow" as an aggregation of packets that go from the same ingress switch to the same egress switch and traverse the same set of NFs. Thus, the "flow" considered in our model can be a set of actual flows, which may span multiple flow entries. We use $b_j$ to represent the traffic rate of flow $j$, and $c_j$ to represent the CPU load on the NFs to process flow $j$. For a flow, the CPU usage of an NF is proportional to its traffic rate. We also use $d_j$ to represent the number of flow entries that need to be installed per switch to route flow $j$. Note that $d_j \geq 1$.

In the network, the old forwarding paths are denoted by $\{P_1, \cdots, P_j, \cdots, P_K\}$ with $P_j$ as the path of flow $j$, and the new paths are denoted by $\{P_1', \cdots, P_j', \cdots, P_K'\}$. A path from source $u_{sr}$ to destination $u_{de}$ is defined as a list of traversed nodes (including both switches and NFs) and the links connecting them, denoted by $\{u_{sr} = u_0, u_1, u_2, \cdots, u_k = u_{de}\}$, where $u_i \in \mathbb{M} \cup \mathbb{S}$ is the next hop node of $u_{i-1}$ and the link $(u_{j-1}, u_j) \in \mathbb{L}$. Each NF is connected to a switch. The paths are loop-free and each node has only one next hop.

We use $R_b(l)$, $R_f(s)$, and $R_c(m)$ to denote the bandwidth of link $l$, the flow table size of switch $s$ and the processing capacity of NF $m$, respectively. The max number of steps is denoted by $N$. Let $f_i(l)$ represent the maximum traffic load on link $l$ during the $i$th update step. Note that $f_0(l)$ indicates the initial link load. Similarly, we use $n_i(s)$ and $g_i(m)$ to denote the maximum number of flow entries on switch $s$, and the maximum CPU load for NF $m$ in step $i$.

### B. Problem Formulation

In the multi-step network update scheduling, our goal is to finish the update as quickly as possible while considering the resource constraints. Thus, we set the limit on the number of steps to be $N$ and find the optimal update solution with the minimum number of steps. We use binary variable $I_i, i \in \{1, \cdots, N\}$ to indicate whether the network update is finished at step $i$. To further describe how the flows are updated in each

step, we use $x_{ij}$ to denote the fraction of flow $j$ that is updated from $P_j$ to $P'_j$ in step $i$. Thus, $x_{ij}$ is constrained by:

$$\begin{cases} 0 \leq x_{ij} \leq 1, \forall 1 \leq i \leq N; \\ \sum_{i=1}^{N} x_{ij} = 1, \forall j; \end{cases} \tag{1}$$

which ensures that the migrated traffic volume of a flow is not larger than its total rate at any step, and the traffic of every flow should be moved to the new path in $N$ steps. Moreover, $I_i$ is restricted by the linear combination of $x_{ij}$ as follows:

$$\begin{cases} I_i \geq \sum_{j=1}^{K} \frac{1}{N} x_{ij}, i \in \{1, \cdots, N\}; \\ I_i \geq I_{i+1}, i \in \{1, \cdots, N-1\}. \end{cases} \tag{2}$$

Since $I_i$ is a binary variable, we can observe that $I_i = 1$, when any value of $x_{ij}, \forall j$, is above zero; $I_i = 0$, when $x_{ij}, \forall j$, are all zeros. The constraint of $I_i \geq I_{i+1}$ ensures that as long as the update is finished in the $i$th step, there will be no more update operations in subsequent steps. Thus, **the optimization goal** can be represented by $\min \sum_{i=1}^{N} I_i$.

Next, we analyze the constraints of the problem. Concerning the link data rate, $f_0(l)$ represents the initial traffic rate on each link, and it is calculated by $f_0(l) = \sum_{j=1}^{K} b_j \alpha_{jl}$, where if $l \in P_j$, $\alpha_{jl} = 1$; otherwise, $\alpha_{jl} = 0$. In the worst case, the max link utilization in step $i$ occurs, when old flows have not started to migrate their traffic and new flows have already moved to link $l$. In this case, we need to guarantee that the link is not congested. If $l \in P'_j$, $x_{ij} b_j$ should be added to $f_i(l)$; if link $l \in P_j$, $x_{ij} b_j$ should be subtracted from $f_i(l)$; otherwise, the value of $f_i(l)$ will not change. In this way, we can calculate $f_i(l)$ iteratively as follows:

$$f_i(l) = \sum_{j=1}^{K} b_j [(1 - \sum_{a=0}^{i-1} x_{aj}) \alpha_{jl} + (\sum_{a=0}^{i} x_{aj}) \alpha'_{jl}]. \tag{3}$$

If $l \in P'_j$, $\alpha'_{jl} = 1$; otherwise, $\alpha'_{jl} = 0$. We set $x_{0j} = 0$ to include the expression of $f_1(l)$ in this equation.

Similarly, concerning the max CPU usage per NF, we derive the formula of $g_i(m)$ as follows:

$$g_i(m) = \sum_{j=1}^{K} c_j [(1 - \sum_{a=0}^{i-1} x_{aj}) \gamma_{jm} + (\sum_{a=0}^{i} x_{aj}) \gamma'_{jm}]. \tag{4}$$

If $m \in P'_j$, $\gamma'_{jm} = 1$; otherwise, $\gamma'_{jm} = 0$.

With regard to the constraint of flow table size, we focus on the maximum number of flow entries on the switches in each step, denoted by $n_i(s)$. For the original number of flow table entries in each switch, we have $n_0(s) = \sum_{j=1}^{K} d_j \beta_{js}$. The value of $\beta_{js}$ is determined by the network topology as follows. If $s \in P_j$ and the next hop of $s$ is some NF $m$, at most double flow entries are required to forward traffic (in order to steer traffic through the NF), *i.e.*, $\beta_{js} = 2$; if $s \in P_j$, in the case that its next hop is not an NF, $d_j$ flow entries are needed, *i.e.*, $\beta_{js} = 1$. If $s$ does not belong to $P_j$, $\beta_{js} = 0$. Thus, we have:

$$\beta_{js} = \begin{cases} 2, \exists m, (s,m) \in P_j; \\ 1, s \in P_j, \forall m, (s,m) \notin P_j; \\ 0, s \notin P_j. \end{cases} \tag{5}$$

In the worst case, only when the whole flow has been updated, the old flow entries can be removed. Thus, we can derive the expression of $n_i(s)$ as follows,

$$n_i(s) = \sum_{j=1}^{K} d_j (\text{sgn}(1 - \sum_{a=0}^{i-1} x_{aj}) \beta_{js} + \text{sgn}(\sum_{a=0}^{i} x_{aj}) \beta'_{js}). \tag{6}$$

$\text{sgn}(x)$ is the signum function. The expression of $\beta'_{js}$ is similar with that of $\beta_{js}$ in (5), except that $P_j$ is replaced with $P'_j$.

Finally, by combining the objective and the constraints, we formulate the optimization problem $P_1$ as follows:

$$\min \quad \sum_{i=1}^{N} I_i$$

$$s.t. \begin{cases} g_i(m) \leq \eta \cdot R_c(m), \forall i, m; & \text{(7a)} \\ f_i(l) \leq \theta \cdot R_b(l), \forall i, l; & \text{(7b)} \\ n_i(s) \leq R_f(s), \forall i, s; & \text{(7c)} \\ (1), (2); & \text{(7d)} \end{cases}$$

where $0 < \theta = \max_{i,l} \frac{f_i(l)}{R_b(l)} \leq 1$ represents the maximum allowed link utilization during the update, and $0 < \eta \leq 1$ represents the CPU load threshold. The number of steps in the optimal solution of $P_1$ is denoted by $N_{opt}$. We can prove that $P_1$ is NP-complete by transforming it to the problem of Fastest Update Scheduling (FUS), which has been proved NP-complete in Theorem 2 of [13].

## IV. Approximation Algorithm

We aim to seek an efficient algorithm for the multi-flow network update problem $P_1$. We first change it to $P_2$ that has a linear objective, and then relax the integer constraints to obtain a linear programming problem $P_3$. Since the solution of $P_3$ does not always satisfy the constraints of $P_1$, we further propose a linear programming based approximation algorithm *LIPBA* with polynomial computation complexity. Now we introduce the main idea of the solution process.

### A. Problem Reformulation

Since the objective function of the problem $P_1$ contains integer variables, we first attempt to reform the objective function. We vary $N$ from 1 to $N_{max}$ with every increment of 1, and examine whether a feasible update solution with $N$ steps exists. With a given value of $N$, we use maximum link utilization as the optimization goal to minimize potential congestions, which is represented by $\min \theta$. Thus, we formulate the optimization problem $P_2$ as follows,

$$N : 1 \sim N_{max}$$

$$\begin{cases} \min \quad \theta \\ s.t. \quad \text{Constraints } (1), (7a)\text{–}(7c). \end{cases} \tag{8}$$

**Until** a feasible solution is obtained with $N_{opt}^{(2)}$ steps.

**Algorithm 1** *LIPBA*: Our LP-based approximation algorithm.

1: Solve $P_3$ by increasing $N$ from 1 to $N_{max}$, and obtain the optimal solution $x_{ij}$ and the minimum number of steps $N_{opt}^{(3)}$.
2: For each flow $j$, choose to update its entire traffic in step $i$ independently with probability $x_{ij}$.
3: If some constraint is not satisfied, obtain a feasible solution by adjusting the update plan.

If a feasible solution cannot be found within $N_{max}$ steps, we suppose that the network update problem is infeasible in practice. Note that there does not always exist an update plan for an arbitrary network update problem without violating these three kinds of constraints. The solution of $P_2$ is equal to that of $P_1$, which is denoted by $N_{opt}^{(2)} = N_{opt}$.

### B. Linear Programming Relaxation

Let us consider the reformulated problem $P_2$. Since the flow table constraints in (7c) contain integer variables, $P_2$ is also NP-complete [11]. Thus, we relax the constraints and transform the problem into a Linear Programming (LP) problem. For simplicity, we use $\sigma_{ij}$ to denote $\sum_{a=0}^{i} x_{aj}$. We can observe that $\text{sgn}(1-\sigma_{(i-1)j})\beta_{js} + \text{sgn}(\sigma_{ij})\beta'_{js} \geq (1-\sigma_{(i-1)j})\beta_{js} + \sigma_{ij}\beta'_{js}$. If and only if $\sigma_{(i-1)j}, \sigma_{ij} \in \{0,1\}$, the equality holds. Based on this inequality, we relax the flow table constraints and obtain an LP problem $P_3$ as follows,

$$
\begin{aligned}
&N: 1 \sim N_{max} \\
&\begin{cases}
\min \quad \theta \\
s.t. \begin{cases}
\text{Constraints } (1), (7a), (7b); \\
\sum_{j=1}^{K} d_j[(1-\sigma_{(i-1)j})\beta_{js} + \sigma_{ij}\beta'_{js}] \leq R_f(s), \forall s, i.
\end{cases}
\end{cases} \\
&\textbf{Until} \text{ a feasible solution is obtained with } N_{opt}^{(3)} \text{ steps.}
\end{aligned} \tag{9}
$$

Note that $P_3$ is a linear relaxation of $P_2$, which can be solved in polynomial time. The optimal solution of $P_3$ provides a lower bound for $P_2$, *i.e.*, $N_{opt}^{(3)} \leq N_{opt}$. If the optimal solution of $P_3$ satisfies $\sigma_{ij} \in \{0,1\}, \forall i, j$, it meets the constraints of $P_2$ and is also the optimal solution for $P_2$. Otherwise, the optimal solution of $P_3$ is not a feasible solution for $P_2$. Thus, we further propose an approximation algorithm to obtain a feasible solution.

### C. An LP-based Approximation Algorithm

As shown in Algorithm 1, we present a LInear Programming Based Approximation algorithm (LIPBA). It first obtains the optimal solution of $P_3$, and then adjusts the solution to fit the feasible region of $P_2$. Specifically, for each flow $j$, the algorithm updates its traffic in step $i$ independently, with probability equal to $x_{ij}$.

If the solution of Algorithm 1 satisfies the constraints of $P_3$, it is also the optimal solution for $P_2$, *i.e.*, $N_{opt}^{(4)} = N_{opt}^{(2)}$, where $N_{opt}^{(4)}$ denotes the number of steps in the optimal solution of *LIPBA*. Otherwise, we can obtain a feasible solution by adjusting the update plan: if a link capacity or CPU constraint of a step is violated, we add an update step and separate a part of the traffic to be updated in this step into the extra step; if a flow table constraint of a step is violated, we add an update step and schedule some flows to be updated in the extra step. $P_3$ can be solved in polynomial time, and the computation complexity of the adjustment process is also polynomial of the instance size. Thus, *LIPBA* is a polynomial time algorithm for the multi-flow update problem.

## V. Performance Evaluation

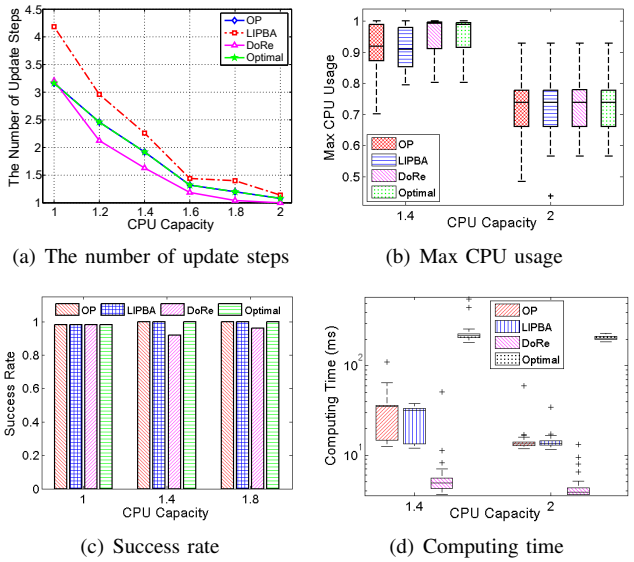
### A. Experimental Settings

We evaluate the performance of *LIPBA* and compare it with three other update schemes: 1) *Optimal*, which seeks the optimal solution of the flow update problem $P_1$ using YALMIP toolbox [8] and Gurobi Optimizer; 2) *OP*, which uses the same toolbox to solve the problem $P_2$; and 3) *DoRe*, which is an extension to an existing greedy heuristic algorithm [17] to solve $P_1$.

*DoRe* first normalizes the resource utilization of each flow by the available resource capacities of its traversed links, NFs, and switches, and defines the resource with the largest utilization as the dominant resource. Then all flows are sorted by the utilizations of their dominant resources and examined in descending order. If updating a flow does not violate resources, it will be scheduled in the current step; otherwise, it will be skipped and re-examined in the next step. The scheduling process is repeated until the network update is completed, or it ends when there is not enough resource to update the remaining flows.

Since NFs are widely deployed in datacenters, we select the Fat Tree [9] topology to carry out simulations, which is widely used in data center networks, with 20 nodes and 32 links. For the traffic rate of each flow, we use the real packet-level traces from a data center of a university (EDU1 in [10]). The capacity of each link is set to 1 GBps, and the flow table size of each switch is set to 1000 rules. The number of flows $K$ is set to 40. We scale down the traffic data to fit into the link capacity, and assume the number of flow entries required by different flows follows uniform distribution of $[d-\delta_d, d+\delta_d]$, where $d$ is set to be 7 and $\delta_d$ is set to be 3. The CPU usage per flow follows uniform distribution of $[c-\delta_c, c+\delta_c]$, and we set $c$ to be 0.05 and $\delta_c$ to be 0.04.

Initially, each flow should go through a certain set of NFs, and the number of NFs on each flow's path follows the uniform distribution of $[1, 3]$. We select $M$ switches to connect to the NFs. Note that in Fat Tree only the switches at the edge layer can be connected with NFs. We consider a typical network update scenario: Forwarding-Change, where the link weights are reassigned to simulate events which can cause flow updates, such as failed links or dynamic traffic engineering. We set $M$ to 10 and $M_{mig}$ to 5 by default. The simulations are run on a computer with an Intel Core2 Quad CPU Q9400 and 4 GB memory.




(a) The number of update steps (b) Max CPU usage

(c) Success rate (d) Computing time

Fig. 3. Performance of the update solutions as a function of the CPU capacity of NFs under the topology of Fat Tree.

### B. Preliminary Results

We compare the performance of our solution *LIPBA* with the optimal one. Since CPU allocated to NFs is an important resource in SDNFV systems, we vary the CPU capacity per NF from 1 to 2 units to evaluate its impact on the update process. We run the simulation 50 times using the topology of Fat Tree, and present the averaged results in Fig. 3. As shown in Fig. 3 (a), the number of update steps of all solutions decreases, when NFs have more powerful CPUs. *Optimal* and *OP* obtain the same number of steps, which indicates that transforming *Optimal* into *OP* is a proper method. *LIPBA* takes about 1.3x steps compared with the optimal solution, when the normalized value of each NF's CPU capacity is 1. The max CPU usage during the update process decreases with the increase in CPU capacity, as represented in Fig. 3 (b). When the CPU capacity is 1.4 and 1.8 units, $DoRe$ cannot achieve $100\%$ success rate[1], as shown in Fig. 3 (c). However, all the other solutions successfully solve the update problem for all cases of the simulation. Regarding the computing time as shown in Fig. 3 (d), we observe that $Optimal \gg OP > LIPBA > DoRe$. Specifically, *LIPBA* saves 10x-100x computing time compared with $Optimal$. By contrast, *DoRe* computes the solution with the shortest time, but its success rate is up to $8\%$ lower.

## VI. Conclusion

In this paper, we focus on reducing the update time for multi-flow update in SDNFV systems. The link bandwidth, flow table memory and CPU capacities of NFs are three interrelated network resources that impact the update process. By jointly considering these three constraints, we formulate the network update problem as a mixed integer programming problem, which is NP-complete. Thus, we propose a linear approximation based polynomial-time algorithm to solve it. Extensive simulations based on real traffic traces demonstrate that the proposed algorithm approaches to the optimal solution with only 10x-100x less computing time and achieves high success rate.

**Acknowledgments.** This work was supported by the National Basic Research Program of China (973 Program) under grants 2013CB329105, the National Nature Science Foundation of China (Grant No. 61273214), and also supported (in part) by European Union's Horizon 2020 research and innovation program under the ENDEAVOUR project (grant agreement 644960).

[1] We say that an update is successful if it can be completed without violating any of the problem constraints.

## References

[1] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," in *IEEE Comm. Mag.*, vol. 53, no. 2, pp. 90–97, Feb. 2015.

[2] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. SIGCOMM'12*, August 13-17, 2012, pp. 323–334.

[3] A. Gember-Jacobson, R. Viswanathan, C. Prakash, *et al.*, "OpenNF: Enabling innovation in network function control," in *Proc. SIGCOMM'14*, August 17-22, 2014, pp. 163–174.

[4] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using FlowTags," in *Proc. NSDI'14*, April 2-4, 2014, pp. 533–546.

[5] Z. A. Qazi, C. C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using SDN," in *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 27–38, Aug. 2013.

[6] Y. Zhang , N. Beheshti , L. Beliveau, *et al.*, "Steering: A software-defined networking for inline service chaining," in *Proc. ICNP'13*, October 7-10, 2013, pp. 1–10.

[7] S. Jain, A. Kumar, S. Mandal, *et al.*, "B4: Experience with a globally-deployed software defined WAN," in *Proc. SIGCOMM'13*, August 12-16, 2013, pp. 3–14.

[8] J. Löfberg, "YALMIP: A toolbox for modeling and optimization in MATLAB Computer Aided Control Systems Design," in *Proc. IEEE Int. Symp. Comput. Aided Control Syst. Design 2004*, September 2-4, 2004, pp. 284–289.

[9] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. SIGCOMM'08*, August 17-22, 2008, pp. 63–74.

[10] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *Proc. IMC'10*, November 1-3, 2010, pp. 267–280.

[11] C. Y. Hong, S. Kandula, R. Mahajan, *et al.*, "Achieving high utilization with software-driven WAN," in *Proc. SIGCOMM'13*, August 12-16, 2013, pp. 15–26.

[12] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz,, "zUpdate: Updating Data Center Networks with Zero Loss," in *Proc. SIGCOMM'13*, August 12-16, 2013, pp. 411–422.

[13] X. Jin, H. H. Liu, R. Gandhi, *et al.*, "Dynamic scheduling of network updates," in *Proc. SIGCOMM'14*, August 17-21, 2014, pp. 539–550.

[14] N. P. Katta, J. Rexford, and D. Walker, "Incremental Consistent Updates," in *Proc. HotSDN'13*, August 16, 2013.

[15] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. B. Godfrey, "Enforcing Customizable Consistency Properties in Software-Defined Networks," in *Proc. NSDI'15*, May 4-6, 2015, pp. 73–85.

[16] P. Perešíni, M. Kuźniar, M. Canini, and D. Kostić., "ESPRES: Transparent SDN Update Scheduling," in *Proc. HotSDN'14*, August 22, 2014, pp. 73–78.

[17] Y. Liu, Y. Li, Y. Wang, A. V. Vasilakos, and J. Yuan, "Achieving Efficient and Fast Update for Multiple Flows in Software-Defined Networks," in *Proc. DCC'14*, August 18, 2014, pp. 77–82.

[18] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "A distributed and robust sdn control plane for transactional network updates," in *Proc. INFOCOM'15*, April 6-May 1, 2015.

[19] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/Merge: System Support for Elastic Execution in Virtual Middleboxes," in *Proc. NSDI'13*, April 2-5, 2013, pp. 227–240.