Placing Traffic-Changing and Partially-Ordered NFV Middleboxes via SDN

Wenrui Ma, Jonathan Beltran, Deng Pan, and Niki Pissinou

***Abstract*—Network Function Virtualization (NFV) enables flexible implementation of network functions, also called middleboxes, as virtual machines running on standard servers. However, the flexibility also makes it a challenge to optimally place middleboxes, because a middlebox may be hosted by different servers at different locations. The middlebox placement challenge is further complicated by additional constraints, including the capability of middleboxes to change traffic volumes and dependency between them. In this paper, we address the optimal placement challenge of NFV middleboxes for the data plane using a software-defined networking (SDN) approach. First, we formulate the optimization problem to place traffic-changing and interdependent middleboxes. When the flow path is predetermined, we design optimal algorithms to place a non-ordered or totally-ordered middlebox set, and propose a low-complexity solution for the general scenario of a partially-ordered middlebox set after proving its NP-hardness. When the flow path is not predetermined, we show that the problem is NP-hard even for a non-ordered or totally-ordered middlebox set, and propose an efficient traffic and space aware routing algorithm. We have evaluated the proposed algorithms using large scale simulations and a real application based SDN prototype, and present extensive evaluation results to demonstrate the superiority of our design over benchmark solutions.**

***Index Terms*—Network function virtualization, software-defined networking, middlebox.**

I. Introduction

NETWORK function virtualization (NFV) [1] transforms the implementation of network functions, also called middleboxes, from proprietary hardware appliances to virtual machines (VMs) running on industry standard servers [2]. Leveraging the underlying virtualization technology, VM-based software middleboxes bring many benefits that were not previously available, such as accelerated time-to-market, reduced hardware and operation cost, and elastic scalability [3].

The flexibility of VMs also poses a challenge for efficient NFV implementation. In particular, traditional hardware middleboxes are deployed at fixed locations in the network, and leave no choice of service locations. On the contrary, in an NFV network, there are multiple NFV servers with standard hardware that can host VMs of arbitrary network functions [2]. It is thus possible to optimize the network performance by carefully selecting the location to place a software middlebox among multiple candidate servers. Improper placement decisions may cause inefficient flow paths and traffic jam.

Furthermore, the NFV service location challenge is complicated by the traffic changing effects of middleboxes. Unlike switches and routers that forward traffic without changing its volume, middleboxes may change the traffic volumes of processed flows, and may do it in different ways. For example, the Citrix NetScaler SD-WAN WAN optimizer compresses traffic before sending it to the next hop, and may reduce the traffic volume by 80% [4]. On the other hand, the BCH(63, 48) encoder, used in wireless communications for forward error correction (FEC), increases the traffic volume by 31% due to the checksum overhead [5].

We use the following example to illustrate the traffic changing effects of middleboxes. Consider an enterprise network of three nodes v_1, v_2 and v_3, and two links (v_1, v_2) and (v_2, v_3). A flow f passes the three nodes before leaving the enterprise network, and its initial traffic rate is 1. Two middleboxes m and m' need to be applied to the flow. m will double the traffic rate, while m' will decrease it by half. By placing m on v_1 and m' on v_3, the loads of links (v_1, v_2) and (v_2, v_3) will be $1 \times 2 = 2$, as shown in Fig. 1(a). However, by placing m on v_3 and m' on v_1, the loads of both links are reduced to $1 \times 0.5 = 0.5$, as shown in Fig. 1(b).

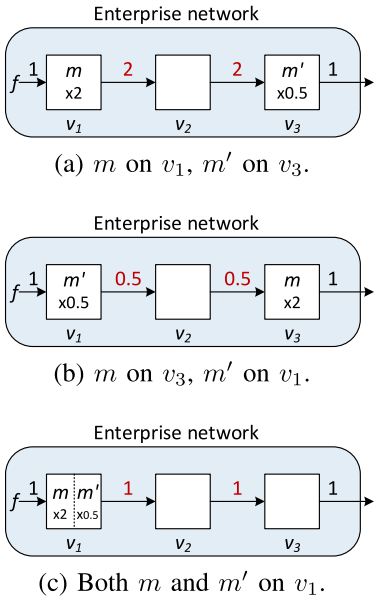
The placement of middleboxes is also constrained by the dependency relation that may or may not exist between middleboxes [6]. For instance, an IPSec decryptor is usually placed before a NAT gateway [7], while a VPN proxy can be placed either before or after a firewall [8]. In the above example, if there is a constraint for m to be applied before m', then the placement scheme in Fig. 1(b) would violate the constraint. However, by placing both m and m' on v_1 in such a way that traffic is processed by m before m', we can still reduce the link loads from 2 to 1 as in Fig. 1(c), in contrast to the case in Fig. 1(a).

In this paper, we study the optimal placement of NFV middleboxes, with a focus on elephant flows, which are large-size and persistent flows [9]. In our design, elephant flows are served by middleboxes specifically placed for them, while mice flows, i.e., small-size and transient flows, are served by already deployed middleboxes instead of new ones. Such differentiated processing is based on the following considerations. First, since elephant flows constitute a majority of the

Manuscript received April 23, 2019; revised August 11, 2019; accepted September 29, 2019. Date of publication October 9, 2019; date of current version December 10, 2019. The associate editor coordinating the review of this article and approving it for publication was S. Schmid. *(Corresponding author: Deng Pan.)*

The authors are with the School of Computing and Information Sciences, Florida International University, Miami, FL 33199 USA (e-mail: wma006@fiu.edu; jbelt021@fiu.edu; pand@fiu.edu; pissinou@fiu.edu).

Digital Object Identifier 10.1109/TNSM.2019.2946347



(a) m on v_1, m' on v_3.

(b) m on v_3, m' on v_1.

(c) Both m and m' on v_1.

Fig. 1. Traffic changing effects of middleboxes.

network traffic [10], optimizing middlebox placement for elephant flows will effectively balance the network load. Second, because mice flows are transient, the yield of their optimization is marginal. Finally, due to their short duration, utilizing existing middleboxes will help mice flows avoid the instantiation delay required by new middleboxes. Elephant flows can be statically determined based on the application types, such as data backup or VM migration, or dynamically detected using many existing techniques in the literature [9], [11]. The processing of mice flows is not in the scope of this paper, and will be studied in future work.

Our solution utilizes the emerging Software-Defined Networking (SDN) technology [12] as the implementation platform, because it enables efficient optimization by decoupling the network control plane and data plane. In the prototype implementation that will be presented in Section VI, we have developed an SDN controller module to manage the elephant flows: routing the flows and determining their middlebox locations.

Our main contributions in this paper are summarized as follows.

1) We formulate the Traffic Aware Placement of Interdependent Middleboxes (TAPIM) problem, considering in particular a generalized partial order for the middlebox dependency relation, as a graph optimization problem with the objective to achieve a network with balanced link load.
2) For topologies with predetermined paths, such as trees, we design optimal algorithms for the special case when the middlebox set is a non-ordered or totally-ordered one. For the general case when the dependency relation is a partial order, we show that the TAPIM problem is NP-hard by reduction from the Clique problem, and propose an efficient solution to convert a partially-ordered set to a totally-ordered one.
3) For topologies without predetermined paths, we show that the TAPIM problem is NP-hard even for a non-ordered or totally-ordered middlebox set. Our proposed solution then works in two steps: first finding a path with enough resources to host all the middleboxes, and then placing the middleboxes on the given path.
4) We have implemented the proposed algorithms in the ns-3 simulator and a SDN based prototype, and present extensive simulation and experiment results to demonstrate the effectiveness of our design.

The remaining of this paper is organized as follows. Section II provides a brief overview of related work. Section III formulates the TAPIM problem. Sections IV and V solve the TAPIM problem with and without predetermined flow paths, respectively. Section VI describes the implementation of our prototype. Section VII presents experiment and simulation results. Finally, Section VIII concludes the paper.

II. Related Work

In this section, we briefly review research results in several related categories and highlight our differences.

It is challenging to optimize the placement of NFV middleboxes, especially with limited available resources. Cohen *et al.* [13] formulate the NFV location problem, and propose near optimal approximation algorithms for middlebox resource allocation. Kuo *et al.* [14] study the joint problem of middlebox placement and path selection by considering the correlation between the link and server usages. Sang *et al.* [15] formulate the middlebox placement problem with the objective to minimize the total number of instances to provide a specific service, and propose asymptotically optimal greedy algorithms with constant approximation ratios. Feng *et al.* [16] propose fast approximation algorithms for the NFV service distribution problem. Wang *et al.* [17] model resource allocation in NFV as a multi-resource load balancing problem, and propose a solution to minimize the maximum dominant load. Ma *et al.* [18] study the middlebox placement problem by considering the traffic changing effects, with the objective to minimize the maximum link load. Fei *et al.* [19] seek a proactive approach to provision new instances for overloaded virtual network functions (VNFs) in the cloud. The formal model defined in this work differs from the above ones in considering general partial-order dependency relations between middleboxes. In addition to the formal model, this work also presents practical algorithms and a prototype system for design validation.

A service function chain is a set of network functions to be performed according to a given order. Multiple models [6], [20]–[25] are proposed to formulate service chain implementation by solving combined middlebox placement and path computation, mostly using a linear programming based approach. Zhang *et al.* [26] formulate the VNF chains placement problem as a bin-packing problem, and propose a priority-driven weighted algorithm. Even *et al.* [27] propose a randomized approximation algorithm with performance guarantee. Lukovszki and Schmid propose a deterministic and asymptotically optimal online algorithm for admission

TABLE I
LIST OF NOTATIONS

Notation	Meaning
$v, sc[v]$	node, its space capacity
$(u,v), bc[u,v]$	link, its bandwidth capacity
$weight(u,v,l)$	weight of link (u,v) with load l
$t, t(u,v)$, $t_{in}(i), t_{out}(i)$	initial traffic rate, rate on link (u,v), input/output rate at i^{th} hop
m, M	middlebox, set of middleboxes
$ratio[m]$	traffic changing ratio of m
$m \leftarrow m'$, $ratio[m] \leftarrow ratio[m']$	m' depends on m
$route(v,i)$	1 if v is i^{th} hop, 0 otherwise
$place(m,i)$	1 if m placed at i^{th} hop, 0 otherwise
v_i	i^{th} hop of given flow path
m_j	j^{th} middlebox of given total order chain
$T(r)$	self-dependent middlebox tree with root r

control of service chains [28], and approximation algorithms based on submodularity for incremental deployment of middleboxes [29]. Amiri *et al.* show that general waypoint routing in most practical scenarios, especially with directed graphs, is computationally intractable [30], and propose polynomial-time algorithms for graphs of bounded treewidth [31]. Different from the above works, this work studies not only the totally-ordered service chain but also non-ordered and partially-ordered middlebox sets. In addition to the dependency relations, this work adds a new dimension by considering the traffic changing effects of NFV middleboxes.

Due to its capability of global optimization, SDN [32] is commonly adopted as the control protocol to automate and simplify the NFV service provisioning. Multiple architectures [33], [34] are proposed to integrate SDN and NFV for efficient traffic maneuver. For example, SIMPLE [35] is a SDN-based policy enforcement layer for efficient middlebox-specific traffic steering. StEERING [36] is a scalable framework based on SDN to dynamically route traffic through a sequence of services. To support dynamic service chaining, Fayazbakhsh *et al.* propose an extended SDN architecture called FlowTags [37], in which services add packet tags to provide the necessary context for systematic policy enforcement. Our work differs from the above ones by not only implementing the correct policies through SDN, but also considering the middlebox traffic changing effects and different dependency relations.

III. PROBLEM STATEMENT

In this section, we formulate the Traffic Aware Placement of Interdependent Middleboxes (TAPIM) problem. Table I summarizes the list of notations for easy reference.

Consider a network represented by a directed graph $G = (V, E)$. A node $v \in V$ has a space capacity of $sc[v]$[1]≥ 0, i.e., the number of middleboxes that can be hosted. Since resource allocation for virtual machines is a well studied problem [38]–[40], we assume for simplicity that each middlebox needs one unit space, and additional processing power can be achieved by multiple load-balanced middlebox instances [41]. A link $(u,v) \in E$ has a bandwidth capacity $bc[u, v] \geq 0$, i.e., the amount of available bandwidth. The existing load of the link is denoted as $load[u, v]$.

For route calculation, each link $(u,v) \in E$ is assigned a weight, denoted as $weight(u, v, l)$, which is a non-decreasing function of the link load l, i.e., $\forall l \leq l', weight(u,v,l) \leq weight(u,v,l')$. A broad category of weight functions satisfy the non-decreasing requirement, such as the ones used by the popular Cisco EIGRP [42] and OSPF [43] protocols. The non-decreasing link weight function helps load-balance network traffic when the routing protocol aims to minimizes the path cost, which is defined as the weight sum of all the path links.

A flow f is defined as a 4-tuple (src, dst, t, M), in which $src \in V$ is the source node, $dst \in V$ is the destination node, t is the initial traffic rate when f arrives at the ingress switch of the network, and M is the set of required middleboxes. (In case the flow needs multiple instances of the same middlebox for increased processing power, M may include multiple copies of the same middlebox type.)

Each middlebox $m \in M$ has an associated traffic changing ratio $ratio[m] \geq 0$, which is the ratio between the traffic rates of a flow after and before being processed by m. In reality, the traffic change ratio may be a constant (e.g., for a BCH encoder) or a variable (e.g., for a WAN optimizer). For convenient and practical modeling, we use the average in case of a variable in the following formulation.

The *dependency* relation $\leftarrow$ is defined as a strict partial order on M that is

1) Irreflexive: $m \nleftarrow m$,
2) Transitive: $m \leftarrow m'$ and $m' \leftarrow m''$ then $m \leftarrow m''$, and
3) Asymmetric: $m \leftarrow m'$ then $m' \nleftarrow m$.

Intuitively, $m \leftarrow m'$ means that the middlebox m should be applied before m'. (With a slight abuse of notation, when there is no confusion, we also use $ratio[m] \leftarrow ratio[m']$ to concisely describe the traffic changing ratios of m and m' and their dependency relation.) For easy representation, define $depend[m, m'] = 1$ if $m \leftarrow m'$, and 0 otherwise.

When the flow f enters the network, a multi-hop path, denoted as *route*, will be assigned for the flow, which is a decision variable defined as

$$route(v,i) = \begin{cases} 1, & \text{if } v \in V \text{ is the } i^{th} \text{ hop on the path.} \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

We define i to start from one, and denote the last hop number as n for convenience. Repeating nodes are allowed on a path to enable more general solutions, and hence the path is also referred to as a walk in classic graph theory terms. Due to state consistency requirements such as in-order packet delivery, splitting traffic among multiple paths is not in the scope of this paper.

In addition, a placement scheme, denoted as *place*, will determine the location for each middlebox $m \in M$, which is a decision variable defined as

$$place(m,i) = \begin{cases} 1, & \text{if } m \in M \text{ is placed at the } i^{th} \text{ hop.} \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

[1]We use square brackets [] to denote properties or known values, and round brackets () denote to functions or variables.

To avoid resource contention between elephant flows and achieve load balance, we do not consider sharing of a middlebox by multiple elephant flows, but instead leave the remaining processing capacity of a placed middlebox to mice flows.

Use $t_{in}(i)$ and $t_{out}(i)$ to denote the incoming and outgoing traffic rate of flow f at the i^{th} hop on the path, respectively. If f is processed by a single middlebox m at the i^{th} hop, then $t_{out}(i) = t_{in}(i)ratio[m]$. Note that the incoming traffic rate at the flow source is the initial traffic rate, i.e., $t_{in}(1) = t$. For convenience, use $t(u, v) = \sum_{i\in[1,n-1]} route(u, i)route(v, i + 1)t_{out}(i)$ to denote the traffic rate of f on link (u, v).

Consistent with most popular routing protocols, such as Cisco EIGRP and OSPF, our optimization objective is to determine *route* and *place* to minimize the path cost of flow f as shown in Equation (3). It should be noted that the proposed solutions can easily adapt to other optimization objectives, such as minimizing the maximum link load in the network.

$$\textbf{minimize} \sum_{i=1}^{n-1} \sum_{u\in V} \sum_{v\in V} route(u, i)route(v, i + 1) \; weight(u, v, t(u, v) + load[u, v]) \tag{3}$$

subject to the following constraints:

$$\forall i > n, \forall v \in V : route(v, i) = 0 \tag{4}$$

$$route(src, 1) = 1, route(dst, n) = 1 \tag{5}$$

$$\forall v \in V : \sum_{i\in[1,n]} \sum_{m\in M} place(m, i)route(v, i) \leq sc[v] \tag{6}$$

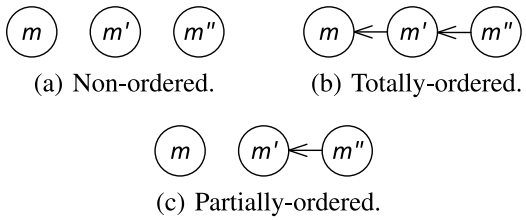
$$\forall (u, v) \in E : t(u, v) + load[u, v] \leq bc[u, v] \tag{7}$$

$$\forall m \in M : \sum_{i\in[1,n]} place(m, i) = 1 \tag{8}$$

$$\forall m, m' \in M : \left(\sum_{i\in[1,n]} place(m', i)\, i - \sum_{i\in[1,n]} place(m, i)i \right) \times depend[m, m'] \geq 0 \tag{9}$$

$$\forall i \in [1, n] : t_{out}(i) = t_{in}(i) \prod_{m\in M} \big(1 + place_f(m, i)(ratio[m] - 1)\big) \tag{10}$$

Brief explanation of the model is as follows. Equation (3) defines the optimization objective. To discourage repeated links on the flow path, when the flow traverses the same link multiple times, the link weight of each traverse will be counted separately in the path cost. Equation (4) states that the n^{th} hop is the last hop of the flow path. Equation (5) enforces the first and last hops of the flow path to be the source *src* and destination *dst*, respectively. Equation (6) states that, for a node v, the total space demand of hosted middleboxes $\sum_{i\in[1,n]} \sum_{m\in M} place(m, i)route(v, i)$ should not exceed its space capacity $sc[v]$. Equation (7) states that, for a link (u, v), the aggregate load of all flows traversing it $t(u, v)+load[u, v]$ should not exceed its bandwidth capacity $bc[u, v]$. Equation (8) states that a middlebox m should be installed once and only once. Equation (9) enforces the dependency relation between middleboxes, or in other words m must be placed no later than m' if the former is depended on by the latter. Equation (10) states that, the outgoing flow traffic rate at a hop, i.e., $t_{out}(i)$, is equal to the incoming rate, i.e., $t_{in}(i)$, multiplying the traffic changing ratios $ratio[m]$ of all the middleboxes m placed at this hop. It also ensures flow conservation, in the sense that no flow traffic can be generated or terminated at an intermediate node, except the effects of middleboxes.



(a) Non-ordered. (b) Totally-ordered.

(c) Partially-ordered.

Fig. 2. Examples of non-ordered, totally-ordered, and partially-ordered middlebox sets.

The above formulation considers only a single flow instead of multiple flows because of the following practical considerations. First, as will be seen in Sections IV and V, the TAPIM problem for a single flow is already NP-hard. Therefore, the multi-flow version will be even harder and unlikely to have practical solutions. Second, for certain applications, it is a small probability for multiple flows to arrive at the same time. For example, decisions for WAN traffic engineering are taken at large time scales, and can be scheduled for sequential processing. Nevertheless, our algorithms are capable of handling multiple flows by updating state variables, such as the server and link capacity, after each flow arrival and departure, and applying the algorithms to individual flows.

IV. Middlebox Placement With Predetermined Paths

In this section, we propose solutions for the TAPIM problem when the flow path, i.e., *route*, is predetermined. For example, in trees, there is a unique path between any pair of leaves. We look at three cases of the problem. First, for the special case when there is no dependency between any middleboxes, we propose the Non-Ordered Set Placement algorithm that uses the least-first-greatest-last rule. Next, for the special case when there is a total dependency order on the middlebox set, we propose the dynamic programming based Totally-Ordered Set Placement algorithm. Finally, for the general scenario of a partial dependency order on the middlebox set, we prove that it is NP-hard by reduction from the Clique problem, and propose an efficient solution to convert a partially-ordered set to a totally-ordered set. Since the flow path is given, for easy description, we denote the i^{th} hop node on the path as v_i.

A. Non-Ordered Middlebox Set

We start with the special case that the middlebox set M is a *non-ordered set*, i.e., $\forall m, m' \in M, m \nleftarrow m'$ and $m' \nleftarrow m$. Thus, different middleboxes can be placed in an arbitrary order. An example is shown in Fig. 2(a), where no dependency

exists between middleboxes. We propose the *Non-Ordered Set Placement* (NOSP) algorithm, and show its optimality.

The basic idea is least-first-greatest-last, i.e., to shrink the flow as early as possible by placing the middleboxes that decrease the traffic rate from the path head, and expand the flow as late as possible by placing the middleboxes that increase the traffic rate from the path tail.

If there are enough spaces on the path, i.e., $\sum_{i\in[1,n]}\sum_{v\in V}\left(route[v,i]sc[v]/\sum_{i'\in[1,n]}route[v,i']\right) \geq |M|$, the NOSP algorithm places the middleboxes as follows.

1) Sort all the middleboxes $m \in M$ based on their traffic changing ratios *ratio*[*m*].
2) Place the middleboxes *m* with *ratio*[*m*] < 1 from the path head in an increasing order of their traffic changing ratios. When a node has no more space, continue with the succeeding node on the path.
3) Place the middleboxes *m* with $ratio[m] \geq 1$ from the path end in an decreasing order. When a node has no more space, continue with the preceding node.
4) Check each link (v_i, v_{i+1}) on the path to ensure no one is oversubscribed.

The pseudo code of NOSP is shown in Algorithm 1. Line 1 sorts all the middleboxes $m \in M$ in the increasing order of their traffic changing ratios *ratio*[*m*]. Lines 2 to 14 are the first stage to place the middleboxes with traffic changing ratios less than one from the head of the flow path. In detail, line 2 initializes by starting with the first hop and the middlebox with the least traffic changing ratio, i.e., *M*[1] after sorting. Line 3 is the loop to process middleboxes with ratios less than one. Line 4 checks if the current hop v_i has available spaces. If yes, Line 5 places the middlebox *M*[*j*] on v_i, decrements the number of available spaces $sc[v_i]$ at v_i, and increments the middlebox index *j*. Lines 7 and 8 check whether the current hop v_i is already the last hop and there are still middleboxes with ratios less than one to install. If yes, line 9 exits since there is no more space on the flow path. Otherwise, if the current hop v_i is not the last hop, line 11 continues with the next hop on the path. Lines 15 to 27 place the middleboxes with ratios greater than or equal to one from the tail of the flow path in a similar manner as above. Lines 28 to 32 check each path link to enforce the bandwidth capacity constraint.

The time complexity of NOSP is $O(\max(|M|\log|M|, n))$, because it sorts the middleboxes with complexity of $O(|M|\log|M|)$, and checks the links with complexity of $O(n)$.

Lemma 1: The Non-Ordered Set Placement algorithm minimizes the flow rate on each link of the path.

The lemma can be proved by contradiction, and the detail is omitted due to space limitations.

Theorem 1: The Non-Ordered Set Placement algorithm achieves the minimum path cost.

Proof: By Lemma 1, NOSP minimizes the flow rate on each path link. Thus, the total load of each link as the sum of the existing load and flow rate is also minimized. Given that the link weight function *weight*(*u*, *v*, *l*) is a non-decreasing function of the total link load *l*, NOSP minimizes the weight of each path link and subsequently the path cost. ■

Algorithm 1 Non-Ordered Set Placement

Require: $G, f, route, M[1..|M|]$
Ensure: *place*

```
1: sort m ∈ M[1..|M|] in non-decreasing order of ratio[m]
2: i = 1, j = 1
3: while j ≤ |M| and ratio[M[j]] < 1 do
4:    while sc[v_i] > 0 and i ≤ n and ratio[M[j]] < 1 do
5:       place(M[j], i) = 1; sc[v_i] − −; j + +
6:    end while
7:    if sc[v_i] = 0 and j ≤ |M| and ratio[M[j]] < 1 then
8:       if i = n then
9:          exit with insufficient-space error
10:      else
11:         i++
12:      end if
13:   end if
14: end while
15: i = n, j = |M|
16: while j ≥ 1 and ratio[M[j]] ≥ 1 do
17:   while sc[v_i] > 0 and j ≥ 1 and ratio[M[j]] ≥ 1 do
18:      place(M[j], i) = 1; sc[v_i] − −; j − −
19:   end while
20:   if sc[v_i] = 0 and i ≥ 1 and ratio[M[j]] ≥ 1 then
21:      if i = 1 then
22:         exit with insufficient-space error
23:      else
24:         i−−
25:      end if
26:   end if
27: end while
28: for i from 1 to n − 1 do
29:   if t(v_i, v_{i+1}) + load[v_i, v_{i+1}] > bc[v_i, v_{i+1}] then
30:      exit with insufficient-bandwidth error
31:   end if
32: end for
```

B. Totally-Ordered Middlebox Set

Next, we solve the other special case of TAPIM when the middlebox set is a totally-ordered set, i.e., $\forall m, m' \in M$, either $m \leftarrow m'$ or $m' \leftarrow m$, or in other words the middleboxes form a dependency chain. An example is shown in Fig. 2(b), in which *m* must be placed before *m′* and *m′* before *m″*. Although the placement order of the middleboxes is known, it is still necessary to determine the optimal placement location for each middlebox, because there may be an excessive number of available spaces on the flow path. For easy description, we use m_j to denote the j^{th} middlebox from the head of the dependency chain, and v_i to denote the i^{th} hop node on the flow path, where *i* and *j* start from 1.

We propose a dynamic programming based algorithm called *Totally-Ordered Set Placement* (TOSP) based on the following observation. Use TOSP(*i*, *j*) to denote the minimum weight sum of the first *i* links when place the first *j* middleboxes, i.e., m_1 to m_j, on the first *i* hops, i.e., v_1 to v_i, of the flow path. The optimal substructure gives the following recursive

formula.

$$\mathrm{TOSP}(i,j) = \begin{cases} w(1;1,j), \text{ if } i = 1. \\ \min_{x\in[1,j+1]} \\ (\mathrm{TOSP}(i-1,x-1) + w(i;x,j)), \text{ otherwise.} \end{cases} \tag{11}$$

where $w(i; x, j)$ is the weight of link (v_i, v_{i+1}) when placing middleboxes m_x to m_j on node v_i, i.e.,

$$w(i;x,j) = \begin{cases} weight\Big(v_i, v_{i+1}, t\prod_{y\in[1,j]} ratio[m_y] + load[v_i, v_{i+1}]\Big) \\ \quad \text{if } sc[v_i] \geq j-x+1 \text{ and } i < n. \\ 0, \quad \text{if } sc[v_i] \geq j-x+1 \text{ and } i = n. \\ 0, \quad \text{if } x > j. \\ \infty, \text{ otherwise.} \end{cases} \tag{12}$$

Equation (11) states that if $i = 1$, TOSP(i, j) is simply the weight of the first path link when placing all the first j middleboxes on the first hop v_1. Otherwise, the optimal result TOSP(i, j) to place the first j middleboxes on the first i hops is to select the minimum link weight sum among $j + 1$ possible solutions, in which the x^{th} solution places the first $x - 1$ middleboxes on the first $i - 1$ hops, i.e., TOSP($i - 1$, $x - 1$), and places the remaining middleboxes m_x to m_j on the i^{th} hop v_i, i.e., $w(i; x, j)$.

Equation (12) calculates the weight of the i^{th} path link, i.e., (v_i, v_{i+1}), when placing middleboxes m_x to m_j at the i^{th} hop v_i, and sets it to zero if $x > j$ or infinity if v_i has fewer than $j - x + 1$ available spaces. A special case is when v_i is the last hop in question, or in other words (v_i, v_{i+1}) does not exist or is not of interest. Then, the weight is zero if v_i has sufficient spaces or infinity otherwise. Note that if $x \leq j$ and v_i has sufficient spaces, $w(i; x, j)$ does not depend on x. Specifically, as long as v_i has no less than $j - x + 1$ spaces to host middleboxes m_x to m_j, the weight of link (v_i, v_{i+1}) is the same, which simplifies the calculation of TOSP(i, j) as the sum of the minimum sub-solution $\min_{x\in[j-sc[v_i]+1,j+1]}\{\mathrm{TOSP}(i-1,x-1)\}$ and a constant. Thus, we can rewrite the recursive relationship as:

$$\begin{aligned} &\mathrm{TOSP}(i,j) \\ &= \min_{x\in[1,j+1]}\{\mathrm{TOSP}(i-1,x-1) + w(i;x,j)\} \\ &= \min_{x\in[j-sc[v_i]+1,j+1]}\{\mathrm{TOSP}(i-1,x-1) + w(i;x,j)\} \\ &= \min_{x\in[j-sc[v_i]+1,j+1]}\{\mathrm{TOSP}(i-1,x-1)\} \\ &\quad + weight\left(v_i, v_{i+1}, load[v_i, v_{i+1}] + t\prod_{y\in[1,j]} ratio[m_y]\right) \end{aligned} \tag{13}$$

The pseudo code of TOSP is shown in Algorithm 2. Lines 1 to 7 conduct the initialization by solving the sub-problems with only the first hop v_1. In detail, line 1 checks if v_1 has j spaces. If yes, TOSP(*1*,*j*) is assigned the weight of the first link in line 3, and otherwise infinity in line 4. Lines 8 and 9 start the iteration to recursively calculate the remaining sub-problems. Based on the previous results, lines 10 to 15 finds among the viable schemes the one with the minimum link weight sum to place a portion of middleboxes in the first $i - 1$ hops. Finally, line 16 calculates the optimal TOSP(i, j) by adding the minimum link weight sum of the first $i - 1$ hops and the link weight of the last hop.

Algorithm 2 Totally-Ordered Set Placement

Require: $G, f, route, M[1..|M|], depend$
Ensure: $place$
1: **for** $j = 1$ **to** $|M|$ **do**
2: **if** $sc[v_1] \geq j$ **then**
3: TOSP$(1, j)$ = $weight(v_1, v_2, t\prod_{y=1}^{j} ratio[m_y] + load[v_1, v_2])$
4: **else**
5: TOSP$(1, j) = \infty$
6: **end if**
7: **end for**
8: **for** $i = 2$ **to** n **do**
9: **for** $j = 1$ **to** $|M|$ **do**
10: $min = \infty$
11: **for** $x = j - sc[v_i] + 1$ **to** $j + 1$ **do**
12: **if** TOSP$(i - 1, x - 1) < min$ **then**
13: $min = TOSP(i - 1, x - 1)$
14: **end if**
15: **end for**
16: TOSP(i, j)= min + $weight(v_i, v_{i+1}, t\prod_{y=1}^{j} ratio[m_y] + load[v_i, v_{i+1}])$
17: **end for**
18: **end for**

TABLE II
TOSP EXAMPLE

TOSP(i, j)	$i = 1$	$i = 2$	$i = 3$
$j = 0$	1	2	–
$j = 1$	0.5	1	–
$j = 2$	∞	1.5	1

Table II shows an example to apply TOSP to a flow f in the network of Fig. 1. The detail of the flow is as follows: $src = v_1$, $dst = v_3$, $t = 1$, and M is a total order chain $0.5 \leftarrow 2$ (the concise notation for two middleboxes m_1 and m_2 with $ratio[m_1] = 0.5$, $ratio[m_2] = 2$, and $m_1 \leftarrow m_2$). Assume that the space capacity of each node is 1, i.e., $\forall v_i, sc[v_i] = 1$, and the link weight is equal to the link load, i.e., $weight(u, v, l) = l$. The algorithm starts from the first column: by placing 0, 1, and 2 middleboxes at the 1^{st} hop, we have TOSP(1, 0) = 1, TOSP(1, 1) = 0.5, and TOSP(1, 2) $= \infty$ (because v_1 can host at most one middlebox), respectively. In the second column, TOSP(2, 1) = 1 is the minimum of two options to place either 0 or 1 middlebox at the 1^{st} hop, i.e., min{TOSP(1,0)+w(2; 1,1) = 1+0.5 = 1.5, TOSP(1, 1)+w(2; 2, 1) = 0.5+0.5 = 1}. In the third column, TOSP$(3, 2) = \mathrm{TOSP}(2, 1) + w(3; 2, 2) =$ 1, and the results for the sub-problems of TOSP(3, 0) and TOSP(3, 1) are not necessary because TOSP(3, 2) is already solved.

When the flow path *route* is not efficient and contains repeating nodes, the above algorithm may obtain a sub-optimal result. The reason is that, different hops of a repeating node share middlebox spaces, but the above algorithm processes

those hops always from the path head, and thus assigns earlier hops higher priority.

A simple solution is to first enumerate all the possibilities to divide the shared spaces among different hops of a repeating node, and then apply TOSP to each possible division. For example, if the flow path contains a repeating node with s spaces that appears twice at the i_1^{th} hop v_{i_1} and the i_2^{th} hop v_{i_2}, we view v_{i_1} and v_{i_2} as two independent nodes by allocating $x \in [0, s]$ spaces to v_{i_1} and $s - x$ spaces to v_{i_2}. TOSP is then applied to each different x value, and the minimum path cost among all the cases is the optimal solution.

When there is no repeating node on the flow path, the time complexity of the TOSP algorithm is $O(n|M|^2)$, because the dynamic programming table has n rows and $|M|$ columns, and it takes up to $O(|M|)$ time to calculate each table entry. When there are r repeating nodes and each node has up to s spaces and appears in up to h hops, the time complexity is $O(s^{(h-1)r} n|M|^2)$, because there are $s^{(h-1)r}$ possible divisions of shared spaces, and the time complexity to apply TOSP to each division is $O(n|M|^2)$. Fortunately, efficient routing paths should have small or zero r and h values, because they have few repeating nodes.

C. Partially-Ordered Middlebox Set

We now solve the general scenario where the dependency relation is a partial order. The following theorem shows the NP-hardness of the problem.

Theorem 2: The TAPIM problem with a predetermined path for a partially ordered middlebox set is NP-hard.

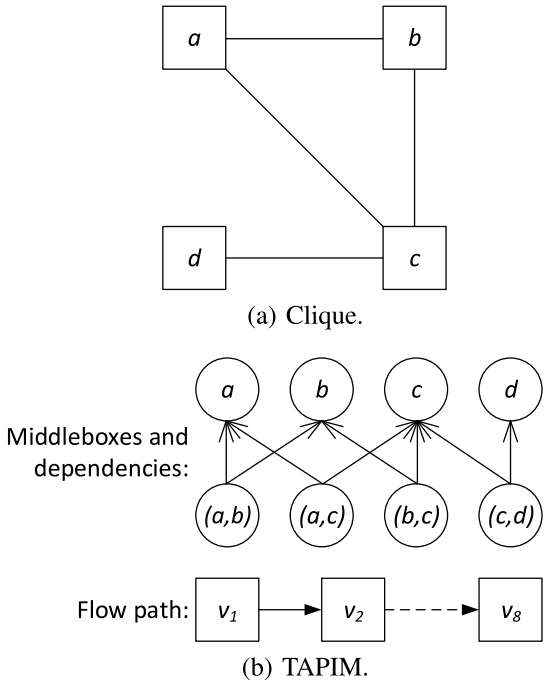
Proof: We prove by reduction from the Clique problem [44]. The clique problem decides whether an undirected graph $G = (V, E)$ has a clique of size k, which is a complete sub-graph with k vertices and $\binom{k}{2}$ edges. For example, the graph in Fig. 3(a) has a clique of size 3: ({*a, b, c*},{(*a, b*), (*a, c*),(*b, c*)}).

Given an instance of the Clique problem with a graph $G = (V, E)$, an instance of the TAPIM problem can be constructed in polynomial time as follows.

1) For each vertex $p \in V$, create a vertex middlebox m_p with $ratio[m_p] = 2$.
2) For each edge $(p, q) \in E$, create an edge middlebox $m_{(p,q)}$ with $ratio[m_{(p,q)}] = 2^{-k/\binom{k}{2}}$.
3) The middlebox corresponding to an edge (p,q) depends on the two middleboxes corresponding to its two incident vertices p and q, i.e., $m_p \leftarrow m_{(p,q)}$ and $m_q \leftarrow m_{(p,q)}$.
4) There is a single flow f with the initial traffic rate of one, i.e., $t = 1$. The path has $|V|+|E|$ nodes. Use v_i to denote the i^{th} node on the path. Each node has a space capacity of one, i.e., $sc[v_i] = 1$.
5) Each link on the path has a bandwidth capacity of infinity, i.e., $bc[v_i, v_{i+1}] = \infty$. The link $(v_{k+\binom{k}{2}}, v_{k+\binom{k}{2}+1})$ is called the critical link, with its weight being one if the link load is no more than one and infinity otherwise, i.e.,

$$weight((v_{k+\binom{k}{2}}, v_{k+\binom{k}{2}+1}), l) = \begin{cases} 1, & \text{if } l \le 1. \\ \infty, & \text{if } l > 1. \end{cases} \tag{14}$$

The weight of any other link is always zero.



(a) Clique.

(b) TAPIM.

Fig. 3. Reduction from Clique to TAPIM with predetermined path.

Next, we show that if the graph $G = (V, E)$ has a clique of size k, then the constructed TAPIM instance has a minimum path weight of one. Assume the solution clique of size k is $G' = (V', E')$, the solution for TAPIM is constructed as follows.

1) For each vertex $p \in V'$, place the corresponding middlebox m_p one by one starting from the path head v_1.
2) For each edge $(p, q) \in E'$, continue placing the corresponding middlebox $m_{(p,q)}$ along the path.
3) For each remaining vertex $p \in V \setminus V'$, continue placing the corresponding middlebox m_p.
4) For remaining edges in $(p, q) \in E \setminus E'$, continue placing the corresponding middlebox $m_{(p,q)}$.

Since $G' = (V', E')$ is a complete sub-graph, for each edge middlebox $m_{(p,q)}$ placed in Step 2, its two predecessor vertex middleboxes m_p and m_q must have been placed in Step 1. Furthermore, since Step 3 places all the remaining vertex middleboxes, the predecessors of all edge middleboxes placed in Step 4 are satisfied. Therefore, there is no dependency violation. Also, when the flow arrives at the critical link $(v_{k+\binom{k}{2}}, v_{k+\binom{k}{2}+1})$, it has traversed $|V'| = k$ vertex middleboxes and $|E'| = \binom{k}{2}$ edge middleboxes, its flow rate is thus $1 \times 2^k \times (2^{-k/\binom{k}{2}})^{\binom{k}{2}} = 1$. As a result, the weight of the critical link is one, and the entire path cost is also one.

Conversely, if the constructed TAPIM instance has a minimum path cost of one, then the graph $G = (V, E)$ has a clique of size k denoted as $G' = (V', E')$. We show by contradiction that the first $k + \binom{k}{2}$ middleboxes placed on the path must be k vertex middleboxes and $\binom{k}{2}$ edge middleboxes.

1) For contradiction, assume that the first $k + \binom{k}{2}$ middleboxes on the path include more than k vertex middleboxes and subsequently fewer than $\binom{k}{2}$ edge middleboxes. Since the traffic changing ratio of a vertex middlebox is 2 and that of an edge middlebox is $2^{-k/\binom{k}{2}}$, the flow rate will be greater than one when it comes to the critical link, and the weight of the critical link would be infinity instead of one.
2) For contradiction, assume that there are fewer than k vertex middleboxes and subsequently more than $\binom{k}{2}$ edge middleboxes. With fewer than k vertex middleboxes, the number of edges generated by those vertices in G must be fewer than $\binom{k}{2}$, and thus there must exist an edge middlebox whose predecessors have not been satisfied.

Thus, the first $k + \binom{k}{2}$ middleboxes placed on the path are exactly k vertex middleboxes and $\binom{k}{2}$ edge middleboxes, and the predecessor vertex middleboxes of all edge middleboxes are included in this set. Therefore, the sub-graph G' corresponding to those vertex and edge middleboxes form a clique of size k. ∎

After proving the NP-hardness, our solution to place a partially-ordered middlebox set is to first convert it to a totally-ordered middlebox set and then apply TOSP.

Following the idea of the least-first-greatest-last rule used in NOSP, the objective of the conversion algorithm is to arrange the middleboxes in the resulting total order chain in the increasing order of their traffic changing ratios. The intuitive solution is thus to iteratively find the middleboxes without dependencies, remove among them the one with the least traffic changing ratio, and add it to the end of the total order chain. For example, given four middleboxes with the following traffic changing ratios and dependencies: $1.4 \leftarrow 1.5$ and $1.6 \leftarrow 0.1$, the conversion result will be the following total order chain: $1.4 \leftarrow 1.5 \leftarrow 1.6 \leftarrow 0.1$.

To increase the solution search space, we also propose a lookahead approach. Specifically, the above intuitive solution compares only individual middleboxes, i.e., those without dependencies, and picks the one with the least traffic changing ratio in each round. Instead, the lookahead approach optimizes by combining multiple middleboxes as a group and calculating the aggregate traffic changing ratio of the group. Define a self-dependent middlebox tree rooted from middlebox r, denoted as $T(r)$, to be a set of middleboxes that all depend on r, i.e., $\forall m \in T(r) \setminus \{r\}, r \leftarrow m$, and depend on only middleboxes in the set, i.e., $\forall m \in T(r)$, if $m' \leftarrow m$ then $m' \in T(r)$. We say that the size of $T(r)$ is k if it contains k middleboxes, i.e., $|T(r)| = k$. The traffic changing ratio of the tree is the product of the traffic changing ratios of all the middleboxes in the tree, i.e., $ratio[T(r)] = \prod_{m \in T(r)} ratio[m]$. In the above example, $1.6 \leftarrow 0.1$ is a self-dependent tree of size 2 with 1.6 being the root, and its traffic changing ratio is $1.6 \times 0.1 = 0.16$.

The conversion algorithm with a lookahead value of k works in iterations as follows. In each iteration, the algorithm first finds all the middleboxes with no dependency. Using each of such middleboxes as the root, the algorithm calculates the self-independent tree of size up to k that has the minimum traffic changing ratio. Among all the calculated trees with different root middleboxes, the algorithm selects the one with the minimum traffic changing ratio, removes its root, and adds it to the total order chain. For the above example, the first iteration generates two trees of size up to 2: 1.4 of size 1 with 1.4 being the root, and $1.6 \leftarrow 0.1$ of size 2 with 1.6 being the root. Since the traffic changing ratio of the latter 0.16 is less than that of the former 1.4, the root of the latter will be removed. The resulting total order chain after the algorithm converges is thus: $1.6 \leftarrow 0.1 \leftarrow 1.4 \leftarrow 1.5$.

Algorithm 3 Converting Partially-Ordered Set to Totally-Ordered Set With Lookahead of k

Require: $k, M, depend$
Ensure: M'
1: **for** each middlebox $m \in M$ **do**
2: **if** m has no dependency **then**
3: $minratio[m] = \min_{x=1}^{k}\{$ratio of size x self-independent tree with root $m\}$
4: **else**
5: $minratio[m] = \infty$
6: **end if**
7: **end for**
8: **for** $j = 1$ **to** $|M|$ **do**
9: select in M middlebox m with least $minratio[m]$
10: $M'[j++] = m$
11: $M = M \setminus \{m\}$
12: **for** each middlebox m' directly depending on m **do**
13: $minratio[m'] = \min_{x=1}^{k}\{$ratio of size x self-independent tree with root $m'\}$
14: **end for**
15: **end for**

The pseudo code of the conversion algorithm is shown in Algorithm 3. Lines 1 to 7 conduct the initialization by calculating the self-independent tree with the minimum traffic changing ratio for each middlebox without dependency. Lines 8 to 15 are the iterations to build the result total order chain. Line 9 finds among the middleboxes without dependency the one with the minimum ratio self-independent tree, line 10 adds it to the total order chain, and line 11 removes it from the original middlebox set. Lines 12 to 14 calculate for each child of the removed middlebox its minimum ratio self-independent tree.

When the lookahead parameter $k = 1$ or 2, the time complexity of the conversion algorithm is $O(|M| \log |M|))$, because there are up to $|M|$ iterations, and the time complexity to select the middlebox with the minimum traffic changing ratio is $O(\log |M|)$ using a heap. When $k = 2$, the optimal self-dependent trees of size up to 2 with each middlebox being the root can be pre-calculated in $O(|M|)$ time. Since $|M|$ is usually small, and $k = 2$ will be sufficient in most cases.

V. Middlebox Placement Without Predetermined Path

In this section, we solve the TAPIM problem when the flow path, i.e., *route*, is not predetermined. We start by showing that the TAPIM problem without a predetermined path is NP-hard

even for a non-ordered or totally-ordered set. We then propose a two-step solution by first finding a flow path with sufficient spaces and then applying the algorithms in Section IV to place middleboxes on the given path.

A. NP-Hardness

When the flow path is not known, the TAPIM problem becomes NP-hard, even for a non-ordered or totally-ordered middlebox set. The challenge to solve TAPIM without a predetermined path is to find an efficient path with sufficient spaces. More specifically, [30, Th. 3] shows that it is NP-hard just to decide the existence of a path that traverses a single node at a fixed location in a directed graph. Therefore, even if the flow f requests only one middlebox, i.e., $|M| = 1$, and there is a single available space in the entire network, i.e., $\sum_{v \in V} sc[v] = 1$, it is still impossible to find a viable path in polynomial time in general.

B. Traffic and Space Aware Routing

Our solution to TAPIM without a predetermined path works in two steps by first finding a viable path for the flow and then applying the algorithms in Section IV to place the middleboxes on the determined path.

We first propose the fast Traffic And Space Aware Routing (TASAR) algorithm. The basic idea is to originate from the source, iteratively route to a nearby node with spaces until sufficient spaces have been accumulated, and finally go to the destination.

In detail, TASAR works as follows. It starts by calculating the number of spaces needed on the path in addition to those in the source and destination, i.e., $|M| - sc[src] - sc[dst]$. Next, it enters iterative loops to accumulate the necessary number of spaces. In the x^{th} iteration, Dijkstra's algorithm [44] is applied from v_x, with $v_1 = src$, to find the nearest (in terms of the path cost) node with spaces, denoted as v_{x+1}, and add the path from v_x to v_{x+1} to the flow path *route*. If sufficient spaces have been accumulated, i.e., $|M| - sc[src] - sc[dst] - \sum_{i=1}^{x+1} sc[v_i] \leq 0$, the iteration stops, and Dijkstra's algorithm is applied for the last time to find the minimum cost path from the current node v_{x+1} to the destination *dst*. Otherwise, if more spaces are needed, the iteration continues.

The pseudo code of TASAR is shown in Algorithm 4. Line 1 calculates the number of missing spaces. Line 2 initializes the loop between line 3 and 7, which uses Dijkstra's algorithm to find the nearest node with spaces and appends it to the flow path. Finally, line 8 applies Dijkstra's algorithm again to find the minimum cost path to the destination.

The time complexity of the heuristic is $O(|M|(|E| + |V|\log|V|))$, because the there will be up to $O(|M|)$ iterations, and the time complexity of each iteration is that of Dijkstra's algorithm $O(|E| + |V|\log|V|)$.

Algorithm 4 Traffic and Space Aware Routing

Require: $G, src, dst, |M|$
Ensure: *route*
1: $missing = |M| - sc[src] - sc[dst]$
2: $v_1 = src; i = 1$
3: **while** $missing > 0$ **do**
4: $\quad v_{i+1}$ = nearest node from v_i with spaces
5: $\quad$ append to flow path *route* the section from v_i to v_{i+1}
6: $\quad missing = missing - sc[v_{++i}]$
7: **end while**
8: append to flow path *route* the section from v_i to *dst*

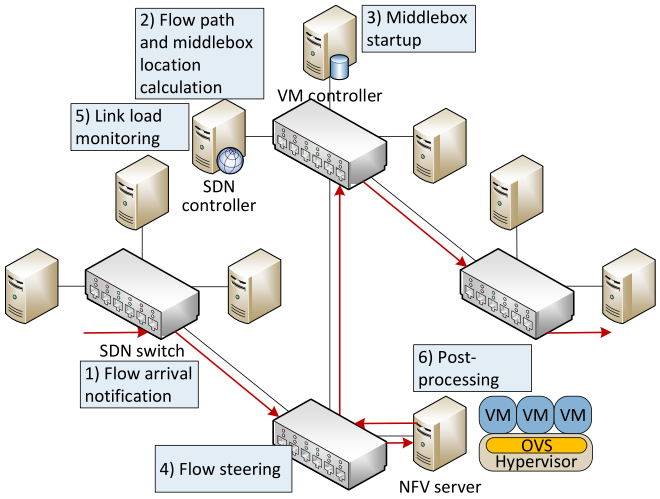


Fig. 4. Flow processing.

VI. Prototype Implementation

To evaluate the proposed design in realistic environments, we have built an SDN based prototype, using the open-source SDN controller Floodlight [45] and network emulator Mininet [46]. The flow routing and middlebox placement algorithms are implemented as a module in Floodlight, and VMs are created in Mininet to emulate switches and NFV servers. Furthermore, to generate realistic traffic in the experiments, we have also developed real application based middlebox programs. In this section, we describe the prototype implementation and discuss deployment issues.

A. Flow Processing

Our implementation utilizes OpenFlow [47], the underlying communication protocol [12] of SDN, to define and steer flows. OpenFlow supports flexible flow definition based on wildcard matching of many different packet headers. An SDN switch receives flow definitions and associated actions from the SDN controller, and stores them in its flow table. An incoming packet that matches a flow table entry will trigger the corresponding actions. In a production environment, it may be necessary to define flows at fine granularity using multiple packet header fields such as IP addresses and port numbers, so that flows of different applications can be distinguished. In the comparatively simple environment of the prototype, it is sufficient for our purpose to define a flow in the switch tables based on the source and destination IP addresses and input switch port.

A flow is processed in the following steps as illustrated in Fig. 4.

1) Flow Arrival Notification: As explained in the introduction, elephant flows can be statically or dynamically detected using different techniques [9], [11] in the literature. After the ingress switch detects a new elephant flow, it performs a lookup in its flow table using the flow header fields. If no flow table entry is found, the switch wraps the first packet of the flow in an *OFPT_PACKET_IN* OpenFlow message and forwards it to the Floodlight controller. The controller thus learns the arrival of a new flow. Otherwise, if a flow table entry is found, the flow will be sent to the next hop according to the routing information.

2) Flow Path and Middlebox Placement Calculation: A module is developed for Floodlight to calculate the flow path and middlebox locations. The module is registered as an *OFPT_PACKET_IN* message listener, and will be triggered upon receiving such a message sent by the ingress switch. It first examines the header information, such as IP addresses and port numbers, of the packet wrapped in the *OFPT_PACKET_IN* message to determine the application type, and applies a set of middleboxes based on predefined profiles for different applications. Next, the module runs the proposed algorithms to obtain a flow path if necessary, and decides the optimal location on the path for each middlebox. In Floodlight, a path is defined as a list of (*DatapathId*, *OFPort*) tuples, where *DatapathId* represents a switch and *OFPort* represents a port of the switch.

3) Middlebox Startup: Once the placement locations have been calculated, corresponding middleboxes will be booted or waken up. This is done with the help of a VM controller, such as Virsh or VMware vCenter, that can remotely start, shutdown, suspend, or resume VMs. In detail, the information of middleboxes to be placed at each node is passed from the SDN controller to the VM controller, which then sends commands to the hypervisor of the selected NFV servers. The hypervisor will load the predefined image to boot up an instance of the corresponding middlebox or wake it up if it had been put to sleep.

4) Flow Steering: To steer a flow to follow the calculated path and visit middleboxes, the SDN controller configures the switch flow tables via OpenFlow. On the one hand, to control flow paths, Floodlight sends an *OFPT_FLOW_MOD* OpenFlow message to each switch on the flow path. The message specifies the matching fields to identify a flow and the output port for the matched flow. On the other hand, to specify middlebox locations, the controller generates three types of *OFPT_FLOW_MOD* messages.

- The first type is sent to the switch on the routing path, instructing the switch to detour packets to the connected NFV server.
- The second type is sent to the Open vSwitch (OVS) [48] in the hypervisor of the NFV server, telling the OVS to forward the packets to one of the hosted middlebox VMs.
- The third type is also sent to the OVS in the hypervisor, asking the OVS to forward the packets back to the flow path after they are being processed by the middlebox.

5) Link Load Monitoring: The Floodlight controller has a built-in statistics module, which can periodically issue port statistics requests and collect replies sent by switches. Based on the statistical information collected from the switches, our module estimates the load of each link using the exponential weighted moving average to calculate the link weight.

6) Post Processing: To release resources after processing a flow, we set a non-zero *idle_timeout* for each entry in the switch flow table. When a flow finishes, its corresponding flow table entries will be automatically removed after the time out. The middlebox VMs will also be shut down or put to sleep after a certain amount of idle time.

B. Middlebox Development

To conduct experiments in realistic environments, we have also developed real application based middlebox programs. Since a normal TCP/UDP socket processes only packets destined to it, the middlebox program utilizes the *libpcap* library [49] to capture all packets from specified interfaces even if it is not the destination. Three types of middleboxes with different traffic changing ratios are developed: a compressor that decreases the traffic volume, an encoder that increases the traffic volume, and a firewall that either completely passes or stops a flow.

1) Compressor: The compressor is developed using the *zlib* [50] library. For every captured packet, the compressor compresses its payload using the *compress* API of the *zlib* library, and then sends it back to the interface where the packet was originally captured using the *pcap_inject* API from the *libpcap* library. Since the actual compression ratio for each packet is a variable that depends on the payload, we use the average of the testing files as traffic changing ratio of the compressor, which is 0.8.

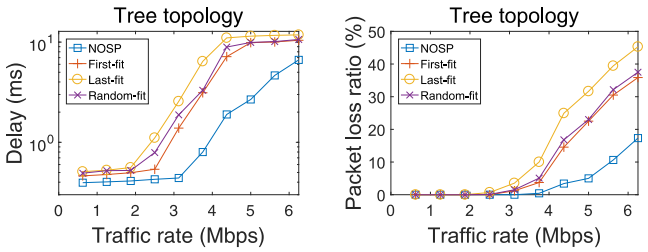
Such a traffic changing ratio is reasonable compared with that of real applications that decrease traffic volumes. For example, WAN optimizers at the sender side accelerate data transfer by compressing traffic before sending it to the next hop, and Citrix's NetScaler SD-WAN WAN optimizer achieves compression rates from 5:1 to 300:1 [4]. Video stream transcoders convert video streams from one format to another on the fly, and Cisco's Digital Media Encoder 1000 [51] supports converting captured video to the MPEG-4/H.264 format that has compression rates from 352:1 to 6086:1 [52], [53].

2) Encoder: The encoder is developed based on open-source Bose-Chaudhuri-Hocquenghem (BCH) code implementation [54]. It adds a series of BCH checksums for the payload of each captured packet. In detail, it divides the packet payload into 36-bit sections, and uses the *encode_bch* API from the BCH(48, 36, 5) implementation to append a 12-bit checksum for each section. As a result, the traffic changing ratio of the encoder is approximately 1.3 ($\approx 48/36$).

Such a traffic changing ratio is reasonable compared to that of real applications that increase traffic volumes. For example, the BCH(31, 21) encoder [55], [56] used in wireless sensor networks add 10 coding bits for every 21 data bits, and hence has a traffic changing ratio of 1.48 ($\approx 31/21$). Video stream transcoders can also convert a stream from a low-bit-rate format to a high-bit-rate format for clients with limited computation resources, and the popular Plex Media Server [57] is capable of transcoding from MPEG4 to MPEG2, which will

result in a traffic changing ratio of 2 [58], [59]. Finally, WAN Optimizers at the receiver side decompress the received compressed traffic to recover the original data, and hence increase the traffic volume up to 300 times.

3) Firewall: The firewall is implemented as a simple rule based packet filter. It matches the header of each captured packet against a set of admission rules that are defined on IP addresses and port numbers, and discards the packet if there is no match. The traffic changing ratio of the firewall is thus one for an admitted flow and zero for a rejected flow.



(a) End-to-end delay. (b) Packet loss ratio.

Fig. 5. NOSP simulation results.

VII. Simulation and Experiment Results

We use a combination of simulations and experiments for performance evaluation. We have conducted simulations to obtain performance data in large scale networks, and experiments that are based on the prototype implementation to validate the design. In this section, we present extensive simulation and experiment results to show the effectiveness of our algorithms. The source code of the simulation and experiment programs is available at the project page [60].

A. Simulation Results

The simulations are conducted in the ns-3 simulator, and the following performance metrics are used for benchmark comparisons.

1) *End-to-end delay:* the interval to transmit a packet from its source to destination.
2) *Packet loss ratio:* the percentage of packets lost with respect to packets sent.

Longer end-to-end delays and higher packet loss ratios indicate inefficient flow paths and middlebox locations, which make certain links become oversubscribed. Under oversubscription, excessive packets will accumulate in the switch queues, and eventually exceed queue capacities and be dropped.

To reflect the burstiness of realistic traffic, we adopt the on-off traffic model [61]. When a flow f is in the on state, its initial traffic rate t is the product of a baseline rate and a random number between 0.5 to 1.5; when in the off state, its traffic rate is zero. A flow is in each of the two states for 50% of the time. There are two candidate middlebox sets with different traffic changing ratios and dependency relations, and each flow will randomly choose one of them. Each link in the network has a bandwidth capacity of 100 Mbps and a propagation delay of 2 μs. The presented data are the average of ten simulation runs, each lasting two minutes.

1) Placing Non-Ordered Set With Predetermined Path: For the NOSP algorithm that places a non-ordered middlebox set on a predetermined path, since there are no existing solutions for the studied problem, we designed the following three benchmark algorithms. Same as NOSP, all the benchmark algorithms sort the middleboxes based on their traffic changing ratios before placing them.

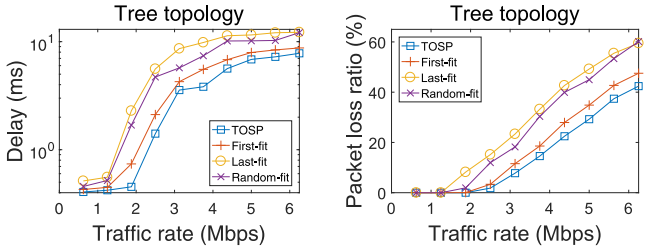
1) *First-fit:* continuously placing the sorted middleboxes in the increasing order from the head of the flow path.
2) *Last-fit:* continuously placing the sorted middleboxex in the decreasing order from the tail of the flow path.
3) *Random-fit:* randomly placing the sorted middleboxes on random nodes on the path that have spaces.

We pick the tree topology, since is a popular choice among institutional networks, and there is only a single path between any pair of nodes. We set up a four-layer quad-tree with 21 switches and 64 hosts. Each switch has 13 spaces to ensure sufficient spaces for all flows. The link weight is set using the Cisco EIGRP [42] metric with only K_2 being one and other parameters being zero, or in other words the link weight is inversely proportional to the percentage of remaining bandwidth. Each host generates a flow to a random destination. The two candidate sets of middleboxes are: {0.7, 0.8, 1.1, 1.2} and {0.8, 0.9, 1.1, 1.3}. The baseline traffic rate of each flow ranges from 0.625 to 6.25 Mbps with a stride of 0.625 Mbps.

Fig. 5(a) shows the average end-to-end delays of the four algorithms. We can see that NOSP consistently achieves the shortest delay due to its optimal middlebox placement scheme. On the other hand, Last-fit has the worst performance, because it places middleboxes at the path end, and the flow rate is $1\times$ on most links of the path. By contrast, First-fit achieves relatively shorter delay by placing middleboxes at the beginning of the path. The reason is that half of the flows picked the first set of middleboxes with an aggregate traffic changing ratio of $0.7 \cdot 0.8 \cdot 1.1 \cdot 1.2 = 0.74$, and the other half picked the second set with a ratio of $0.8 \cdot 0.9 \cdot 1.1 \cdot 1.3 = 1.03$, so on average the middleboxes placed at the path head would reduce the traffic rate of a flow to $(0.74+1.03)/2 = 0.885\times$, which is the traffic rate on most links of the path. Finally, the delay of Random-fit is between that of Last-fit and First-fit due to its randomized strategy.

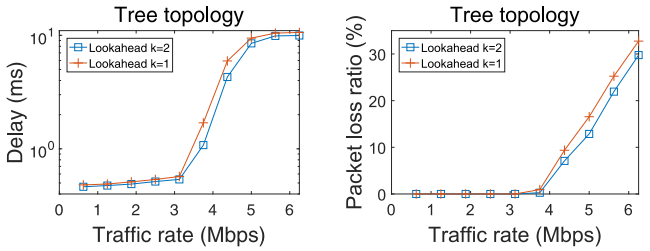
Fig. 5(b) plots the packet loss ratio data. We can observe a similar trend that NOSP always achieves the lowest packet loss ratio. When the flow traffic rate is small, all the algorithms have zero packet loss ratios. Compared with NOSP, other algorithms experience packet losses at smaller traffic rates, and their ratios increase much faster.

2) Placing Totally-Ordered Set With Predetermined Path: Next, we evaluate the TOSP algorithm to place a totally-ordered set with similar benchmark algorithms as above, in which First-fit, Last-fit, and Random-fit place the middleboxes based on the given total order chain from the path head, tail, and randomly, respectively. The traffic changing ratios and dependency chains of the two candidate sets of middleboxes are: $\{0.8 \leftarrow 1.1 \leftarrow 0.7 \leftarrow 1.2\}$ and $\{1.2 \leftarrow 0.7 \leftarrow 1.1 \leftarrow 0.8\}$. Other simulation settings are the same as above.



(a) End-to-end delay. (b) Packet loss ratio.

Fig. 6. TOSP simulation results.



(a) End-to-end delay. (b) Packet loss ratio.

Fig. 7. Partial to total order conversion simulation results.

As shown in Fig. 6(a), TOSP achieves the shortest end-to-end delay because of its dynamic programming based optimal middlebox placement scheme. Similar as above, the delays of the other three algorithms increase in the sequence of First-fit, Random-fit, and Last-fit. The packet loss ratio results in Fig. 6(b) are consistent, and TOSP outperforms others.

3) Placing Partially-Ordered Set with Predetermined Path: To evaluate the placement of partially-ordered middlebox sets on predetermined paths, we use the proposed algorithm to convert the partially-ordered sets to fully-ordered sets, and then apply TOSP. We adjust the lookahead parameter k from one to two and compare their performances. The traffic changing ratios and the dependencies of the two candidate sets of middleboxes are: $\{1.1 \leftarrow 0.8, 1.2 \leftarrow 0.7\}$ and $\{1.1 \leftarrow 1.2, 1.3 \leftarrow 0.7\}$. Note that different total order chains will be generated when using the two different lookahead values.

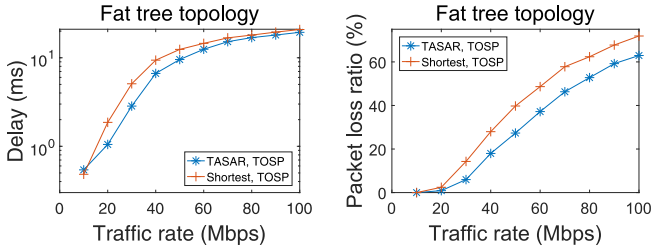
Fig. 7(a) compares the end-to-end delay of the two different lookahead values. We can see that the lookahead value of two achieves shorter delays with a deeper search into the solution space. Similarly, Fig. 7(b) shows that the lookahead value of two achieves lower packet loss ratios.

4) Placing Middleboxes Without Predetermined Path: Finally, we evaluate the Traffic And Space Aware Routing (TASAR) algorithm by comparing it with a hop count based and ECMP (i.e., load-balancing) enabled shortest-path routing algorithm. For the multi-path topology, we choose an 8-pod fat tree with 80 switches and 128 hosts. Each host generates a flow to a random host in other pods. The baseline traffic rate of each flow ranges from 10 to 100 Mbps with a stride of 10 Mpbs. The two sets of candidate middleboxes after conversion are: $\{1.2 \leftarrow 0.7 \leftarrow 1.1 \leftarrow 0.8\}$ and $\{1.3 \leftarrow 0.7 \leftarrow 1.1 \leftarrow 1.2\}$.

We first compare the routing success ratios of the two algorithms. We adjust the number of spaces per switch from 6 to 8, and measure the percentage of flows that can successfully find

TABLE III
FLOW ROUTING SUCCESS RATIO

Spaces per switch	TASAR	Shortest-path routing
6	94.39%	88.66%
7	100%	96.18%
8	100%	100%



(a) End-to-end delay. (b) Packet loss ratio.

Fig. 8. TASAR simulation results.

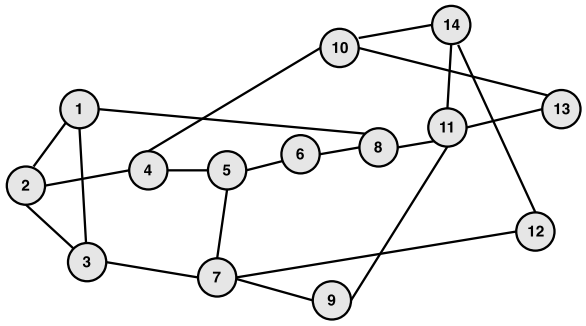


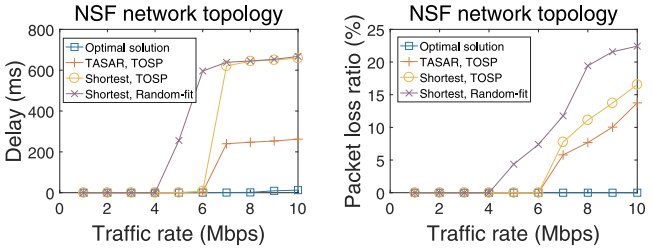
Fig. 9. NSF network topology.

paths with sufficient middlebox spaces. As shown in Table III, when the space number per switch is 6, the routing success ratio of TASAR is 5.7% higher than that of shortest-path routing. When the number increases 7, TASAR achieves a 100% routing success ratio, while shortest-path routing cannot find path for 3.82% of the flows. Finally, when it increases to 8, both algorithms achieve 100% routing success ratios.

Next, we fix the space number per switch to 8, and compare the end-to-end delay and packet loss ratio of the two algorithms. As shown in Fig. 8(a), when the baseline flow rate is 10 Mbps, TASAR has a slightly longer delay, because its paths are not as short as those generated by shortest-path routing. However, once the flow rate increases beyond 10 Mbps, TASAR consistently delivers shorter delays due to its traffic awareness in path selection. Fig. 8(b) also shows that TASAR consistently achieves lower packet loss ratios.

B. Prototype Experiment Results

We have also developed a prototype system and conducted experiments using real traffic and applications. Following the experiment settings in [62], we select the NSF network topology with 14 switches and 21 links, as shown in Fig. 9. Each switch has an associated NFV server with five middlebox spaces, and each link has a bandwidth capacity of 10 Mbps. For traffic generation, we create seven flows from node 1 to 2, 1 to 8, 2 to 7, 7 to 12, 7 to 14, 8 to 14, and 12 to 13. Each



(a) End-to-end delay. (b) Packet loss ratio.

Fig. 10. Prototype experiment results.

flow sends a file of 1 Gbytes, and its initial traffic rate ranges from 1 to 10 Mbps with a stride of 1 Mbps. Each flow needs to go through three middleboxes: a compressor with a traffic changing ratio of 0.8, a firewall of 1, and an encoder of 1.3, as described in Section VI-B. By the least-first-greatest-last rule, the three middleboxes will be arranged in a total-order chain of $\{0.8 \leftarrow 1.0 \leftarrow 1.3\}$. Thus, the aggregate traffic changing ratio of the three middleboxes is $0.8 \cdot 1.0 \cdot 1.3 \approx 1$.

The experiment results are consistent with the simulation ones. In detail, we compare four solutions: shortest-path routing with random-fit placement, shortest-path with TOSP, TASAR with TOSP, and a manually calculated optimal solution. Fig. 10(a) shows that when the traffic rate is equal to or less than 4 Mbps, all the solutions have a short end-to-end delay of less than 1 ms. The delay of shortest-path with random-fit starts increasing significantly at 5 Mbps and is the longest thereafter, because both its routing and placement strategies are not optimized. By changing to a more efficient placement strategy, shortest-path with TOSP postpones the delay spike to 7 Mbps, and by further optimizing the routing strategy, TASAR with TOSP reduces the maximum delay from over 650 to 260 ms. Finally, the optimal solution achieves the shortest delay of no more than 13 ms. Note that the TAPIM problem is NP-hard, and the optimal solution is obtained at the cost of large computation overhead. Fig. 10(b) shows that packet losses happen to shortest-path with random-fit as early as at 5 Mbps. By optimizing middlebox placement, shortest-path with TOSP and TASAR with TOSP postpone packet losses to 7 Mbps, and the latter further reduces the loss ratio with optimized flow routing. The manually calculated optimal solution experiences no packet loss. Although delay spikes and packet losses happen at different rates due to different topologies and experiment settings, the results and those in [63] demonstrate consistency that TASAR outperforms short-test path routing thanks to its traffic awareness.

VIII. Conclusion and Future Work

The advancement of virtualization technology has made NFV a promising platform for network function provisioning. However, the flexibility to run an NFV middlebox on any available standard server also creates a challenge for efficient NFV implementation. In this paper, we have studied the optimal placement of NFV middleboxes by considering different middlebox traffic changing effects and dependency relations. We first formulate the Traffic Aware Placement of Interdependent Middleboxes problem as a graph optimization problem with the objective to load-balance the network. Next, we solve the problem when the flow path is predetermined, and propose optimal algorithms for a non-ordered or totally-ordered middlebox set. For the general scenario of a partially-ordered middlebox set, we show that the problem is NP-hard by reduction from the Clique problem, and propose an efficient heuristic to convert a partially-ordered set to a totally-ordered one. On the other hand, when the flow path is not predetermined, we show that the studied problem is NP-hard even for a non-ordered or totally-ordered middlebox set, and propose the Traffic And Space Aware Routing heuristic. We have conducted large scale simulations to evaluate the proposed solutions, and have also implemented an SDN based prototype to validate them in realistic environments. Extensive simulation and experiment results are presented to demonstrate the effectiveness of our design.

In our future work, we plan to investigate routing of mice flows among already placed middleboxes. On the one hand, the problem context will be different in that middleboxes of different types and capacities have been placed in the network. On the other hand, the same set of constraints including dependency relations and traffic changing effects will still apply. Due to the nature of waypoint routing [30], we expect the general problem to be NP-hard, and efficient approximation is necessary to make a solution practical.

References

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

[2] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proc. USENIX NSDI*, 2012, pp. 323–336.

[3] B. Yi, X. Wang, K. Li, S. K. Das, and M. Huang, "A comprehensive survey of network function virtualization," *Comput. Netw.*, vol. 133, pp. 212–262, Feb. 2018.

[4] *Citrix WAN Optimization With NetScaler SD-WAN*. Accessed: Jul. 31, 2019. [Online]. Available: https://www.citrix.com/content/dam/citrix/en_us/documents/products-solutions/wan-optimization-with-netscaler-sdwan.pdf

[5] M. J. Miller, B. Vucetic, and L. Berry, *Satellite Communications: Mobile and Fixed Services*. New York, NY, USA: Springer, 1993. [Online]. Available: https://books.google.com/books/about/Satellite_Communications.html?id=Ug96CTcvQMkC

[6] S. Mehraghdam, M. Keller, and H. Karl, "Specifying and placing chains of virtual network functions," in *Proc. IEEE Cloud Netw.*, 2014, pp. 7–13.

[7] *Cisco: NAT Order of Operation*. Accessed: Jul. 31, 2019. [Online]. Available: http://www.cisco.com/c/en/us/support/docs/ip/network-address-translation-nat/6209–5.html

[8] *Microsoft TechNet: VPNs and Firewalls*. Accessed: Jul. 31, 2019. [Online]. Available: https://technet.microsoft.com/en-us/library/cc958037.aspx

[9] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *Proc. IEEE INFOCOM*, 2011, pp. 1629–1637.

[10] J. S. Otto, M. A. Sánchez, D. R. Choffnes, F. E. Bustamante, and G. Siganos, "On blind mice and the elephant: Understanding the network impact of a large distributed system," in *Proc. ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, 2011, pp. 110–121.

[11] T. Mori, M. Uchida, R. Kawahara, J. Pan, and S. Goto, "Identifying elephant flows through periodically sampled packets," in *Proc. ACM IMC*, 2004, pp. 115–120.

[12] *Software-Defined Networking: The New Norm for Networks*. Accessed: Jul. 31, 2019. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf

[13] R. Cohen, L. Lewin-Eytan, J. Naor, and D. Raz, "Near optimal placement of virtual network functions," in *Proc. IEEE INFOCOM*, 2015, pp. 1346–1354.

[14] T.-W. Kuo, B.-H. Liou, K. C.-J. Lin, and M.-J. Tsai, "Deploying chains of virtual network functions: On the relation between link and server usage," in *Proc. IEEE INFOCOM*, 2016, pp. 1–9.

[15] Y. Sang, B. Ji, G. R. Gupta, X. Du, and L. Ye, "Provably efficient algorithms for joint placement and allocation of virtual network functions," in *Proc. IEEE INFOCOM*, 2017, pp. 1–9.

[16] H. Feng, J. Llorca, A. M. Tulino, D. Raz, and A. F. Molisch, "Approximation algorithms for the NFV service distribution problem," in *Proc. IEEE INFOCOM*, 2017, pp. 1–9.

[17] T. Wang, H. Xu, and F. Liu, "Multi-resource load balancing for virtual network functions," in *Proc. IEEE ICDCS*, 2017, pp. 1322–1332.

[18] W. Ma, J. Beltran, Z. Pan, D. Pan, and N. Pissinou, "SDN-based traffic aware placement of NFV middleboxes," *IEEE Trans. Netw. Service Manag.*, vol. 14, no. 3, pp. 528–542, Sep. 2017.

[19] X. Fei, F. Liu, H. Xu, and H. Jin, "Adaptive VNF scaling and flow routing with proactive demand prediction," in *Proc. IEEE INFOCOM*, 2018, pp. 486–494.

[20] H. Moens and F. De Turck, "VNF-P: A model for efficient placement of virtualized network functions," in *Proc. IEEE CNSM*, 2014, pp. 418–423.

[21] S. Sahhaf *et al.*, "Network service chaining with optimized network function embedding supporting service decompositions," *Comput. Netw.*, vol. 93, pp. 492–505, Dec. 2015.

[22] H. Moens and F. De Turck, "Customizable function chains: Managing service chain variability in hybrid NFV networks," *IEEE Trans. Netw. Service Manag.*, vol. 13, no. 4, pp. 711–724, Dec. 2016.

[23] Y. Li, L. T. X. Phan, and B. T. Loo, "Network functions virtualization with soft real-time guarantees," in *Proc. IEEE INFOCOM*, 2016, pp. 1–9.

[24] V. Eramo, E. Miucci, M. Ammar, and F. G. Lavacca, "An approach for service function chain routing and virtual function network instance migration in network function virtualization architectures," *IEEE/ACM Trans. Netw.*, vol. 25, no. 4, pp. 2008–2025, Aug. 2017.

[25] M. C. Luizelli, W. L. D. C. Cordeiro, L. S. Buriol, and L. P. Gaspary, "A fix-and-optimize approach for efficient and large scale virtual network function placement and chaining," *Comput. Commun.*, vol. 102, pp. 67–77, Apr. 2017.

[26] Q. Zhang, Y. Xiao, F. Liu, J. C. S. Lui, J. Guo, and T. Wang, "Joint optimization of chain placement and request scheduling for network function virtualization," in *Proc. IEEE ICDCS*, 2017, pp. 731–741.

[27] G. Even, M. Rost, and S. Schmid, "An approximation algorithm for path computation and function placement in SDNs," in *International Colloquium on Structural Information and Communication Complexity*. Cham, Switzerland: Springer Int., 2016. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-48314-6_24

[28] T. Lukovszki and S. Schmid, "Online admission control and embedding of service chains," in *International Colloquium on Structural Information and Communication Complexity*. Cham, Switzerland: Springer Int., 2015, pp. 104–118. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-25258-2_8

[29] T. Lukovszki, M. Rost, and S. Schmid, "Approximate and incremental network function placement," *J. Parallel Distrib. Comput.*, vol. 120, pp. 159–169, Oct. 2018.

[30] S. A. Amiri, K.-T. Foerster, R. Jacob, and S. Schmid, "Charting the algorithmic complexity of waypoint routing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 48, no. 1, pp. 42–48, 2018.

[31] S. A. Amiri, K.-T. Foerster, and S. Schmid, "Walking through waypoints," in *Proc. Latin Amer. Symp. Theor. Informat.*, 2018, pp. 37–51.

[32] N. Mckeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2006.

[33] W. Ding, W. Qi, J. Wang, and B. Chen, "OpenSCaaS: An open service chain as a service platform toward the integration of SDN and NFV," *IEEE Netw.*, vol. 29, no. 3, pp. 30–35, May/Jun. 2015.

[34] J. Matias, J. Garay, N. Toledo, J. Unzilla, and E. Jacob, "Toward an SDN-enabled NFV architecture," *IEEE Commun. Mag.*, vol. 53, no. 4, pp. 187–193, Apr. 2015.

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

[36] Y. Zhang *et al.*, "Steering: A software-defined networking for inline service chaining," in *Proc. IEEE ICNP*, 2013, pp. 1–10.

[37] 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. USENIX NSDI*, 2014, pp. 543–546.

[38] C. Zeng, F. Liu, S. Chen, W. Jiang, and M. Li, "Demystifying the performance interference of co-located virtual network functions," in *Proc. IEEE INFOCOM*, 2018, pp. 765–773.

[39] H. Jin, D. Pan, J. Xu, and N. Pissinou, "Efficient VM placement with multiple deterministic and stochastic resources in data centers," in *Proc. IEEE GLOBECOM*, 2012, pp. 2505–2510.

[40] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. USENIX NSDI*, 2011, pp. 1–14.

[41] A. Singh, M. R. Korupolu, and D. Mohapatra, "Server-storage virtualization: Integration and load balancing in data centers," in *Proc. ACM/IEEE Supercomput.*, 2008, p. 53.

[42] *Cisco EIGRP*. Accessed: Jul. 31, 2019. [Online]. Available: http://www.cisco.com/c/en/us/support/docs/ip/enhanced-interior-gateway-routing-protocol-eigrp/16406-eigrp-toc.html

[43] *OSPF: Frequently Asked Questions*. Accessed: Jul. 31, 2019. [Online]. Available: http://www.cisco.com/c/en/us/support/docs/ip/open-shortest-path-first-ospf/9237-9.html

[44] T. H. Cormen *et al.*, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[45] *Floodlight OpenFlow Controller*. Accessed: Jul. 31, 2019. [Online]. Available: http://www.projectfloodlight.org/floodlight/

[46] *MiniNet: Rapid Prototyping for Software Defined Networks*. Accessed: Jul. 31, 2019. [Online]. Available: http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet

[47] *OpenFlow Switch Specification Version 1.5.0*. Accessed: Jul. 31, 2019. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf

[48] *Open vSwitch*. Accessed: Jul. 31, 2019. [Online]. Available: http://openvswitch.org/

[49] *TCPDUMP/LIBPCAP Public Repository*. Accessed: Jul. 31, 2019. [Online]. Available: http://www.tcpdump.org/

[50] *ZLIB Compression Library*. Accessed: Jul. 31, 2019. [Online]. Available: https://zlib.net/

[51] *Cisco Digital Media Suite: Cisco Digital Media Encoder 1100*. Accessed: Jul. 31, 2019. [Online]. Available: https://www.cisco.com/c/en/us/products/collateral/video/digital-media-encoders/data_sheet_c78–504499.pdf

[52] J. Golston, "Comparing media codecs for video content," in *Proc. Embedded Syst. Conf. San Francisco*, 2004, pp. 1–18.

[53] N. Karpinsky and S. Zhang, "3D range geometry video compression with the H.264 codec," *Opt. Lasers Eng.*, vol. 51, no. 5, pp. 620–625, 2013.

[54] *BCH (48, 36, 5) C Implementation*. Accessed: Jul. 31, 2019. [Online]. Available: http://www.eccpage.com/bch4836.c

[55] Z. H. Kashani and M. Shiva, "BCH coding and multi-hop communication in wireless sensor networks," in *Proc. IEEE IFIP Int. Conf. Wireless Opt. Commun. Netw.*, 2006, pp. 1–5.

[56] M. R. Islam, "Error correction codes in wireless sensor network: An energy aware approach," *Int. J. Comput. Inf. Eng.*, vol. 4, no. 1, pp. 59–64, 2010.

[57] *PLEX Transcoding Media*. Accessed: Jul. 31, 2019. [Online]. Available: https://support.plex.tv/articles/200250377-transcoding-media/

[58] N. Kamaci and Y. Altunbasak, "Performance comparison of the emerging H.264 video coding standard with the existing standards," in *Proc. IEEE Int. Conf. Multimedia Expo*, 2003, pp. 345–348.

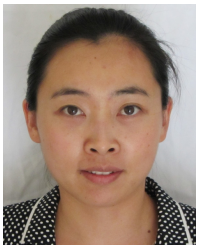
[59] J.-R. Ohm, G. J. Sullivan, H. Schwarz, T. K. Tan, and T. Wiegand, "Comparison of the coding efficiency of video coding standards including high efficiency video coding (HEVC)," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1669–1684, Dec. 2012.

[60] *Project Homepage*. Accessed: Jul. 31, 2019. [Online]. Available: https://users.cs.fiu.edu/ pand/TAPIM/

[61] K. Park and W. Willinger, *Self-Similar Network Traffic and Performance Evaluation*. New York, NY, USA: Wiley, 2000.

[62] T. Lin, Z. Zhou, M. Tornatore, and B. Mukherjee, "Demand-aware network function placement," *J. Lightw. Technol.*, vol. 34, no. 11, pp. 2590–2600, Jun. 1, 2016.

[63] W. Ma, O. Sandoval, J. Beltran, D. Pan, and N. Pissinou, "Traffic aware placement of interdependent NFV middleboxes," in *Proc. IEEE INFOCOM*, 2017, pp. 1–9.



Wenrui Ma received the B.S. degree in computer science from Zhengzhou University, China, in 2007. She is currently pursuing the Ph.D. degree with the School of Computing and Information Sciences, Florida International University. Her research interests include software-defined networking, network virtualization, and data center networking.

Jonathan Beltran received the B.S. degree in computer science from Florida International University in 2017, where he is currently pursuing the Graduation degree with the School of Computing and Information Sciences. His research interest is in computer networks.

Deng Pan received the B.S. and M.S. degrees in computer science from Xi'an Jiaotong University, China, in 1999 and 2002, respectively, and the Ph.D. degree in computer science from Stony Brook University in 2007. He is currently an Associate Professor with the School of Computing and Information Sciences, Florida International University. His research interests include high performance network architecture and network function virtualization.

Niki Pissinou received the B.S. degree in industrial and systems engineering from the Ohio State University, and the M.Sc. degree in computer science from the University of California, Riverside, and the Ph.D. degree from the University of Southern California. She is currently a Professor with the School of Computing and Information Sciences, Florida International University. Her current research interests include high speed networking, insider attacks detection and prevention in mobile ad-hoc networks, and trust, privacy and data cleaning mechanisms in trajectory sensor networks.