Bandwidth Guaranteed Virtual Network Function Placement and Scaling in Datacenter Networks

Fangxin Wang*,Ruilin Ling*, Jing Zhu*, Dan Li*

*Department of Computer Science and Technology, Tsinghua University

***Abstract*—Enterprises deploy their middlebox services in cloud seeking for easy management, flexible scalability and economic savings. However, existing elastic virtual network function(VNF) placement strategy often leads to an unpredictable placing location due to the ever-changing workload, which may waste much precious bandwidth resource and bring a lot of VM operation overhead(*e.g.* VM launch, termination and migration). A key problem for cloud providers is how to conduct an effective service placement and provide resource provision according to various workload, satisfying the bandwidth requirement of each service while saving as much cloud resource as possible.**

In this paper we solve both the virtual network function(VNF) placement and scaling problem based on preplanned allocation with bandwidth guarantee. We first propose a concept of VNF instance communication graph to describe the bandwidth demand of each VNF instance and explore the placement requirement for bandwidth savings. Then we design an on-line heuristic algorithm to achieve approximate optimal allocation. At last, we also provide an off-line optimal solution for comparison. Our simulation shows that our heuristic solution saves 20% more bandwidth resource and reduce more VM migration overhead than existing elastic placement solution. Its performance is also very close to the optimal solution.

I. Introduction

The technique of Network Function Virtualization (NFV) enables network functions(*e.g.* Firewall, Load balancer, IDS, *etc.*) to run on generic computing resources instead of dedicated hardware appliances, which provides great flexibility for network function management. Enterprises or tenants are motivated to deploy their network function service in cloud datacenters because the combination of NFV and cloud computing brings a lot of advantages – reducing operating cost, saving one-off capital expenditures and providing scalable services [1].

In current NFV environment, users are able to construct their network services by composing a set of VNFs in predefined order as service chains. They install specific policies to VNFs [2] to process traffic traversing these chains and accomplish different network functions. For tenants, they only need to specify policies and composition for VNFs without caring about the specific service chain enforcement in physical datacenters. However, it is a key problem for cloud provider to conduct an effective VNF instance placement.

This work is supported by the National Key Basic Research Program of China (973 program) under Grant 2014CB347800, the National Natural Science Foundation of China under Grant No.61170291, No.61432002, the National High-tech R&D Program of China (863 program) under Grant 2013AA013303, and Tsinghua University Initiative Scientific Research Program.

Existing VNF instance placement solution employs elastic allocation [3], *i.e.* cloud system dynamically adjusts VNF instance number and placement to meet the CPU and bandwidth demand of different workload. Although the elasticity brings much convenience, it still faces two problems. First, the elastic placement may waste much computing and bandwidth resource. Since the scaling of VNF instances is determined dynamically based on the temporal workload, the specific placement is unpredictable. A careless placement may cause traffic a much longer traveling distance, which as a result wastes the scarce bandwidth [4] as well as CPU resource. Second, the elastic instance scaling may arouse much VM operation overhead(*e.g.* VM launch, termination, migration and state consistency). It often takes several seconds to conduct a VM migration due to the overhead of consistency maintenance [3], which introduces much extra traffic and increases delays especially when traffic load is heavy. The ever-changing workload in datacenter [5] may lead to frequent VNF instance scaling and migration, which inevitably brings much VM operation overhead.

In contrast to prior solutions, we propose a preplanned VNF placement and scaling scheme for resource saving and overhead reduction. Tenants specify multiple bandwidth requirements between VNFs in their service chains during different periods. We guarantee the required bandwidth and allocate VNF instances into datacenter based on the preplanned bandwidth requirement instead of employing the elastic allocation method. In this paper, we solve both the VNF instance placement and scaling problems utilizing preplanned allocation, which saves much network resource for future tenant acceptance and reduces much VM migration overhead. We mainly have the following contributions:

- We propose the VNF instance communication graph abstraction, where tenants can specify bandwidth reservation between VNFs in a service chain. We also formulate the placement principles for bandwidth saving.
- Building on top of the communication graph and placement principle, we design an on-line heuristic algorithm to allocate VNF instances effectively, achieving minimum overall bandwidth occupancy, VM usage and migration overhead while accepting as many requests as possible.
- We also provide an off-line programming based algorithm to achieve optimal placement, which solves the initial placement problem and scaling problem in a unified way. Our simulation shows that our heuristic algorithm

performs close to the optimal allocation and accepts about 20% more bandwidth requests than state-of-the-art elastic allocating algorithm.

II. Background and Model Statement

In this section, we begin with some backgrounds and the statement of our preplanned model.

A. Background

Service chain. Network function virtualization enables users to flexibly compose rich and custom virtual network functions into a service chain according to their particular demand [6]. Users install predefined rules into each VNF to execute particular function and steer traffic through different VNFs in particular order [2]. Fig.1 shows an example of service chains in which each node represents a VNF. Traffic is divided into two paths by service classifier according their different types. VNFs on each path are composed into a service to accomplish a traffic processing function. For example, the video traffic traverses path 2 for acceleration.

Traffic characteristics. Unlike traditional cloud computing, VNF resource consumption can be quite diverse and workload dependent [7]. We explore the characteristics of middlebox service workload by observing a real traffic trace [8] of an enterprise cluster, which captured the total network traffic across the enterprise Big Marketing for one week. According to our investigation shown in Fig.2, middlebox services may encounter various traffic workloads during different periods, and sometimes the workloads are regular and periodic(*e.g.* a high bandwidth requirement at peak time while a low requirement at trough time for several days).

VNF placement. Since VNFs are running on generic servers whose traffic processing ability is limited, we usually need several VNF instances to undertake one virtual function cooperatively, especially when workload is heavy. We assume all the servers in datacenters are homogeneous with the same number of identical VM slots and each VM slot can hold only one VNF instance. So what we need is to determine the placement location for these VNF instances in datacenters. An effective placement should satisfy two basic requirement: (1) all VNF instances are assigned into available VM slots and (2) the bandwidth requirement of each physical link does not exceed its capacity.

Existing elastic placement. Existing VNF placement mostly employ an elastic allocation strategy that VNF instances elastically increase or decrease to satisfy the processing demand for different workload. Elastic solutions such as stratos [3] dynamically monitor the computing status of each VNF instance and overall network status, and detect the source of bottleneck. If there is a computing shortage, it adds VNF instances to share the heavy workload. And when link congestion occurs, the affected instances will migrate to other location with enough bandwidth.

B. Model Statement of Preplanned Placement

Elastic placement is able to dynamically adjust the scale of VNF instances, but the elasticity always comes with much

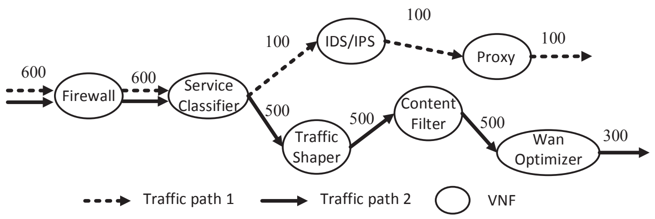


Fig. 1. A case of NF service chains.

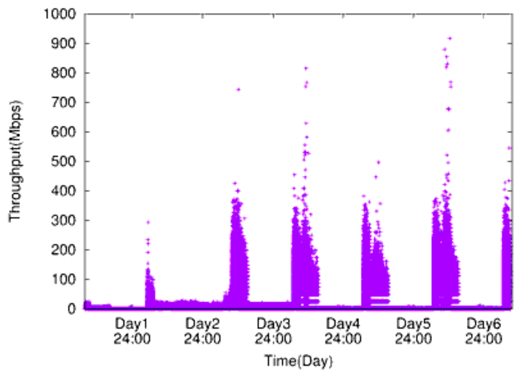


Fig. 2. A one week real traffic trace from Big Marketing company that records the total traffic volume traversing its computer clusters.

resource wastage and migration overhead due to a lack of global information for a optimized placement. In contrast to elastic placement, we propose preplanned VNF instance placement strategy.

Bandwidth requirement and guarantee. Beside the service chain composition and rules for each VNF, users are required to specify the bandwidth demand between each upstream and downstream VNFs. As illustrated in Fig.1, each value represents the bandwidth requirement between VNFs. We provide bandwidth guarantee and conduct the VNF instance placement according to the bandwidth requirement. When a service chain request arrives, we judge whether the residual cloud resource is able to satisfy the computing and bandwidth demand. If so, we accept it and conduct a VNF instance placement, otherwise we reject this request. We will discuss the detail in §III.

In traditional cloud computing, tenants have to subscribe for bandwidth based on the largest possible traffic load. However, it is too stiff to restrict unchanged bandwidth reservation since tenants are not responsible to pay for their unnecessary bandwidth during trough time. Generally, considering the regular and periodic features of workload, we allow tenants to specify multiple bandwidth requirement on different time periods according to their business. It not only helps tenants to reduce their capital cost but also avoids much network resource waste. Taking the case in Fig.2 for example, this enterprise can specify 500Mbps and 100Mbps for the busy period and idle period, respectively. We accept a service request only when we can deploy it under the maximum bandwidth requirement.

Benefits. Based on the knowledge of service chain composition and bandwidth requirement provided by tenants, we can conduct a more effective preplanned VNF instance placement by fully considering the characteristics of the service chain and current datacenter situation. Compared with existing elastic placement, the preplanned placement have a lot of benefits as follows.

- *Saving bandwidth and VM slot resource by colocation.* Our preplanned placement strategy tries to place VNF instances in a localized way by searching a minimum feasible subtree in datacenter. By this way we can save much bandwidth resource, especially the precious core-level bandwidth.
- *Reducing overhead for frequent VM migrations.* For a particular bandwidth requirement, we employ an one-off VNF instances placement. When bandwidth requirement changes, we try to conduct an incremental deployment with little affect on the existing placement. By this way we can largely reduce the times of VM migration, which reduces migration overhead as a result.

III. Design for VM placement and Scaling

In this section, we introduce our solutions to allocate VNFs into physical datacenters. Given that current datacenters are mostly organized in oversubscribed tree-like topology[4], we consider three-layer single root tree-shaped datacenters for simplicity, which is easy to expanded to multi-root topology. The variables in this section are explained in Table.I.

TABLE I
Notations

t	A particular subtree for VNF instance placement
G	The VNF instance communication graph
Ub_i	Maximum traffic that individual instance of VNF i can support
N_i	The instance number of VNF i
b_i	$b^{in}_{u\to i}$ and $b^{out}_{i\to v}$ represent incoming and outgoing bandwidth of each instance of VNF i from VNF u and to VNF v, respectively. b^{in}_i=$b^{in}_{all\to i}$ and b^{out}_i=$b^{out}_{i\to all}$, b_i=b^{in}_i+b^{out}_i
$B_{u\to v}$	The required bandwidth for communication between VNF u and VNF v.
B_i	The B^{in}_i and B^{out}_i are total incoming and outgoing bandwidth for VNF i, B_i=B^{in}_i+B^{out}_i
γ_i	γ_i means the gain/drop factor of VNF i, *i.e.* the change ratio for ingress-to-egress traffic, γ_i=B^{in}_i/B^{out}_i
$T^t_{u\to v}$	The internal bandwidth from VNF u to VNF v in subtree t
T^t	The total internal bandwidth in subtree t
$C_{i,t}$	The $C^{in}_{i,t}$ and $C^{out}_{i,t}$ are incoming and outgoing bandwidth for instances of VNF i in subtree t respectively.
C_t	The total bandwidth for all instances in subtree t, also include C^{in}_t and C^{out}_t, C_t=C^{in}_t+C^{out}_t
S_t	A set of selected instances for placement in subtree t
$Cost_t$	We assign different costs for links in different layer, *e.g.* the bandwidth of upper layer is more scarce and precious, so it deserves higher cost.
α	The weight for VM slot utilization
β	The weight for overhead of VM migration
$X_{s,i}$	$X^{in}_{s,i}$ represents the incoming bandwidth of VNF i on VM slot s, and $X^{out}_{s,i}$ represents the outgoing bandwidth. If $X_{s,i}$=0, it means slot s doesn't hold instance of VNF i.
M	The migration times for VM scaling.

A. VNF instance communication graph and Bandwidth Saving Principles

VNF instance communication graph. For each VNF in the service chain, we need to provide enough capacity to process the traffic traversing this chain. We assume each instance of VNF i has a maximal traffic processing capacity Ub_i, which can be easily obtained by running a pressure test. So the minimal number of instances of VNF i is:

$$N_i = \lceil B^{in}_i / Ub_i \rceil. \tag{1}$$

By this way we can get the least necessary number of instances for all VNFs. Meanwhile, the required bandwidth for each instance is equally shared with a load balance strategy.

$$b^{in}_i = B^{in}_i / N_i; \quad b^{out}_i = B^{out}_i / N_i; \tag{2}$$

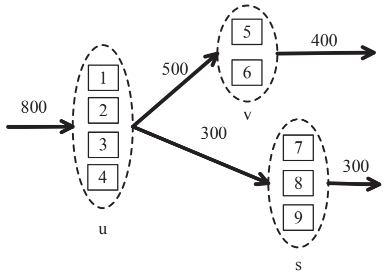


Fig. 3. An example of VNF instance communication graph.

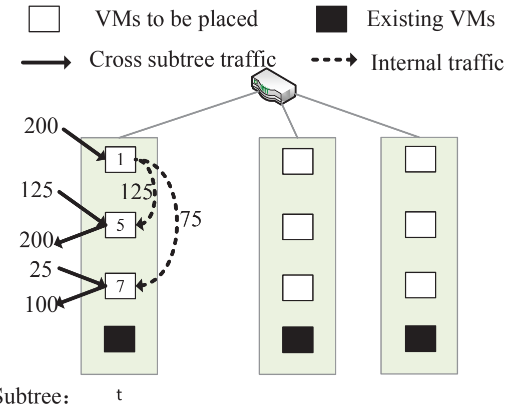


Fig. 4. A case of VM placement as well as the bandwidth requirement. Assume that VNF instance 1, 5, 7 are placed in subtree t, the internal bandwidth and crossing subtree bandwidth are illustrated.

Thus, we can construct a *VNF instance communication graph* composed of VNF instances and the bandwidth requirement between them. Fig.3 describes an example of communication graph based on a service chain with bandwidth requirement labeled in it. Since Ub of each VNF is 200, 250 and 100(in Mbps), the least instance number for each VNF is 4, 2 and 3. We can also calculate bandwidth requirement of each instance by equation (2). We notice that some network functions may change the aggregate traffic volume through them(*e.g.* Wan optimizer may compress traffic for acceleration). So we define the traffic change ratio as γ.

Placement principles for bandwidth saving. The placement procedure is actually allocating VNF instances into each subtree of datacenter recursively until all VNF instances are put into VM slots. For each subtree that holds a set of VNF instances, we need to reserve internal bandwidth and cross subtree bandwidth.

Principle 1: A VNF instance prefers to steel the traffic to its downstream VNF instance inside the same subtree rather than to those outside the subtree.

This principle is intuitive and easy to realize. We can steer the incoming flows through SDN to achieve this goal, which is our ongoing work. We assume some instances of VNF u and VNF v are placed in subtree t(u is in upstream position and v is in downstream position in communication graph). Based on *principle 1*, we can get the internal bandwidth requirement:

$$T^t_{u \to v} = min(N^t_u b^{out}_{u \to v}, N^t_v b^{in}_{u \to v}); \tag{3}$$

Besides, instances of VNF i have to send or receive traffic from other VMs outside subtree t when internal traffic can not totally satisfy bandwidth demand. We can get the incoming and outgoing traffic of VNF i across subtree t:

$$C^{in}_{i,t} = N^t_i b^{in}_i - \sum_{u \in t} T^t_{u \to i};\ C^{out}_{i,t} = N^t_i b^{out}_i - \sum_{v \in t} T^t_{i \to v}; \tag{4}$$

Thus, we can get the total traffic across subtree and the total internal traffic by adding all of them.

$$C_t = C^{in}_t + C^{out}_t = \sum_{i \in t} C^{in}_{i,t} + \sum_{i \in t} C^{out}_{i,t}; \tag{5}$$

$$T^t = \sum_{u \in t} \sum_{v \in t} T^t_{u \to v}; \tag{6}$$

By adding all the cross subtree bandwidth in one layer l in datacenter, we can get the following formula.

$$\begin{aligned} \sum_{t \in l} C_t &= \sum_{t \in l} \sum_{i \in t} (N^t_i b^{in}_i - \sum_{u \in t} T^t_{u \to i} + N^t_i b^{out}_i - \sum_{v \in t} T^t_{i \to v}); \\ &= \sum_{i \in all} N_i b_i - \sum_{t \in l} 2T^t; \end{aligned} \tag{7}$$

From this formula we know that $\sum_t (C_t + 2T^t)$ is a constant,*i.e.* the sum of cross subtree bandwidth and two times of internal bandwidth of each layer is fixed. So if we want to minimize the overall link bandwidth consumption, we should try to minimize the cross subtree bandwidth and maximize internal bandwidth for each subtree. Thus the localized placement is an effective placement strategy. Then we get the following greedy placement principle.

Principle 2: For each placement in a subtree, we try to place as many VNF instance as possible while minimizing C_t and maximizing T^t.

Fig.4 describes an example of VNF instance placement according to the service chain in Fig.3. We assume that VM 1, 5 and 7 are placed in subtree t, while other VMs are placed outside t. In this example, we get C^{in}_t=350, C^{out}_t=300 and T^t=200, respectively.

B. Heuristic Algorithm

Building on top of the VNF instance communication graph and bandwidth saving principles above, we design an on-line heuristic algorithm to achieve approximate optimal placement. Our heuristic algorithm tries to place VNF instances in a localized way recursively to save as much network resource as possible.

Algorithm 1: Handling request algorithm

```
Input: G:communication graph, T:tree topology
Output: Request acceptance or rejection.
1  while true do
2      l = 0;
3      while $l \le height(T)$ do
4          foreach subtree t at level l do
5              if $N \le slot(t)$ then
6                  put all VMs into $S_t$;
7                  $\Phi$ = $Alloc(G, S_t, t)$;
8                  if $\Phi.flag$ == true then
9                      return true;
10         l = l + 1;
11     if $\Phi.flag$ == false then
12         G=Scale(G);
13     if the utilization of every instance is below $\varepsilon$ then
14         break;
15 return false;
```

Algorithm.1 describes the overall process to handle tenant request. We traverse the datacenter topology from the lowest level(level 0, physical servers) up to the root level(level 3)(line 2-3). For each subtree t, we first judge whether it is likely to hold all the VNF instances by calculating the number of available VM slots within it(line 4-5). If there are enough slots, then instances are put into a set S_t and *Alloc(·)* is invoked to try allocating the request into subtree t. Since bandwidth resource inside t may be insufficient for the bandwidth demand, we try another subtree once this trial fails. If all the possible subtrees are unable to hold this request, we try to scale out the communication graph by adding one instance to VNF i which has the largest aggregate bandwidth requirement(*i.e.* largest b_i) according to formula (2)(line 12). By this way we narrow the gap of bandwidth discrepancy among different VNF instances and avoid the bandwidth bottleneck, which also increases the possibility for request acceptance. However, we can not keep on splitting VNF instances forever, because more slot usage means more capital cost. Thus, we set a lowest instance utilization threshold as ε, representing the ratio of actual processing traffic to the maximum processing capacity. Once utilization of every VNF instances is below ε, we stop splitting and reject this tenant request(line 13-14). The value of ε can be flexibly determined by different accounting strategy. By traversing the datacenter network topology in a bottom-up manner, our algorithm allocates VNFs into the lowest feasible subtree in order to save precious core level bandwidth.

Algorithm.2 (*Alloc(·)*) shows each attempt to allocate the selected instance set S_t in graph G into the given subtree t. It is a trivial problem if we are trying to allocate S_t into a lowest level subtree (physical servers) with enough VM slots(line 1).

Algorithm 2: Alloc algorithm

```
Input: $G$:communication graph, $S_t$:instance set for $t$
       $t$:subtree for allocation
Output: $\Phi$:the VM allocation and bandwidth reservation
1  if $level(t)$==$0$ then
2      map $S_t$ into $\Phi^t$; $\Phi_t.flag$ == true;
3  else
4      sort subtrees of $t$ in descending order of available
       resource;
5      foreach subtree $v$ of $t$ do
6          $cnt$ is assigned as the smaller value between
           unused VM numbers in $S_t$ and $slot(v)$;
7          while $cnt--$ do
8              $S_v$ = $\varnothing$;
9              First select the unused VM of the largest
               aggregate bandwidth into $S_v$;
10             while $|S_v| \leq cnt$ and $C_v^{in} \leq bw^{in}(v)$ and
               $C_v^{out} \leq bw^{out}(v)$ do
11                 $S_v$ += $SelectVM(G, S_t)$;
12             $\Phi_v$ = Alloc$(G, S_v, v)$;
13             if $\Phi_v.flag$ == true then
14                 reserve bandwidth for $t$;
15                 $\Phi_t$ += $\Phi_v$;
16                 mark VMs in $S_v$ as used;break;
17     if not all VMs in $S_t$ are used then
18         DeAlloc($\Phi_t$);
19 return $\Phi_t$;
```

The algorithm just puts instances into the server and returns success(line 2). Note that communication between slots in the same physical server does not occupy link bandwidth and enjoys plenty of network bandwidth resources. Otherwise, if t consists of multiple subtrees, the algorithm attempts to split instances into as few subtrees as possible by colocation.

We sort the subtrees of t in a descending order according to their available slots as well as bandwidth(line 4). For each subtree v of t, we tries to allocate as much instances as possible. We first decide the upper bound of VNF instance numbers that subtree v can hold as cnt, which is initialized as the smaller one between the residual VM slots of v and the unused VM numbers in S_t(line 6). For each trial, we first pick up an unused instance with the largest aggregate bandwidth requirement and put it into S_v(line 9), because larger subtree is more likely to accept instance with larger aggregate bandwidth demand. Then we repeatedly pick up one VNF instance into S_v until the aggregation bandwidth across the subtree or slot usage exceeds the constraint(line 10-11). For each *SelectVM*, we refer to principle 2 in §III-A. We invoke *Alloc(·)* in a recursive way until all instances are placed in physical servers(line 12). If the allocation for v is successful, we reserve bandwidth for the link of subtree v and record the placement in Φ(line 14-15). Besides, we mark those VMs in S_v as used and quit this process(line 16). Otherwise, we decrease the upper bound cnt and begin another trial(line 7). If VMs in S_t are not completely allocated after trying all its subtrees, *DeAlloc* is invoked to release the reserved slots and bandwidth resources(line 18). At last, we return the resource allocation mapping Φ(line 19).

TABLE II
STEPS AND BANDWIDTH REQUIREMENTS TO PLACE THE COMMUNICATION GRAPH INTO A GIVEN SUBTREE.

Step	VM No.	C_t^{in}	C_t^{out}	T^t	can hold?
1	5	250	200	0	√
2	1	325	275	125	√
3	7	350	300	200	√
4	2	400	350	350	√
5	8	450	400	400	√
6	9	550	500	400	×

We explain this heuristic approach utilizing the communication graph example in Fig.3. Notice that VNF u has two downstream VNFs, and the ratio of traffic distribution is 5:3. So $b_{u\to v}^{out}$=125Mbps and $b_{u\to s}^{out}$=75Mbps respectively. VNF v is able to compress traffic and γ_v is 80%. Assuming that we are considering putting this communicate graph into subtree t with 6 slots and 500Mbps bisectional bandwidth. Table.II lists the specific steps as well as the bandwidth requirement of each step. We can put five VNF instances into this subtree, while the placement of the 6-th instance will exceed the bandwidth constraints.

Our algorithm can also handle the scaling problem when bandwidth requirement changes. We provide two alternatives for VNF scaling. The first one is *incremental* deployment. When demand increases, we treat the extra part of traffic as another service chain. So we just execute resource allocation for this new service chain without affecting existing deployment. Similarly, we just remove those newly increased part when bandwidth demand falls. The second alternative is *unified* deployment, where we treat this service as an entirety and reallocate all the traffic by invoking the heuristics again. However, we still prefer to search subtrees holding old VNF placement and try to place relevant VNF instances at their original place. If fails, we then search other subtrees for any possible allocation. We mainly consider to employ the *incremental* deployment due to the high migration overhead, while the *unified* deployment is also acceptable when traffic load is very light.

C. Programming Solution

We then provide an off-line programming based algorithm for comparison, which achieves optimal placements. For each placement, our programming algorithm seeks to minimize (1) the network wide bandwidth utilization of all links and (2) the overall VM slot usage. By this way we can maximize the available system capacity, which is important for scaling and subsequent tenant acceptance.

In most situation we expect a load balance strategy among instances of the same VNF, but sometimes we can loose this restriction by distributing traffic to those instances asymmetrically, which may save bandwidth as a result. For example, two VNF instances with Ub=500Mbps need to process 900Mpbs cooperatively. We can distribute 450Mbps traffic to each of them, while we can also assign traffic as 500Mbps and 400Mbps respectively. Even we can add an instance and allocate 300Mbps for each of them if needed. In our programming solution, we mainly consider the loose condition(*i.e.* without load balance).

Based on the condition above, we formulate the programming model for VNF instance placement as follows:

$$Min: \quad \sum_t ((C_t^{in} + C_t^{out}) * Cost_t) + \alpha * Slot \tag{8}$$

$s.t.$

$$\forall s, \ \sum_{i \in G} X_{s,i}^{in} = \max_{i \in G}\{X_{s,i}^{in}\} \tag{9}$$

$$\forall s, i, \ 0 \leq X_{s,i}^{in} \leq Ub_i \tag{10}$$

$$\forall s, i, \ X_{s,i}^{out} = X_{s,i}^{in} * \gamma_i \tag{11}$$

$$\forall i, \ B_i^{out} = B_i^{in} * \gamma_i \tag{12}$$

$$\forall i, \ B_i^{in} = \sum_{u \in upstream(i)} B_u^{out} \tag{13}$$

$$\forall i, \ \sum_{s \in S} X_{s,i}^{in} = B_i^{in} \tag{14}$$

$$\forall u, v, \ T_{u \to v}^t = \min\{\sum_{s \in t} X_{s,u}^{out}, \sum_{s \in t} X_{s,v}^{in}\} \tag{15}$$

$$\forall i, u, \ C_t^{in} = \sum_{i \in t}\{\sum_{s \in t} X_{s,i}^{in} - \sum_{u \in t} T_{u \to i}^t\} \tag{16}$$

$$\forall i, u, \ C_t^{out} = \sum_{i \in t}\{\sum_{s \in t} X_{s,i}^{out} - \sum_{v \in t} T_{i \to v}^t\} \tag{17}$$

The instance placement result is stored in X. We try to minimize the both the bandwidth and VM slot usage in (8). The VM slot usage is easily to get by counting the non-zero $X_{s,i}^{in}$. We define the weight of VM slot usage as α, which is used to adjust the importance of the two parts. Equation (9) guarantees there is at most one VNF instance existing in one slot at the same time. Equation (10) restricts the maximum traffic each VNF instance can hold. Considering that workload may change after going through some kinds of VNFs such as wan optimizer, equation (12) reflects the ratio of ingress-to-egress traffic. We guarantee that all the traffic for one VNF is processed by (14). Equation (15) calculates the internal traffic volume between VNF u and VNF v. Equation (16) gets the total traffic coming into this subtree, and so is (17) for outgoing traffic.

Scaling. A key consideration for VNF placement is how and where to scale instances when traffic demand changes. Our programming algorithm is also suitable to solve the VM scaling problem. We just need to modify the input bandwidth demand and invoke the programming algorithm again.

When considering scaling in or scaling out, we can add the *VM migration times* as an optional optimizing objective. We modify the original objective as follows:

$$Min: \quad \sum_t ((C_t^{in} + C_t^{out}) * Cost_t) + \alpha * Slot + \beta * M \tag{18}$$

M represents the migration times and β is the weight for the overhead of migration. M can be easily obtained by comparing the new placement with the old placement. β can be flexibly specified according to the actual system overhead, which may vary in different situations. For example when traffic volume is very low and flow numbers is very small, the migration overhead can be small. But it may become a heavy burden when load is heavy or traffic congestion exists.

Besides, we have another option for load balance among VMs belonging to the same VNF. We just need to add a constraint that for each particular VNF i, every $X_{s,i}^{in}$ must be zero or the same non-zero value. Tenants are able to make their own choices by need.

IV. Evaluation

In this section, we carry out a group of simulation based on java and CPLEX [9] to evaluate the performance of our heuristic and programming algorithm for resource allocation. We consider the elastic placement strategy in Stratos [3] as a baseline.

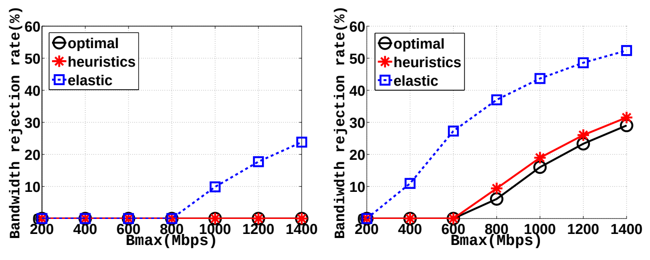
A. Simulation setup

Topology setting. We build a three-layer tree-shaped network topology, *i.e.* the edge switches layer, the aggregate switches layer and the core switch layer, similar to real datacenter topology[4]. The core switch is connected to 8 aggregate switches and each aggregate switch has 16 edge switches. Every rack contains 16 identical servers with the same computing/storage resource, which we divide into 8 VM slots. That is a total of 2048 servers with 16,384 VM slots(denoted as V_{total}). The link capacity of physical server to switch is 1 Gbps. We set the oversubscription ratio between each layer as 2:1.

Tenant Request synthesis. We randomly synthesize tenant requests in a customized way based on our knowledge of common service chain compositions. The length of every chain is distributed equally between 3 to 6, which randomly has one single path or is divided into two paths. The maximal capacity of each kind of VNF instance is random defined between 100 to 300. For simplicity, we assume each tenant only request a minimum and a maximum incoming bandwidth requirement. The bandwidth requirement follows normal distribution, with the mean value of the maximum bandwidth requirement as B_{max} and variation equal to $B_{max}/3$. Thus, we can get the least necessary VNF instance number under maximum bandwidth requirement, denoted as N_l. We assume traffic change ratio for each VNF is 1.

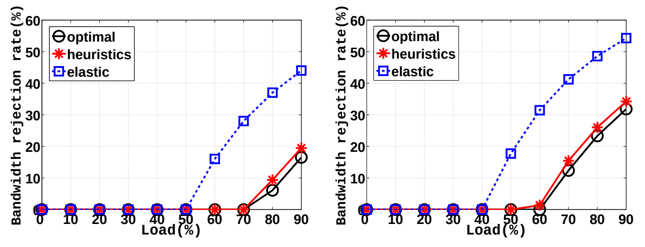
Tenant arrival and departure. In our simulation, we assume the tenant will arrive and leave, which follows a Poisson process [10]. The average arrival rate of tenant is λ and each tenant has a resident time as T for average. We suppose that each tenant request consists of at least N_l VMs. Then we define *load*=$\frac{\lambda N_l T}{V_{total}}$ to describe the load of datacenter excluding the influence of request size.

Besides, we set ε in heuristics as 30% and α in programming objective as the value of Ub_i for each kind of VNF. We set $Cost_i$ of each link as the oversubscription ratio of its layer.



(a) Load=50% (b) Load=80%

Fig. 5. The bandwidth rejection ratio for different Bmax. The Y-axis represents the ratio of the total bandwidth of rejected requests to the total bandwidth of all coming requests.



(a) Bmax=800Mbps (b) Bmax=1200Mbps

Fig. 6. The bandwidth rejection ratio under different load. The Y-axis represents the ratio of the total bandwidth of rejected requests to the total bandwidth of all coming requests.

B. Simulation Result

We first evaluate the impact of average Bmax and different load on tenant acceptance, which is mainly represented by bandwidth rejection ratio. We consider the allocation of requests with Bmax. In Fig.5, we vary the Bmax from 200(in Mbps) to 1400 to evaluate how requests with different bandwidth affect the bandwidth rejection ratio. We find that under 50% workload in Fig.5(a), our *heuristics* algorithm is able to accept all the tenant request with various Bmax request, while rejection ratio of *elastic* reaches more than 20% at a large Bmax. When the load is 80% in Fig.5(b), all three algorithms begins to reject tenants. *heuristics* can accept all request when Bmax is small, while *elastic* still rejects requests under small Bmax(400 and 600). We can accept about 20% bandwidth than *stratos* even under high Bmax requests. These results reflect that our preplanned placement scheme can better utilize the network resource and spare more bandwidth for future request acceptance. Besides, these figures indicate that *heuristics* achieves a approximate optimal result, since it is very close to the *optimal* programming solution.

Similarly, we vary the load from 0 to 90% to study the impact of diverse load under different Bmax in Fig.6. With a relatively small bandwidth requirement(Bmax=800Mbps in Fig.6(a)), *heuristics* can always control their rejection ratio at a low level, while the rejection ratio of *elastic* increases sharply when workload exceeds 50%. When it comes to a large Bmax(1200Mbps in Fig.6(b)), our algorithms accept 20% bandwidth request than *elastic*, which even rejects more than half requests under 90% load.

We then evaluate the VM slot utilization of different algorithms. We consider various objectives for the programming algorithm, *optimal_b* and *optimal_bv*, each minimizing only bandwidth and both bandwidth and VM slot, respectively. Fig.7 describes the VM slot utilization under different load. When traffic load is low, all the four solutions performs the same, which do not bring much VNF scaling since the slope of all lines is nearly one. As workload increases, *optimal_b* increases the fastest due to its ignorance of VM slot usage. *optimal_bv* and *heuristics* have a lower VM utilization than *elastic* under middle workload but a higher utilization under high workload. This is because both *heuristics* and *optimal_bv* avoid unnecessary scaling under the middle workload and thus can accept more VMs under the high workload. When the load is nearly full, our algorithms can achieve more than 95% VM utilization, while *elastic* can only utilize 85% VM resource, leaving the datacenter full of fragments.

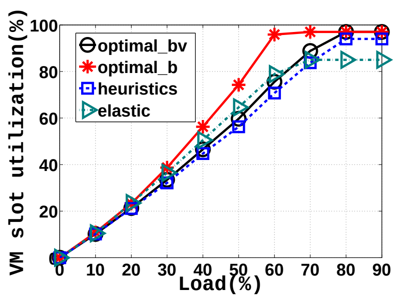


Fig. 7. Average VM slot utilization of different algorithms under various load.

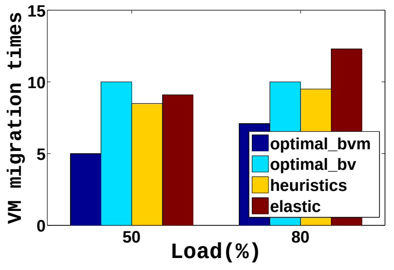


Fig. 8. Average VM migration times for each algorithms.

We also evaluate the impact of different algorithms on the VM migration when bandwidth demand changes. We assume each request has a minimum bandwidth and a maximum bandwidth, with the maximum value as twice of the minimum value. The request will increase from the minimum to the maximum, each of which lasts the same time. Fig.8 shows the number of migration times of each algorithm under different load when Bmin=400Mbps and Bmax=800Mbps. *optimal_bvm* brings fewer VM migration than *optimal_bv* since it takes migration overhead into consideration. *Heuristics* prefers to employ the *incremental* deployment and it still migrate some existing instances. *Elastic* even migrates newly added VMs under high load. This is because when network resource is scarce(load=80%), it has to migrate frequently once it detects a bottleneck due to its per-instance scaling strategy.

V. Related Work

Integrating VNFs. To enforce diverse virtual network functions in generic software defined platforms, previous works seek to integrate VNFs by novel architectures. CoMb [11] exploits middlebox structure to consolidate heterogeneous middlebox applications onto commodity hardware, but does not address the issue of placement and scaling. xMob [12] is a framework for programmable middleboxes using commodity servers. This novel architectures provide implementing basis for VNFs.

VNF control plane design. A lot of works have focused on NFV control plane and seek solutions for better management. OpenNF [13] utilizes central controller to maintain network wide state consistency on VNF instance migration. [14] studies the stability of middlebox states and presents a dynamically elastic virtual network function provision system. SIMPLE and FlowTags [15], [16] consider the traffic modification and integrate an effective flow steering and management by using tags.

VM placement and traffic steering. Proper VM placement and traffic steering help to save network resource wastage and improve efficiency. StEERING [17] and [18] both consider VNF placement and steer traffic through network services in order, while neither of them have bandwidth guarantee. [19] only focuses on initial placement by minimizing VM communication distance and setup cost, which ignores the VNF scaling problem. Stratos [3] builds up a network-aware orchestration layer for virtual middleboxes. It considers a native rack aware heuristic VM placement, while its elastic placement strategy consumes much bandwidth resource and brings a lot of migration overhead.

bandwidth guaranteed virtual network abstraction. To provide predictable network application performance, many previous works have proposed novel virtual network abstractions to specify bandwidth requirement among VMs. Oktopus [20] and [21] consider to solve homogeneous and heterogeneous hose models for VM communication, while [22] and [23] mainly focus on pipe models requiring tenant to specify more fine grained bandwidth between every VM pairs. CloudMirror [24] proposes an application driven network abstractions, guaranteeing bandwidth components. Our work is related to these network abstraction models.

VI. Conclusion and Future Work

The combination of NFV and cloud computing introduces a lot of benefits for enterprise to deploy their middlebox services in cloud, while the elastic VNF allocation strategy often leads to an ineffective instance placement. We propose a novel preplan allocating solution according to the bandwidth requirement between each VNF provided by users, which includes an abstraction of VNF instance communication graph, an on-line heuristic algorithm and an off-line optimal programming model. Our solution saves as much network resource as possible and reduces VM migration overhead.

Our ongoing work is to explore flow schedule strategy using SDN technique to flexibly steer traffic according to our bandwidth restriction. Besides, we seek to implement our placement approach in practical system, such as OpebMano[25], OPNFV[26], OpenStack Nova[27].

References

[1] J. Sherry *et al.*, "Making middleboxes someone else's problem: network processing as a cloud service," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012.

[2] P. Quinn *et al.*, "Network service chaining problem statement," *Internet Engineering Task Force, Internet-Draft draft-quinn-nsc-problem-statement-03*, 2013.

[3] A. Gember *et al.*, "Stratos: A network-aware orchestration layer for virtual middleboxes in clouds," *arXiv preprint arXiv:1305.0209*, 2013.

[4] N. Farrington and A. Andreyev, "Facebook's data center network architecture," in *IEEE Optical Interconnects Conf.* Citeseer, 2013.

[5] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010, pp. 267–280.

[6] "NFVWhitePaper," http://www.etsi.org/technologies-clusters/technologies/nfv.

[7] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica, "Multi-resource scheduling for packet processing," in *Proc. SIGCOMM*, 2012.

[8] "Vast Challenge." http://vacommunity.org/VAST+Challenge+2013%3A+Mini-Challenge+3.

[9] "CPLEX," http:www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

[10] Z. Guo *et al.*, "Improving the performance of load balancing in software-defined networks through load variance-based synchronization," *Computer Networks*, vol. 68, pp. 95–109, 2014.

[11] V. Sekar and e. a. Egi, Norbert, "Design and implementation of a consolidated middlebox architecture," in *NSDI*, 2012, pp. 24–24.

[12] J. W. Anderson *et al.*, "xomb: extensible open middleboxes with commodity servers," in *ANCS*. ACM, 2012, pp. 49–60.

[13] A. Gember-Jacobson *et al.*, "Opennf: Enabling innovation in network function control," in *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 2014, pp. 163–174.

[14] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes." in *NSDI*, 2013, pp. 227–240.

[15] Z. A. Qazi *et al.*, "Simple-fying middlebox policy enforcement using sdn," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. ACM, 2013, pp. 27–38.

[16] S. K. Fayazbakhsh *et al.*, "Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags," in *Proc. USENIX NSDI*, 2014.

[17] Y. Zhang, N. Beheshti *et al.*, "Steering: A software-defined networking for inline service chaining," in *Network Protocols (ICNP), 2013 21st IEEE International Conference on*. IEEE, 2013, pp. 1–10.

[18] A. Mohammadkhan, S. Ghapani, G. Liu, W. Zhang, K. Ramakrishnan, and T. Wood, "Virtual function placement and traffic steering in flexible and dynamic software defined networks," *Mij*, vol. 101, p. 1.

[19] L.-E. Liane, S. N. Joseph, C. Rami, and R. Danny, "Near optimal placement of virtual network functions," in *Proc. IEEE INFOCOM*, 2015.

[20] H. Ballani *et al.*, "Towards predictable datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 242–253.

[21] J. Zhu, D. Li *et al.*, "Towards bandwidth guarantee in multi-tenancy cloud computing networks," in *Network Protocols (ICNP), 2012 20th IEEE International Conference on*. IEEE, 2012, pp. 1–10.

[22] C. Guo *et al.*, "Secondnet: a data center network virtualization architecture with bandwidth guarantees," in *Proceedings of the 6th International COnference*. ACM, 2010, p. 15.

[23] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *INFOCOM, 2010 Proceedings IEEE*. IEEE, 2010, pp. 1–9.

[24] J. Lee *et al.*, "Application-driven bandwidth guarantees in datacenters," in *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 2014, pp. 467–478.

[25] "OpenMano," https://github.com/nfvlabs/openmano.

[26] "OPNFV," https://www.opnfv.org/.

[27] "Nova," https://wiki.openstack.org/wiki/Nova.