Piecing Together the NFV Provisioning Puzzle: Efficient Placement and Chaining of Virtual Network Functions

Marcelo Caggiani Luizelli, Leonardo Richter Bays, Luciana Salete Buriol
Marinho Pilla Barcellos, Luciano Paschoal Gaspary
Institute of Informatics – Federal University of Rio Grande do Sul (UFRGS)
{mcluizelli,lrbays,buriol,marinho,paschoal}@inf.ufrgs.br

***Abstract*—Network Function Virtualization (NFV) is a promising network architecture concept, in which virtualization technologies are employed to manage networking functions via software as opposed to having to rely on hardware to handle these functions. By shifting dedicated, hardware-based network function processing to software running on commoditized hardware, NFV has the potential to make the provisioning of network functions more flexible and cost-effective, to mention just a few anticipated benefits. Despite consistent initial efforts to make NFV a reality, little has been done towards efficiently placing virtual network functions and deploying service function chains (SFC). With respect to this particular research problem, it is important to make sure resource allocation is carefully performed and orchestrated, preventing over- or under-provisioning of resources and keeping end-to-end delays comparable to those observed in traditional middlebox-based networks. In this paper, we formalize the network function placement and chaining problem and propose an Integer Linear Programming (ILP) model to solve it. Additionally, in order to cope with large infrastructures, we propose a heuristic procedure for efficiently guiding the ILP solver towards feasible, near-optimal solutions. Results show that the proposed model leads to a reduction of up to 25% in end-to-end delays (in comparison to chainings observed in traditional infrastructures) and an acceptable resource over-provisioning limited to 4%. Further, we demonstrate that our heuristic approach is able to find solutions that are very close to optimality while delivering results in a timely manner.**

I. Introduction

Middleboxes (or Network Functions - NF) play an essential role in today's networks, as they support a diverse set of functions ranging from security (*e.g.*, firewalling and intrusion detection) to performance (*e.g.*, caching and proxying) [1]. As currently implemented nowadays, middleboxes are difficult to deploy and maintain. This is mainly because cumbersome procedures need to be followed, such as dealing with a variety of custom-made hardware interfaces and manually chaining middleboxes to ensure the desired network behavior. Further, recent studies show that the number of middleboxes in enterprise networks (as well as in datacenter and ISP networks) is similar to the number of physical routers [2]–[4]. Thus, the aforementioned difficulties are exacerbated by the complexity imposed by the high number of network functions that a network provider has to cope with, leading to high operational expenditures. Moreover, in addition to costs related to manually deploying and chaining middleboxes, the need for frequent hardware upgrades adds up to substantial capital investments.

Network Function Virtualization (NFV) has been proposed to shift middlebox processing from specialized hardware appliances to software running on commoditized hardware [5]. In addition to potentially reducing acquisition and maintenance costs, NFV is expected to allow network providers to make most of the benefits of virtualization on the management of network functions (*e.g.*, elasticity, performance, flexibility, etc.). In this context, Software-Defined Networking (SDN) can be considered a convenient complementary technology, which, if available, has the potential to make the chaining of the aforementioned network functions much easier. In fact, it is not unreasonable to state that SDN has the potential to revamp the Service Function Chaining (SFC)[1] problem. In short, the problem consists of making sure network flows go efficiently through end-to-end paths traversing sets of middleboxes. In the NFV/SDN realm and considering the flexibility offered by this environment, the problem consists of (sub)optimally defining how many instances of virtual network functions are necessary and where to place them in the infrastructure. Furthermore, the problem encompasses the determination of end-to-end paths over which known network flows have to be transmitted so as to pass through the required placed network functions.

Despite consistent efforts to make NFV a reality [1], [6], little has been done to efficiently perform the placement and chaining of virtual network functions on physical infrastructures. This is particularly challenging mainly for two reasons. First, depending on how virtual network functions are positioned and chained, end-to-end latencies may become intolerable. This problem is aggravated by the fact that processing times tend to be higher, due to the use of virtualization, and may vary, depending on the type of network function and the hardware configuration of the device hosting it. Second, resource allocation must be performed in a cost-effective manner, preventing over- or under-provisioning of resources. Therefore, placing network functions and programming network flows in a cost-effective manner while ensuring acceptable end-to-end delays represents an essential step toward enabling the use of NFV in production environments.

In this paper, we formalize the network function placement and chaining problem and propose an optimization model to solve it. Additionally, in order to cope with large infrastructures, we propose a heuristic procedure. Both optimal and heuristic approaches are evaluated considering different use cases and metrics, such as the number of instantiated virtual network functions, physical and virtual resource consumption, and end-to-end latencies. The main contributions of this paper are then: *(i)* the formalization of the network function placement and chaining problem by means of an ILP model; *(ii)* the proposal of a heuristic solution, and *(iii)* the evaluation of both proposed approaches and discussion of the obtained results.

The remainder of this paper is organized as follows. In Section 2, we discuss related work in the area of network function virtualization. In Section 3, we formalize the network function placement and chaining problem and propose both an optimal ILP model and a heuristic approach to solve it. In

[1]In this paper, the terms network function and service function are used interchangeably.

Section 4, we present and discuss the results of an evaluation of the model and heuristic. Last, in Section 5 we conclude the paper with final remarks and perspectives for future work.

II. Related Work

We now review some of the most prominent research work related to network function virtualization and the network function placement and chaining problem. We start the section by discussing recent efforts aimed at evaluating the technical feasibility of deploying network functions on top of commodity hardware. Then, we review preliminary studies carried out to solve different aspects of the virtual network function placement and chaining problem.

Hwang *et al.* [6] propose the NetVM platform to allow network functions based on Intel DPDK technology to be executed at line-speed (*i.e.*, 10 Gb/s) on top of commodity hardware. According to the authors, it is possible to accelerate network processing by mapping NIC buffers to user space memory. In another investigation, Martins *et al.* [1] introduce a high-performance middlebox platform named ClickOS. It consists of a Xen-based middlebox software, which, by means of alterations in I/O subsystems (back-end switch, virtual net devices and back and front-end drivers), can sustain a throughput of up to 10 Gb/s. The authors show that ClickOS enables the execution of hundreds of virtual network functions concurrently without incurring significant overhead (in terms of delay) in packet processing. The results obtained by Hwang *et al*. and Martins *et al*. are promising and definitely represent an important milestone to make the idea of virtual network functions a reality.

With respect to efficient placement and chaining of network functions, the main contribution of this paper, Barkai *et al.* [7] and Basta *et al.* [8] have recently taken a first step toward modeling this problem. Barkai *et al.*, for example, propose mechanisms to program network flows to an SDN substrate taking into account virtual network functions through which packets from these flows need to pass. In short, the problem consists of mapping SDN traffic flows properly (*i.e.*, in the right sequence) to virtual network functions. To solve it in a scalable manner, the authors propose a more efficient topology awareness component, which can be used to rapidly program network flows. Note that they do not aim at providing a (sub)optimal solution to the network function placement and chaining problem as we do in this paper. Instead, the scope of their work is more of an operational nature, *i.e.*, building an OpenFlow-based substrate that is efficient enough to allow flows – potentially hundred of millions, with specific function processing requirements – to be correct and timely mapped and programmed. Our solution could be used together with Barkai's and therefore help the decision on where to optimally place network functions and how to correctly map network flows.

The work by Basta *et al.*, in turn, proposes an ILP model for network function placement in the context of cellular networks and crowd events. More specifically, the problem addressed is the question on whether or not virtualize and migrate mobile gateway functions to datacenters. When applicable, the model also encompasses the optimal selection of datacenters that will host the virtualized functions and SDN controllers. Although the paper covers optimal virtual function placement, the proposed model is restricted, as it does not have to deal with function chaining. Our proposal is, in comparison, a broader, optimal solution. It can be applied to plan not only the placement of multiple instances of virtual network functions on demand, but also to map and chain service functions.

Before summarizing this section, we add a note on the relation between the network function placement and chaining problem and the Virtual Network Embedding (VNE) problem [9]–[12]. Despite some similarities, solutions to the latter are not appropriate to the former. The reason is twofold. First, while in VNE we observe one-level mappings (virtual network requests $\rightarrow$ physical network), in NFV environments we have two-level mappings (service function chaining requests $\rightarrow$ virtual network function instances $\rightarrow$ physical network). Second, while the VNE problem considers only one type of physical device (*i.e.*, routers), a much wider number of different network functions coexist in NFV environments.

As one can observe from the state-of-the-art, the area of network function virtualization is still in its early stages. Most of the effort has been focused on engineering ways of running network functions on top of commodity hardware while keeping performance roughly the same as the one obtained when deploying traditional middlebox-based setups. As far as we are aware of, this paper consolidates a first consistent step towards placing virtual network functions and mapping service function chains. Besides, it captures and discusses the trade-off between resources employed and performance gains in the particular context of NFV.

III. The Network Function Placement and Chaining Problem

In this section, we describe the network function placement and chaining problem and introduce our proposed solution. Next, we formalize it as an Integer Linear Programming model, followed by an algorithmic approach.

A. Problem Overview

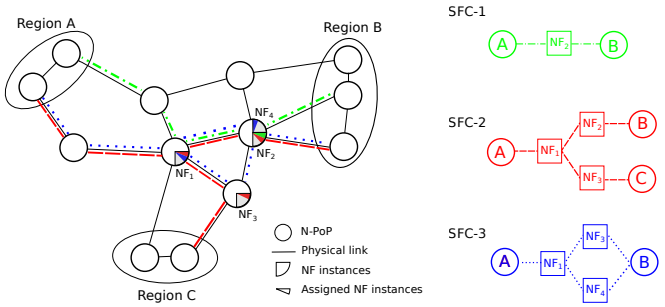
As briefly explained earlier, network function placement and chaining consists of interconnecting a set of network functions (*e.g.*, firewall, load balancer, etc.) through the network to ensure network flows are given the correct treatment. These flows must go through end-to-end paths traversing a specific set of functions. In essence, this problem can be decomposed into three phases: *(i)* placement, *(ii)* assignment, and *(iii)* chaining.

The placement phase consists of determining how many network function instances are necessary to meet the current/expected demand and where to place them in the infrastructure. Virtual network functions are expected to be placed on network points of presence (N-PoPs), which represent groups of (commodity) servers in specific locations of the infrastructure (with processing capacity). N-PoPs, in turn, would be potentially set up either in locations with previously installed commuting and/or routing devices or in facilities such as datacenters.

The assignment phase defines which placed virtual network function instances (in the N-PoPs) will be in charge of each flow. Based on the source and destination of a flow, instances are assigned to it in a way that prevents processing times from causing intolerable latencies. For example, it may be more efficient to assign network function requests to the nearest virtual network function instance or to simply split the requested demand between two or more virtual network functions (when possible).

In the third and final phase, the requested functions are chained. This process consists of creating paths that interconnect the network functions placed and assigned in the previous phases. This phase takes into account two crucial factors, namely end-to-end path latencies and distinct processing delays added by different virtual network functions. Figure 1 depicts the main elements involved in virtual network function placement and chaining. The physical network is composed of N-PoPs interconnected through physical links. There is a set of SFC requests that contain logical sequences of network

functions as well as the endpoints, which implicitly define the paths. Additionally, the provider has a set of virtual network function images that it can instantiate. In the figure, larger semicircles represent instances of network functions running on top of an N-PoP, whereas the circumscribed semicircles represent network function requests assigned to the placed instances. The gray area in the larger semicircles represents processing capacity allocated to network functions that is not currently in use. Dashed lines represent paths chaining the requested endpoints and network functions.



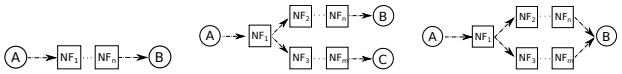
(a) Physical infrastructure. (b) SFC requests.

Fig. 1. Example SFC deployment on a physical infrastructure to fulfill a number of requests.

B. Topological Components of SFC Requests

SFC requests may exhibit different characteristics depending on the application or flow they must handle. More specifically, such requests may differ topologically and/or in size. In this paper, we consider three basic types of SFC components, which may be combined with one another to form more complex requests. These three variations – *(i)* line, *(ii)* bifurcated path with different endpoints, and *(iii)* bifurcated path with a single endpoint – are explained next.

The simplest topological component that may be part of an SFC request is a line with two endpoints and one or more network functions. This kind of component is suitable for handling flows between two endpoints that have to pass through a particular sequence of network functions, such as a firewall and a Wide Area Network (WAN) accelerator. The second and third topological components are based on bifurcated paths. Network flows passing through bifurcated paths may end up at the same endpoint or not. Considering flows with different endpoints, the most basic component contains three endpoints (one source and two destinations). Between them, there is a network function that splits the traffic into different paths according to a certain policy. A classical example that fits this topological component is a load balancer connected to two servers. As for bifurcated paths with a single end point, we consider a scenario in which different portions of traffic between two endpoints must be treated differently. For example, part of the traffic has to pass through a specific firewall, while the other part, through an encryption function. Figure 2 illustrates these topological components. As previously mentioned, more sophisticated SFC requests may be created by freely combining these basic topological components among themselves or in a recursive manner.



(a) Line. (b) Bifurcated path with different endpoints. (c) Bifurcated path with a single endpoint.

Fig. 2. Basic topological components of SFC requests.

C. Definitions and Modeling

Next, we detail the inputs, variables, and constraints of our optimization model. Superscript letters represent whether a set or variable refers to service chaining requests (S) or physical (P) resources, or whether it relates to nodes (N) or links (L).

NFV Infrastructure and Service Function Chaining. The topology of the NFV infrastructure, as well as that of each SFC, is represented as a directed graph $G = (N, L)$. Vertices N represent network points of presence (N-PoPs) in physical infrastructures or network functions in SFCs. Each N-PoP represents a location where a network function may be implemented. In turn, each edge $(i, j) \in L$ represents an unidirectional link. Bidirectional links are represented as a pair of edges in opposite directions (*e.g.*, (a, b) and (b, a)). Thus, the model allows the representation of any type of physical topology, as well as any SFC forwarding graph.

In real environments, physical devices have a limited amount of resources. In our model, the CPU capacity of each N-PoP is represented as C_i^P. In turn, each physical link in the infrastructure has a limited bandwidth capacity and a particular delay, represented by $B_{i,j}^P$ and $D_{i,j}^P$, respectively. Similarly, SFCs require a given amount of resources. Network functions require a specific amount of CPU, represented as C_i^S. Additionally, each SFC being requested has a bandwidth demand and a maximum delay allowed between its endpoints, represented as $B_{i,j}^S$ and D^S, respectively.

Virtual Network Functions. Set F represents possible virtual network functions (*e.g.*, firewall, load balancer, NAT, etc.) that may be instantiated/placed by the infrastructure operator on top of N-PoPs. Each network function can be instantiated at most U_m times, which is determined by the number of licenses the provider has purchased. Each virtual function instance requires a given amount of physical resources (which are used by SFCs mapped to that instance). Each instance provides a limited amount of resources represented by F_m^{cpu}. This enables our model to represent instances of the same type of network function with different sizes (*e.g.*, pre-configured instances for services with higher or lower demand). Each function $m \in F$ has a processing delay associated with it, which is represented by F_m^{delay}. Moreover, we consider that each mapped network function instance may be shared by one or more SFCs whenever possible.

SFC requests. Set Q represents SFCs that must be properly assigned to network functions. SFCs are composed of chains of network functions and each requested function is represented by q. Each link interconnecting the chained functions requires a given amount of bandwidth, represented by $B_{i,j}^V$. Furthermore, each request has at least two endpoints, representing specific locations on the infrastructure. The required locations of SFC endpoints are stored in set S^C. Likewise, the physical location of each N-PoP i is represented in S^P. Since the graph of an SFC request may represent any topology, we assume that the set of virtual paths available to carry data between pairs of endpoints is known in advance. As there are efficient algorithms to compute paths, we opted to compute them in advance without loss of generality to the model. This set is represented by P.

Variables. The variables are the outputs of our model, and represent the optimal solution of the service function chaining problem for the given set of inputs. These variables indicate in which N-PoP virtual network functions are instantiated (placed). Further, these variables indicate the assignment of SFCs being requested to virtual network functions placed in the infrastructure. If a request is accepted, each of its virtual functions is mapped to an N-POP, whereas each link in the

chain is mapped to one or more consecutive physical links (*i.e.*, a physical path).

- $y_{i,m,j} \in \{0,1\}$ – Virtual network function placement, indicates whether instance j of network function m is mapped to N-PoP i.
- $A^N_{i,q,j} \in \{0,1\}$ – Assignment of required network functions, indicates whether virtual network function j, required by SFC q, is serviced by a network function placed on N-PoP i.
- $A^L_{i,j,q,k,l} \in \{0,1\}$ – Chaining allocation, indicates whether physical link (i,j) is hosting virtual link (k,l) from SFC q.

Based on the above inputs and outputs, we now present the objective function and its constraints. The objective function of our model aims at minimizing the number of virtual network function instances mapped on the infrastructure. This objective was chosen due to the fact that this aspect has the most significant and direct impact on the network provider's costs. However, our model could be easily adapted to use other objective functions, such as multi-objective ones that consider different factors simultaneously (*e.g.*, number of network function instances and end-to-end delays). The purpose of each constraint of our model is explained next.

Objective:

$$\textbf{Min} \sum_{i \in R^P, m \in F, j \in U_m} y_{i,m,j} \tag{1}$$

Subject to:

$$\sum_{m \in F, j \in U_m} y_{i,m,j} \cdot F^{cpu}_{m,j} \leq C^P_i \qquad \forall i \in R^P \tag{2}$$

$$\sum_{q \in Q, j \in R^V_q : q = F_m} C^S_{q,j} \cdot A^N_{i,q,j} \leq \sum_{j \in U_m} y_{i,m,j} \cdot F^{cpu}_{m,j} \qquad \forall i \in R^P, m \in F \tag{3}$$

$$A^N_{i,q,k} \leq \sum_{m \in F, j \in U_m : m = F_k} y_{i,m,j} \qquad \forall i \in R^P, q \in Q, k \in R^S_q \tag{4}$$

$$\sum_{q \in Q, (k,l) \in L^V} B^S_{q,k,l} \cdot A^L_{i,j,q,k,l} \leq B^P_{i,j} \qquad \forall (i,j) \in L^P \tag{5}$$

$$\sum_{i \in R^P} A^N_{i,q,j} = 1 \qquad \forall q \in Q, k \in R^S_q \tag{6}$$

$$\sum_{j \in R^P} A^L_{i,j,q,k,l} - \sum_{j \in R^P} A^L_{j,i,q,k,l} = A^N_{i,q,k} - A^N_{i,q,l} \qquad \forall q \in Q, i \in R^P, (k,l) \in L^S_q \tag{7}$$

$$A^N_{i,q,j} \cdot j = A^N_{i,q,k} \cdot l \qquad \forall (i,j) \in S^P, q \in Q, (k,l) \in S^S_q \tag{8}$$

$$\sum_{(i,j) \in L^P, (k,l) \in L^S_q : (k,l) \in p} A^L_{i,j,q,k,l} \cdot D^P_{i,j} + \sum_{i \in R^P, k \in R^S_q : k \in p} A^N_{i,q,j} \cdot F^{delay}_k \leq D^S_k \qquad \forall q \in Q, p \in P_q \tag{9}$$

Constraint 2 ensures that the sum of CPU capacities required by network function instances mapped to N-PoP i does not exceed the amount of available physical resources. In turn, constraint 3 ensure that the sum of processing capacities required by elements of SFCs does not exceed the amount of virtual resources available on network function m mapped to N-PoP i. Constraint 4 ensures that, if a network function being requested by an SFC is assigned to N-PoP i, then at least one network function instance should be running (placed) on i. Constraint 5 ensures that the virtual path between the required endpoints has enough available bandwidth to carry the amount of flow required by SFCs. Constraint 6 ensures that every required SFC (and its respective network functions) is mapped to the infrastructure. Constraint 7 consists in building the virtual paths between the required endpoints. Constraint 8 ensures that the required endpoints are mapped to devices in the requested physical locations. Last, constraint 9 ensures that end-to-end latency constraints on mapped SFC requests will be met (the first part of the equation is a sum of the delay incurred by end-to-end latencies between mapped endpoints, while the second part defines the delay incurred by packet processing on virtual network functions).

D. Proposed Heuristic

In this subsection we present our heuristic approach for efficiently placing, assigning, and chaining virtual network functions. We detail each specific procedure it uses to build a feasible solution, and present an overview of its algorithmic process.

In this particular problem, the search procedure performed by the integer programming solver leads to an extensive number of symmetrical feasible solutions. This is mainly because there is a considerable number of potential network function mappings/assignments that satisfy all constraints, in addition to the fact that search schemes conducted by commercial solvers are not specialized for the problem in hand.

To address the aforementioned issues, our heuristic approach dynamically and efficiently guides the search for solutions performed by solvers in order to quickly arrive at high quality, feasible ones. This is done by performing a binary search to find the lowest possible number of network function instances that meets the current demands. In each iteration, the heuristic employs a modified version of the proposed ILP model in which the objective function is removed and transformed into a constraint, resulting in a more bounded version of the original model. This strategy takes advantage of two facts: first, there tends to be a significant number of feasible, symmetrical solutions that meet our criteria for optimality, and once the lowest possible number of network function instances is determined, only one such solution needs to be found; and second, commercial solvers are extremely efficient in finding feasible solutions.

Algorithm 1 presents a simplified pseudocode version of our heuristic approach, and its details are explained next. The heuristic performs a binary search that attempts to find a more constrained model by dynamically adjusting the number of network functions that must be instantiated on the infrastructure. The upper bound of this search is initially set to the maximum number of network functions that may be instantiated on the infrastructure (line 2), while the lower bound is initialized as 1 (line 3). In each iteration, the maximum number of network function instances allowed is represented by variable nf, which is increased or decreased based on the aforementioned upper and lower bounds (line 6). After nf is updated, the algorithm transforms the original model into the bounded one by removing the objective function (line 7) and adding a new constraint (line 8), considering the computed value for nf. The added constraint is shown in Equation 10.

$$\sum_{i \in R^P, m \in F, j \in U_m} y_{i,m,j} \leq nf \quad (10)$$

In line 9, a commercial solver is used to obtain a solution for the bounded model within an acceptable time limit. In each iteration, the algorithm stores the best solution found so far (*i.e.*, the solution s with the lowest value for nf – line 11). Afterwards, it adjusts the upper or lower bound depending on whether the current solution is feasible or not (lines 12 and 14). Last, it returns the best solution found (s', which represents variables y, A^N and A^L).

Although the proposed heuristic uses an exact approach to find a feasible solution for the problem, $timeLimit$ (in line 9) should be fine-tuned considering the size of the instance being handled to ensure the tightest solution will be found. In our experience, for example, a time limit in the order of minutes is sufficient for dealing with infrastructures with 200 N-PoPs.

Input: $Infrastructure\ G$, $set\ Q\ of\ SFCs$, $set\ VNF\ of\ network\ functions$, $timeLimit$
Output: $Variables\ y_{i,m,j}$, $A^N_{i,q,j}$, $A^L_{i,j,q,k,l}$

```
1  s, s' ← ∅
2  upperBound ← |F|
3  lowerBound ← 1
4  nf ← (upperBound + lowerBound)/2
5  while nf ≥ lowerBound and nf ≤ upperBound do
6  |  nf ← (upperBound + lowerBound)/2
7  |  Remove objective function
8  |  Add constraint : Σ_{i∈R^P, m∈F, j∈U_m} y_{i,m,j} ≤ nf
9  |  s ← solveAlteredModel(timeLimit)
10 |  if s is feasible then
11 |  |  s' ← s
12 |  |  upperBound ← nf
13 |  else
14 |  |  lowerBound ← nf
15 |  end
16 end
17 if s' = ∅ then
18 |  return infeasible solution
19 else
20 |  return s'
21 end
```

Algorithm 1: Overview of the proposed heuristic.

IV. Evaluation

In order to evaluate the provisioning of different types of SFCs, the ILP model formalized in the previous section was implemented and run in *CPLEX Optimization Studio*[2] version 12.4. The heuristic, in turn, was implemented and run in Python. All experiments were performed on a machine with four Intel Xeon E5-2670 processors and 56 GB of RAM, using the Ubuntu GNU/Linux Server 11.10 x86_64 operating system.

A. Workloads

We consider four different types of SFC components. Each type uses either one of the topological components described in Subsection III-B or a combination of them. The first component is a line composed of a single firewall between the two endpoints (Figure 2(a)). The second component used consists of a bifurcated path with different endpoints (Figure 2(b)). This component is composed of a load balancer splitting the traffic between two servers. These two types of components are comparable since their end-to-end paths pass through exactly one network function. The third and fourth components use the same topologies of the previously described ones, but vary in size. The third component is a line (like Component 1) composed of two chained network functions – a firewall followed by an encryption network function (*e.g.*, VPN). The fourth component is a bifurcated path (like Component 2), but after the load balancer, traffic is forwarded to one more network function – a firewall. These particular network functions were chosen due to being commonly referenced in recent literature; however, they could be easily replaced with any other functions if so desired. All network functions requested by SFCs have the same requirements in terms of CPU and bandwidth. Each network function requires 12.5% of CPU, while the chainings between network functions require 1Gbps of bandwidth. When traffic passes through a load balancer, the required bandwidth is split between the paths. The values for CPU and bandwidth requirements were fixed after a preliminary evaluation, which revealed that they did not have a significant impact on the obtained results. Moreover, the establishment of static values for these parameters facilitates the assessment of the impact of other, more important factors.

The processing times of virtual network functions (*i.e.*, the time required by these functions to process each incoming packet) considered in our evaluation are shown in Table I. These values are based on the study conducted by Dobrescu *et al.* [13], in which the authors determine the average processing time of a number of software-implemented network functions.

TABLE I. Processing times of physical and virtual network functions used in our evaluation.

Network Function	Processing Time (physical)	Processing Time (virtual)
Load Balancer	0.2158 sec	0.6475 sec
Firewall	2.3590 sec	7.0771 sec
VPN Function	0.5462 sec	1.6385 sec

Networks used as physical substrates were generated with Brite[3]. The topology of these networks follows the Barabasi-Albert (BA-2) [14] model. This type of topology was chosen as an approximation of those observed in real ISP environments. Physical networks have a total of 50 N-PoPs, each with total CPU capacity of 100%, while the bandwidth of physical links is 10 Gbps. The average delay of physical links is 30ms. This value is based on the study conducted by Choi *et al.* [15], which characterizes typical packet delays in ISP networks.

In order to provide a comparison between virtualized network functions and non-virtualized ones, we consider baseline scenarios for each type of SFC. These scenarios aim at reproducing the behavior of environments that employ physical middleboxes rather than NFV. Our baseline consists of a modified version of our model, in which the total number of network functions is exactly the number of different functions being requested. Moreover, the objective function attempts to find the minimum chaining length between endpoints and network functions. In baseline scenarios, function capacities are adjusted to meet all demands and, therefore, we do not consider capacity constraints. Further, processing times are three times lower than those in virtualized environments. This is in line with the study of Basta *et al.* [8]. These processing

[2]http://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/

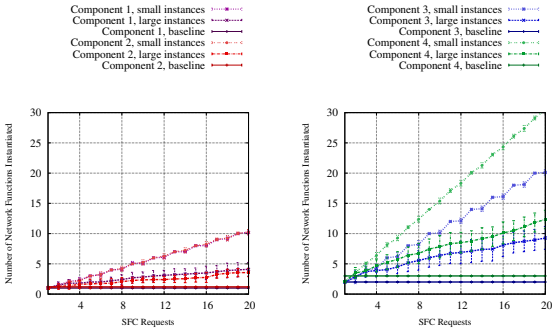
[3]http://www.cs.bu.edu/brite/

times, like the ones related to virtual network functions, are shown in Table I.

In our experiments, we consider two different profiles of network function instances. In the first one, instances may require either 12.5% or 25% of CPU, leading to smaller instance sizes. In the second profile, instances may require 12.5% or 100%, leading to larger instances overall. We first evaluate our optimal approach considering individual types of requests. Next, we evaluate the effect of a mixed scenario with multiple types of SFCs. Last, we evaluate our proposed heuristic using large instances. Each experiment was repeated 30 times, with each repetition using a different physical network topology. All results have a confidence level of 90% or higher.

B. Results

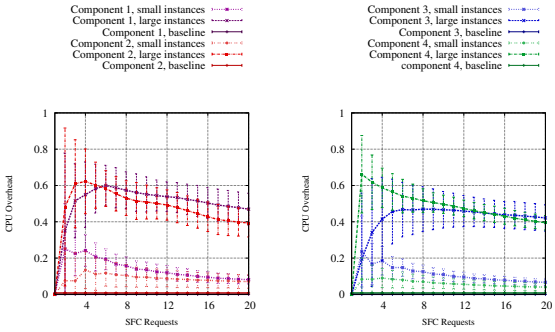
First, we analyze the number of network functions instances needed to cope with an increasing number of SFC requests. Figure 3 depicts the average number of instantiated network functions with the number of SFC requests varying from 1 to 20. At each point on the graph, all previous SFC requests are deployed together. It is clear that the number of instances is proportional to the number of SFC requests. Further, we observe that smaller instance sizes lead to a higher number of network functions being instantiated. Considering small instances, scenarios with Components 1 and 2 require, on average, 10 network function instances (Figure 3(a)). In contrast, scenarios with Components 3 and 4 require, on average, 20 and 30 instances (Figure 3(b)), respectively. For large instances, scenarios with Components 1 and 2 require, respectively, 4 and 3 network function instances, while those with Components 3 and 4 require 9 and 12 instances on average. These results demonstrate that the number of virtual network functions in a SFC request has a much more significant impact on the number of instances needed to service such requests than the chainings between network functions and endpoints. This can be observed, for example, in Figure 3(a), in which Components 1 and 2 only differ topologically and lead to, on average, the same number of instances. In contrast, Figure 3(b) shows that when handling components of type 4 (which have a higher number of network functions than those of type 3), a significantly higher number of network function instances is required.



(a) Components 1 and 2. (b) Components 3 and 4.

Fig. 3. Average number of network function instances.

Figure 4 illustrates the average CPU overhead (*i.e.*, allocated but unused CPU resources) in all experiments. Each point on the graph represents the average overhead from the beginning of the experiment until the current point. In all experiments, CPU overheads tend to be lower when small instances are used. When large instances are allocated, more resources stay idle. Considering small instances, Components 1 and 2 (Figure 4(a)) lead to, on average, CPU overhead of 7.80% and 7.28%, respectively. Components 3 and 4 (Figure 4(b)) lead to, on average, 6.58% and 3.18% CPU overhead. In turn, for large instances, Components 1 and 2 lead to average CPU overheads of 45.61% and 38.68%, respectively. Components 3 and 4 lead to, on average, 40.21% and 40.36% CPU overhead. Observed averages demonstrate that the impact of instance sizes is notably high, with smaller instances leading to significantly lower overheads. Further, we can observe that, in general, CPU overheads tend to be lower when higher numbers of SFCs are being deployed. As more requests are being serviced simultaneously, network function instances can be shared among multiple requests, increasing the efficiency of CPU allocations. In these experiments, the baseline has 0% of CPU overhead as network functions are planned in advance to support the exact demand. Since in a NFV environment network function instances are hosted on top of commodity hardware (as opposed to specialized middleboxes), these overheads – especially those observed for small instances – are deemed acceptable, as they do not incur high additional costs.



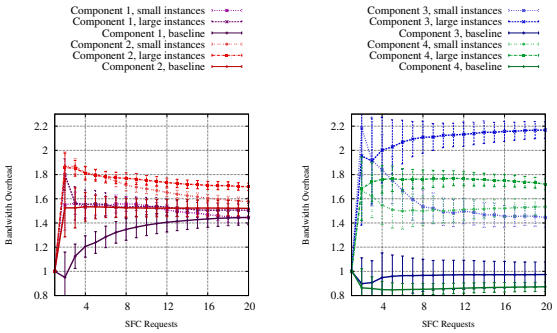
(a) Components 1 and 2. (b) Components 3 and 4.

Fig. 4. Average CPU overhead of network function instances.

Next, Figure 5 shows the average overhead caused by chaining network functions (through virtual links) in each experiment. This overhead is measured as the ratio between the effective bandwidth consumed by SFC virtual links hosted on the physical substrate and the bandwidth requested by such links. In general, the actual bandwidth consumption is higher than the total bandwidth required by SFCs, due to the frequent need to chain network functions through paths composed of multiple physical links. The absence of overhead is observed only when each virtual link is mapped to a single physical link (ratio of 1.0), or when network functions are mapped to the same devices as the requested endpoints (ratio < 1.0). Lower overhead rates may potentially lead to lower costs and allow more SFC requests to be serviced.

Considering large instances, the observed average overhead is 50.49% and 69.87% for scenarios with Components 1 and 2, respectively. In turn, Components 3 and 4 lead to overhead ratios of 116% and 72.01%. This is due to the low number of instantiated network functions (Figure 3), which forces instances to be chained through long paths. Instead, when small instances are considered (*i.e.*, more instances running in a distributed way), overheads tend to be lower. Components 1 and 2 lead to, on average, 44.30% and 57.60% bandwidth overhead, while Components 3 and 4, 44.53% and 53.41%, respectively. When evaluating bandwidth overheads, we can observe that the topological structure of SFC requests has the most significant impact on the results (in contrast to previously discussed experiments). More complex chainings tend to lead to higher bandwidth overheads, although these results are also influenced by other factors such as instance sizes and the number of instantiated functions. In these experiments the baseline overhead tends to be lower than the others as the

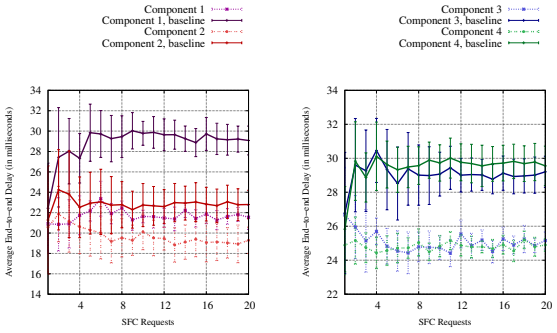
objective function prioritizes shortest paths (in terms of number of hops) between endpoints and network functions.



(a) Components 1 and 2. (b) Components 3 and 4.

Fig. 5. Average bandwidth overhead of SFCs deployed in the infrastructure.

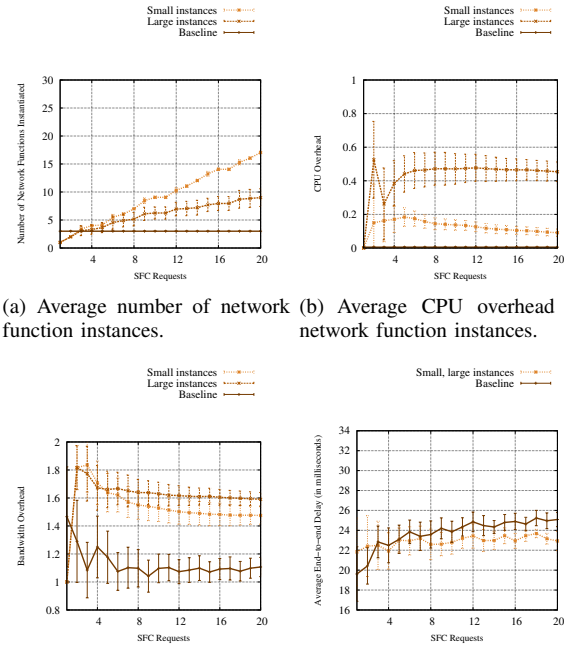
Figure 6 depicts the average end-to-end delay, in milliseconds, observed between endpoints in all experiments. The end-to-end delay is computed as a sum of the path delays and network function processing times. In this figure, results for scenarios with small and large instances are grouped together, as average delays are the same. The observed end-to-end delay for all components tends to be lower than the delay observed for the baseline scenario. This is mainly due to the better positioning of network functions and chainings between them. Furthermore, the model promotes a better utilization of the variety of existing paths in the infrastructure. Although the baseline scenario aims at building minimum chainings (in terms of hops), we observe that: *(i)* minimum chaining does not always lead to global minimum delay; *(ii)* when baseline scenarios overuse the shortest paths, other alternative paths remain unused due to the depletion of resources in specific locations (mainly in the vicinity of highly interconnected nodes). In comparison with baseline scenarios, Component 1 leads to, on average, 25% lower delay (21.55ms compared to 29.07ms), while Component 2 leads to, on average, 15.40% lower delay (19.28ms compared to 22.79ms). In turn, Component 3 leads to, on average, 13.86% lower delay than its baseline (25.15ms compared to 29.20ms), while Component 4 leads to 15.75% lower delay (24.89ms compared to 29.55ms). In summary, even though our baseline scenarios are planned in advance to support exact demands and we consider processing times of virtual network functions to be three times those of physical ones, end-to-end delays are still lower in virtualized scenarios. This advantage may become even more significant as the estimated processing times of virtual network functions get closer in the future to those observed in physical middleboxes.



(a) Components 1 and 2. (b) Components 3 and 4.

Fig. 6. Average end-to-end delay of SFCs deployed in the infrastructure.

After analyzing the behavior of SFCs considering homogeneous components, we now analyze the impact of a mixed scenario. In it, Components 1, 2, and 4 are repeatedly deployed in the infrastructure sequentially. Figure 7 presents the results for the mixed scenario. Although there are different topological SFC components being deployed together in the same infrastructure, the results exhibit similar tendencies as those of homogeneous scenarios. In Figure 7(a), we observe that the average number of network functions (on average, 17 network functions when considering small instances and 9 functions considering large instances) is proportional to the obtained average values depicted in Figures 3(a) and 3(b). The average CPU overhead also remains similar (9.12% considering small instances and 45.37% considering large ones). In turn, the average overhead caused by chaining network functions in the mixed scenario is of 59.06% and 47.51%, for small and large instances, respectively. Despite these similarities, end-to-end delays tend to be comparatively lower than the ones observed in homogeneous scenarios. The delay observed in the proposed chaining approach is 8.57% lower than that of the baseline (22.93ms in comparison to 25.08ms). This is due to the combination of requests with different topological structures, which promotes the use of a wider variety of physical paths (which, in turn, leads to lower overutilization of paths). Similarly to homogeneous scenarios, average end-to-end delays are the same considering small and large instances.



(a) Average number of network function instances. (b) Average CPU overhead of network function instances.

(c) Average bandwidth overhead. (d) Average end-to-end delay.

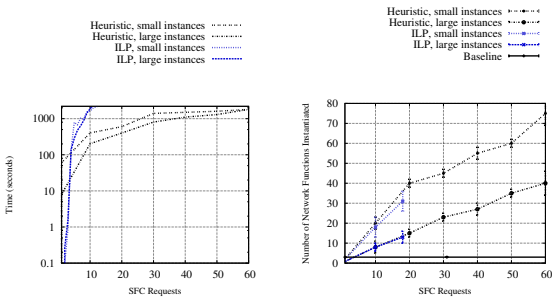
Fig. 7. Mixed scenario including Components 1, 2, and 4.

We now proceed to the evaluation of our proposed heuristic approach. The heuristic was subjected the same scenarios as the ILP model, in addition to ones with a larger infrastructure. Considering the scenarios presented so far (*i.e.*, with physical infrastructures with 50 nodes and 20 SFC requests), our heuristic was able to find an optimal solution in all cases. We omit such results due to space constraints. We emphasize, however, that the heuristic approach was able to find an optimal solution in a substantially shorter time frame in comparison to the ILP model, although the solution times of both approaches remained in the order of minutes. The average solution times of the ILP model and the heuristic considering all scenarios were of, respectively, 8 minutes and 41 seconds and 1 minute

and 21 seconds.

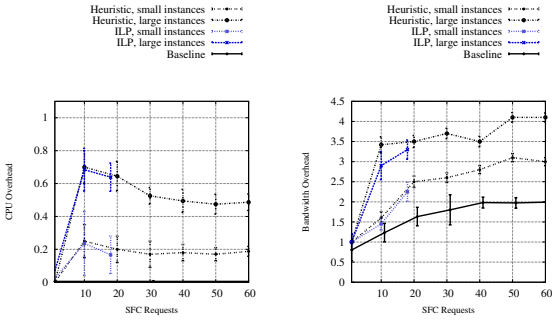
Last, we evaluate our heuristic approach on a large NFV infrastructure. In this experiment, we consider a physical network with 200 N-PoPs and a maximum of 60 SFC components of type 4. The delay limit was scaled up to 90ms in order to account for the larger network size. Figure 8(a) depicts the average time needed to find a solution using both the ILP model and the heuristic. The ILP model was not able to find a solution in a reasonable time in scenarios with more than 18 SFCs (the solution time was longer than 48 hours). The heuristic approach, in turn, is able to properly scale to cope with this large infrastructure, delivering feasible, high-quality solutions in a time frame of less than 30 minutes. As in previous experiments, small network function instances lead to higher solution times than large ones. This is mainly because smaller instances lead to a larger space of potential solutions to be explored.

Although the heuristic does not find the optimal solution (due to time constraints), Figures 8(b), 8(c), 8(d) and 8(e) show that the solutions obtained through this approach present a similar level of quality to the ones obtained optimally. Figure 8(b) depicts the average number of instantiated network functions with the number of SFC requests varying from 1 to 60. As in previous experiments, the number of instances remains proportional to the number of SFC requests. Smaller instance sizes lead to a higher number of network functions being instantiated. Considering small sizes, 75 network functions instances are required on average. In contrast, for large sizes, 40 instances are required on average. Figure 8(c), in turn, illustrates the average CPU overhead. For small instances, CPU overhead is limited to 18.77%, while for large instances it reaches 48.65%. Similarly to the results concerning the number of network function instances, CPU overheads in these experiments also follow the trends observed in previous ones. Next, Figure 8(d) presents bandwidth overheads. Small instances lead to a bandwidth overhead of 300%, while for large instances this overhead is, on average, 410%. These particularly high overheads are mainly due to the increase on the average length of end-to-end paths, as the physical network is significantly larger. Note that the bandwidth overhead observed in the baseline scenario (198%) is also significantly higher than those observed in experiments employed on the small infrastructure. Last, Figure 8(e) depicts the average end-to-end delay observed in large infrastructures. In line with previous results, the end-to-end delay tends to be lower than the delay observed in the baseline scenario. The scenario considering Component 4 presents, on average, 17.72% lower delay than the baseline scenario (70.30ms compared to 82.76ms). In short, these results demonstrate that: *(i)* the heuristic is able to find solutions with a very similar level of quality as the optimization model for small infrastructures; and *(ii)* as both infrastructure sizes and the number of requests increase, the heuristic is able to maintain the expected level of quality while still finding solutions in a short time frame.



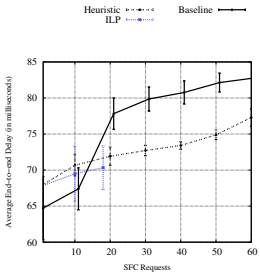
(a) Average time needed to find a solution.

(b) Average number of network function instances.



(c) Average CPU Overhead of network function instances.

(d) Average bandwidth overhead of SFCs deployed in the infrastructure.



(e) Average end-to-end delay of SFCs deployed in the infrastructure.

Fig. 8. Scenario considering a large infrastructure and components of type 4.

V. Conclusion

NFV is a prominent network architecture concept that has the potential to revamp the management of network functions. Its wide adoption depends primarily on ensuring that resource allocation is efficiently performed so as to prevent over- or under-provisioning of resources. Thus, placing network functions and programming network flows in a cost-effective manner while guaranteeing acceptable end-to-end delays represents an essential step towards a broader adoption of this concept.

In this paper, we formalized the network function placement and chaining problem and proposed an optimization model to solve it. Additionally, in order to cope with large infrastructures, we proposed a heuristic procedure that dynamically and efficiently guides the search for solutions performed by commercial solvers. We evaluated both optimal and heuristic approaches considering realistic workloads and different use cases. The obtained results show that the ILP model leads to a reduction of up to 25% in end-to-end delays and an acceptable resource over-provisioning limited to 4%. Further, we demonstrate that our heuristic scales to larger infrastructures while still finding solutions that are very close to optimality in a timely manner.

As perspectives for future work, we envision extending the evaluation of the proposed solutions by applying them to other types of SFCs and ISP topologies, as well as conducting an in-depth analysis of the inter-relationships between their parameters. Moreover, we intend to explore mechanisms to reoptimize network function placements, assignments, and chainings. Further, we intend to explore exact solutions for the problem, such as matheuristics.

REFERENCES

[1] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, 2014.

[2] D. A. Joseph, A. Tavakoli, and I. Stoica, "A policy-aware switching layer for data centers," in *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2008.

[3] T. Benson, A. Akella, and A. Shaikh, "Demystifying configuration challenges and trade-offs in network-based isp services," in *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2011.

[4] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[5] Network Functions Industry Specification Group, "Network function virtualisation (nfv): An introduction, benefits, enablers, challenges and call for action," in *SDN and OpenFlow World Congress*, 2012, pp. 1–16.

[6] J. Hwang, K. K. Ramakrishnan, and T. Wood, "Netvm: High performance and flexible networking using virtualization on commodity platforms," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, 2014.

[7] S. Barkai, R. Katz, D. Farinacci, and D. Meyer, "Software defined flow-mapping for scaling virtualized network functions," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013.

[8] A. Basta, W. Kellerer, M. Hoffmann, H. J. Morper, and K. Hoffmann, "Applying nfv and sdn to lte mobile core gateways, the functions placement problem," in *Proceedings of the 4th Workshop on All Things Cellular: Operations, Applications and Challenges*, 2014.

[9] M. Yu, Y. Yi, J. Rexford, and M. Chiang, "Rethinking virtual network embedding: Substrate support for path splitting and migration," *SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 17–29, Mar. 2008.

[10] M. Chowdhury, M. R. Rahman, and R. Boutaba, "Vineyard: Virtual network embedding algorithms with coordinated node and link mapping," *IEEE/ACM Transactions on Networking*, vol. 20, no. 99, pp. 206–219, 2012.

[11] M. Rabbani, R. Pereira Esteves, M. Podlesny, G. Simon, L. Zambenedetti Granville, and R. Boutaba, "On tackling virtual data center embedding problem," in *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, May 2013, pp. 177–184.

[12] L. R. Bays, R. R. Oliveira, L. S. Buriol, M. P. Barcellos, and L. P. Gaspary, "A heuristic-based algorithm for privacy-oriented virtual network embedding," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Krakow, Poland, May 2014.

[13] M. Dobrescu, K. Argyraki, and S. Ratnasamy, "Toward predictable performance in software packet-processing platforms," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[14] R. Albert and A.-L. Barabási, "Topology of evolving networks: Local events and universality," *Physical Review Letters*, vol. 85, pp. 5234 – 5237, Dec 2000.

[15] B.-Y. Choi, S. Moon, Z.-L. Zhang, K. Papagiannaki, and C. Diot, "Analysis of point-to-point packet delay in an operational network," *Computer Networks*, vol. 51, pp. 3812–3827, 2007.