SPRING: Scaling, Placement, and Routing of Heterogeneous Services with Flexible Structures

Sevil Dräxler
Paderborn University
Paderborn, Germany
sevil.draexler@uni-paderborn.de

Holger Karl
Paderborn University
Paderborn, Germany
holger.karl@uni-paderborn.de

***Abstract*—Network softwarization and the establishing solutions in the areas of virtualized resource management and orchestration increase the interoperability of heterogeneous, multi-domain, multi-technology infrastructures. Having different hosting platforms available, service providers can develop different deployment versions for services and virtual network functions, each optimized different resource types with different characteristics (e.g., CPUs vs. FPGAs). We formalize the problem of scaling, placement, and routing for heterogeneous services that consist of multi-version components and present a single-step optimization approach and a heuristic algorithm for solving it. We study the trade-offs between deployment costs and service performance and show that our solution approaches can adapt to different requirements, by instantiating different deployment versions of the service components in different locations.**

***Index Terms*—scaling, placement, heterogeneous service**

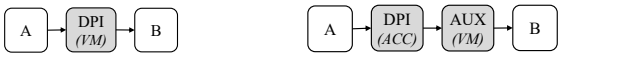
I. Introduction

Network softwarization [1] is complemented by the attempts towards unifying the tools and mechanisms required for the control and orchestration of distributed infrastructures providing heterogeneous resources [2]. This allows deploying and modifying cloud services and virtual network functions (VNFs) on a variety of multi-technology compute, storage, and networking resources, fast and cost-efficiently.

One opportunity that rises from the interoperability of these infrastructures is that service providers can develop and offer flexibly defined services. In contrast to fully specified services with fixed descriptors, these services consist of components that are produced in different *versions*, i.e., using different software implementations, each made for running on a different resource type, offering different advantages. For example, a deep packet inspection (DPI) function can be deployed as a virtual machine (VM) at a low cost only using general-purpose hardware. DPI is a network- and compute-intensive function, so it can achieve a higher performance using special-purpose hardware support, which of course is a more expensive option.

To conform to different needs of the service users (e.g., low cost vs. high performance), the service provider can submit both versions of the DPI to a network operator that offers general-purpose as well as special-purpose hardware (like GPUs or FPGAs). Using appropriate service management and orchestration tools and algorithms, the right version of the DPI can be deployed in the required locations to serve the users. We refer to service components with different deployment options as *multi-version* service components.

This paper has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 762057 (5G-PICTURE), and from the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).



(a) DPI as a VM (b) Accelerated DPI with an auxiliary function

Fig. 1. Example service including a DPI in different deployment versions

In a more complex scenario, different versions of such a function may consist of different number of components. E.g., the hardware-accelerated version of a DPI might require an additional VM for post-processing its results, while the VM version of it can perform all of the required operations within the same VM. In this way, as shown in Fig. 1, the services that include these DPI versions, have a different *structure*. We refer to such a service as a *multi-structure* service consisting of multi-version components or a **heterogeneous service**.

In this paper, we tackle the problem of joint **Scaling, Placement, and RoutING (SPRING)** for heterogeneous services. Similar to our previous work [3], we describe the services using abstract *templates* that include the required components for creating the service as well the desired interconnections among them. The exact number of instances for each component is not a part of the service description; it is determined upon embedding the template into the network, depending on the source and data rate of the service flows. The processing time and the resource demands of each component (for each possible deployment version of it) is defined as *a function of the input data rate* in the template. The actual processing time and resource demands of the instances are then determined while embedding the template into the network.

Our main contributions in this work are as follows.

- Formulation of the joint, single-step scaling, placement, and routing problem for heterogeneous services
- Proof of NP-completeness of the problem
- Mixed-integer programming (MIP) approach for exacts solution
- Heuristic algorithm for fast and close-to-optimal solutions

We first give an overview of related work in Section II. We describe our model and problem in Section III. We present

the MIP formulation in Section IV and the heuristic algorithm in Section V. Finally, we evaluate our presented solution in Section VI before concluding the paper.

II. Related Work

The SPRING problem has similarities to the virtual network embedding (VNE) problem [4] that also aims at an efficient resource allocation. Unlike the fixed structure of virtual networks in VNE, our templates can be embedded with different structures on heterogeneous resources. Among VNE studies, some also consider a heterogeneous substrate network. For example, Li et al. [5] propose a joint resource allocation and VNE solution in 5G core networks. They enable efficient physical resource sharing by optimizing the resource demands before embedding. However, the nodes of the virtual networks in their approach (the service components in our model) have a pre-defined number of instances and a fixed deployment version. Baumgartner et al. [6] consider the VNE problem in the mobile core network, optimizing the structure of the the virtual core network service chain. The flexibility of the service structure in their solution is limited to the way a fixed number of VNFs are grouped and distributed.

Resource allocation is an important problem in the fields of distributed cloud computing [7] and network softwarization [8]. Most of the cloud computing solutions focus on resource allocation for single-component services [9] or consider only a subset of our problem [10]. Keller et al. [11] have introduced a similar problem to SPRING, in distributed clouds, with a more limited set of assumptions and objectives. Different solutions exist for resource allocation in network function virtualization area, each following different objectives, e.g., scalability of the approach [12], maximizing the admitted requests [13], minimizing the costs [14], [15]. We follow a multi-objective optimization approach (Section [**?**]). Similar to our approach, Mehta and Elmroth [16] study the trade-off between cost and performance in mobile edge clouds within heterogeneous 5G networks. None of the mentioned solutions consider the ability of instantiating different versions of the same service component.

Previously, we have tackled the problem of joint scaling, placement, and routing for services [3]. The current work is a major extension of the former one to support heterogeneous services. We are not aware of any similar work to our approach that combines the following steps: (i) scaling, i.e., deciding the right number of instances and allocating the right amount of resources to each of them, (ii) placement, i.e., deciding the location of each instance, (iii) routing, i.e., deciding the optimal paths among instances, (iv) selecting the optimal deployment version for each instance, based on the source and data rate of the service flows. Moreover, in contrast to many of the existing solutions, our approaches can be used for finding the initial embedding of a template, as well as for adjusting existing embeddings.

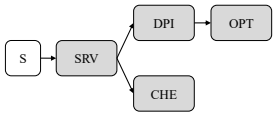


Fig. 2. Example service template including a source, a server (SRV), a deep packet inspector (DPI), a video optimizer (OPT), and a cache (CHE).

III. Model and Problem Formulation

In this section, we describe our model and assumptions and, based on them, formalize the problem we are tackling.

A. Substrate network

The substrate network is a connected, directed graph, $G_{\text{sub}} = (V, L)$. Each network *node* $v \in V$ has a limited capacity of general-purpose processing resources $\text{cap}_{\text{cpu}}(v) \geq 0$ and special-purpose processing (e.g., GPU) resources $\text{cap}_{\text{gpu}}(v) \geq 0$. This can be extended to other types of resources, e.g., memory, FPGAs, etc. These resources are available at a pre-defined cost on each node. We denote the cost of using a unit of CPU and GPU resources for one time unit on node v by $\text{cost}_{\text{cpu}}(v)$ and $\text{cost}_{\text{gpu}}(v)$, respectively. If a certain resource type is not available on a node, we assume the cost of using is infinitely large. Each network *link* $l \in L$ supports a maximum data rate of $\text{cap}(l)$ and has a given delay of $d(l)$.

B. Templates

Service deployment requests are given as templates that describe the general structure of a service. Each service template is a connected, directed, acyclic graph $G_T = (C_T, A_T)$, e.g., as shown in Fig. 2. Each *component* $c \in C_T$ in the template represents a VNF, cloud service component, etc. We describe the details of components in Section III-C.

Each *arc* $a \in A_T$ of the template represents the connectivity among two components. It has a maximum tolerable delay $d_a^{\max}$, which specifies the upper bound for the total delay of the path in the substrate network to which a is mapped. An arc a may be described using additional details regarding the connectivity type and the underlying networking technology. They impose additional constraints to the links that can be used for realizing the connection between the two endpoints of it, i.e., the components src_a, dst_a. Adding these details to the model is straightforward but for simplicity, we do not consider these aspects in the this model.

C. Components and Deployment Versions

Each component c has a given number of inputs n_c^{in} and outputs n_c^{out}, representing the number of ingoing/outgoing connection points. The outgoing data rate of a component depends on the data rate on all its inputs. This is calculated using a given function $\text{fout}_c(\Lambda)$. Λ is a vector of length n_c^{in}. The value of $\text{fout}_c(\Lambda)$ is a vector of length n_c^{out}. This function can be obtained, e.g., by referring to historical usage data or by testing and profiling the component.

Each component may optionally be deployable using different resource types, e.g., as a VM version that can only use

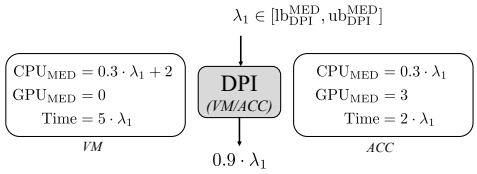
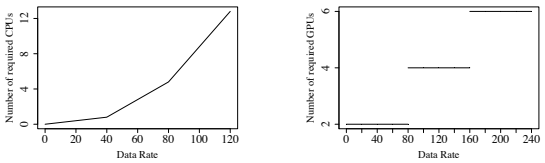


Fig. 3. Example DPI as a VM and as an accelerated version (ACC).



(a) (b)

Fig. 4. Example CPU and GPU demands based on incoming data rate

CPUs, or an accelerated version that needs special-purpose processing resources (in our model, GPU) in addition to CPUs. Each such *deployment version* of a component requires a specific software version, which is available by the tenant. Each component description must include at least one deployment version. We refer to components with more than one deployment version as *multi-version components*.

Each deployment version may consume different types and different number of resources and can result in different processing times and costs. We assume a resource cost model based on the usage duration of a unit of a resource in the substrate network. We describe the specification of required resource units in the rest of this section. Without loss of generality, we assume three possible deployment versions for a component, expressed as the set VER including: a virtual machine version (VM) or a container version (CON) that only need CPUs as processing units, and an accelerated version (ACC) that needs CPUs and GPUs.

Each deployment version $\mathrm{ver} \in \mathrm{VER}$ of component c includes a function $\mathrm{fpt}_c(\mathrm{ver}, \lambda)$ that shows its *maximum* processing time for a total input data rate of λ[1]. Fig. 3 shows the DPI component from the example template of Fig. 2, defined with two different deployment versions: a VM version and an accelerated version. For this DPI, if the input data rate is λ_1, the outgoing data rate from its only output is expected to be at most $0.9 \cdot \lambda_1$, for example because the tenant expects 10 % of video streaming requests to be unauthorized. The VM version requires maximum 2.5 times more processing time than the accelerated version.

CPU and GPU demands of components rarely have a linear relationship with the input load. To overcome computational difficulties in the problem formulation (e.g., in the MIP described in Section IV), we assume the CPU and GPU demands are given as piecewise linear functions that approximate non-linear dependencies of the resource demands on the load. Fig. 4a shows how such a piecewise linear function could represent the CPU demands of an example component.

In case a resource type like GPU cannot be shared among different processes, piecewise constant functions can be used, representing the step-wise increase of the number of required resource units by increasing load, e.g., as shown in Fig. 4b. The piecewise linear functions $\mathrm{fcpu}_c(\mathrm{ver}, \Lambda)$, $\mathrm{fgpu}_c(\mathrm{ver}, \Lambda)$ specify the CPU and GPU demands of the deployment version ver of component c. The demand is calculated based on the vector Λ of ingoing data rates on inputs of the component.

The condition for selecting the right linear function to calculate the resource demand is defined based on the total load that should be handled, i.e., the sum of data rates on all inputs of the component. For example, the function shown in Fig. 4b can be expressed as in Equation 1, where $\lambda = \sum_{i \in n_c^{\mathrm{in}}} (\Lambda)_i$ is the sum of data rates on all inputs of the accelerated deployment version of component c and $(\Lambda)_i$ is the i-th element of vector Λ.

$$\mathrm{fgpu}_c(\mathrm{ACC}, \Lambda) = \begin{cases} 2 & \text{if } \lambda \in (0, 80] \\ 4 & \text{if } \lambda \in (80, 160] \\ 6 & \text{if } \lambda \in (160, 240] \\ \infty & \text{else} \end{cases} \tag{1}$$

For simplifying our notations, we assume the resource demands of all components are defined for the same *number* of load levels, i.e., the same number of non-overlapping intervals defining the domain of the function. We express this with a set LEV, consisting of four possible load levels, i.e., low (LOW), medium (MED), and high (HIG) that can be handled by a specific deployment version of a component and infinite load (INF) above those. Infinite load is any amount of total data rate, which cannot be handled efficiently by the corresponding deployment version using any reasonable (as defined by the tenant who owns the service) amount of resources in a reasonable amount of time.

We use the notations $\mathrm{lb}_c^{\mathrm{lev}}(\mathrm{ver})$, $\mathrm{ub}_c^{\mathrm{lev}}(\mathrm{ver})$ to show the lower and upper bounds of a load level lev for the deployment version ver of a component c. For example, considering an accelerated version of c, if the sum of input data rates to c is between $\mathrm{lb}_c^{\mathrm{MED}}(\mathrm{ACC})$ and $\mathrm{ub}_c^{\mathrm{MED}}(\mathrm{ACC})$, its CPU and GPU demands are calculated by the linear functions $\mathrm{fcpu}_c^{\mathrm{lev}}(\mathrm{ACC}, \Lambda)$, $\mathrm{fgpu}_c^{\mathrm{lev}}(\mathrm{ACC}, \Lambda)$.

The functions are given as a part of the template and can be obtained, e.g., using profiling methods. Fig. 3 shows example functions for CPU and GPU demands of the VM and ACC versions of the DPI function, at medium load level.

Source components are special components that represent the starting point of the flows in the service, e.g., end users or content distribution servers. Source components have zero resource demands, zero processing time, no input, and exactly one outgoing connection to another component. S is the source component of the template in Fig. 2. Each template has exactly one source component. Several instances of each source component can be mapped to different nodes of the

[1]This description is only meaningful if accompanied by the specification of the attributes of the processing unit that has been used to profile the component. Different deployment versions can be defined for different processing unit architectures, resulting in different processing times for a given load.

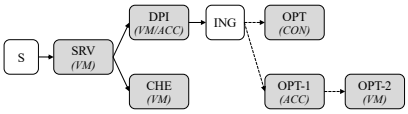


Fig. 5. Example multi-structure video streaming service template including multi-version components.

network where flows are initiated, e.g., to model different locations where users of a service are located.

Fixed components are also special components that are pinned to a certain location in the substrate network, e.g., modeling the endpoints (sinks) of service flows or legacy, physical network functions. They cannot be relocated or scaled. We also assume their resource demands are zero, as they are predefined and fixed.

D. Multi-Structure Templates

By specifying multiple deployment versions for components of a template, we can deploy services in which the best version of each component can be selected considering the trade-offs between the processing time and the deployment cost.

In a more complex scenario, a certain functionality might even be realized in different variations, each including more than one component. For example, the video optimizer function in Fig. 5 can either be deployed as one single container or as a chain of one accelerated component and one VM that need to work together. Selecting one version or another of such a function not only affects the resource demands and processing time of the function but may also influence the structure of the whole service graph. We refer to such templates as *multi-structure templates*.

To support this, we introduce special *ingress* components in our model. These components have at least two outputs, among which only one output can be active. Any component of the template (except the source component) can be marked as an ingress component. For this, the component must include such a load balancing and classification functionality. Otherwise, additional placeholder components with zero resource demands can be defined and added to the template. The placeholder ingress components will not be a part of the actual deployment of the service.

Multi-structure templates may include multi-version components, modeling heterogeneous services. For example, the video streaming template in Fig. 5 includes a multi-version DPI and can take multiple structures depending on which version of the video optimizer is selected.

E. Template Embedding

A template specifies that the corresponding service requires the included components with the specified interconnections for functioning correctly. The *template embedding* process involves deciding:

- how many instances (*horizontal scaling*),
- of which components (*structure selection*)
- using which deployment version of each component (*version selection*),
- with how many resources (*vertical scaling*),
- need to be instantiated in which locations (*placement*),
- and how the traffic should be routed among them (*routing*).

The outcome is an overlay, described in Section III-F.

This process can be used for the initial embedding of templates as well as for updating existing embeddings.

For template embedding, a number of inputs are required. In addition to the templates to be embedded (including the description of their components and arcs), each template T must be accompanied by a set S_T of at least one *source instance*. Source instances are given as tuples $(c, v, \lambda) \in S_T$. $c \in C_T$ refers to the source component of template T and is used to differentiate between source instances of different templates. $v \in V$ specifies the location in the substrate network where the flow initiates. λ shows the data rate of the flow.

If a template T includes fixed components, they should also be given as a part of the input. Fixed components are described as a set X_T of tuples (c, v). $c \in C_T$ shows the fixed component and $v \in V$ is the network node where it is located. Fixed components influence the embedding process of the template, e.g., because the template contains non-fixed components that can be placed at locations with a given maximum delay to the fixed component.

Another optional input is the set of previously existing instances of a template's components. This input is required if the template embedding is used for optimizing and updating an existing embedding. If a template is being embedded for the first time for a tenant, no previous embedding is required. Previous embeddings of the components of template T are given as a set P_T of tuples (c, v, ver). Such a tuple specifies that an instance of component $c \in C_T$ exists on node $v \in V$ with the deployment version ver.

F. Overlay

The template embedding process (Section III-E) maps the abstract description of the service (template), to a concrete deployable graph, i.e., the *overlay*, embedded into the substrate network. Each overlay is a connected, directed graph, $G_{\mathrm{OL}}(T) = (I_{\mathrm{OL}}, E_{\mathrm{OL}})$. It consists of *instances* and *edges*. Each overlay has exactly one template. One template can be embedded several times, e.g., each with different identifiers, belonging to different tenants.

For each instance $i \in I_{\mathrm{OL}}$ in the overlay of template T, there exists a component $c \in C_T$ that contains its specification. Instances are mapped to network nodes and have resources allocated to them. For each component, there can be multiple instances. If there are several deployment versions of a component, each instance of it can have a different version, if required. For simplicity, we assume only one instance of each component can be mapped to one network node, which also means, two different deployment versions of one component cannot be mapped to one network node. However, if one template is embedded multiple times, e.g., each for a different tenant, there are no limitations for embedding multiple

instances of the same component from different embeddings, as they are considered different instances in our model.

Similarly, for each edge $e \in E_{\mathrm{OL}}$ in the overlay of template T, there exists an arc $a \in A_T$ that specifies its endpoints and its maximum allowed latency. Each edge e mapped to a path in the substrate network. This path is a set of network links that starts at the network node on which src_a is mapped and connects it to the node on which dst_a is mapped. Paths cannot include loops.

G. Problem Formulation Summary

As defined in Section III-E, our problem involves version and structure selection, placement, scaling, and routing decisions. Our aim is to take these decisions as a single-step template embedding process. The required inputs are:

- Substrate network
- A template for each service
- Location and data rate of source instances
- Location and deployed version of previously embedded components (optional)
- Location of fixed components (optional)

The output is an overlay mapped to the substrate network, possibly modifying an existing embedding. While embedding the templates, node and link capacities cannot be exceeded. The delay bounds defined between every two component cannot be exceeded either. Our desired solution to this problem *minimizes* the values of the following metrics of interest: deployment cost, processing time, used link capacity, number of added/removed/modified instances.

H. Complexity

Using polynomial-time reduction, we can show that for an instance of our problem, deciding whether a solution with no violation of capacity constraints exists is an NP-complete problem. It is possible in polynomial time in the size of inputs of the problem to check whether the embedding is valid. The output of the problem has also a polynomial relation to the size of the problem inputs. Therefore, our problem is in NP.

In previous work [3], we have proven the NP-hardness of the template embedding problem, JASPER, without considering multiple deployment options and multiple template structures. Our current problem is an extension to JASPER. Given an instance of JASPER, we can construct an instance of the current problem. We assume every component in every template to be embedded has exactly one deployment version that only consumes CPUs, e.g., a VM version and has zero memory demand. We assume the templates have only one possible structure. We set the processing time of every component to zero. We also set the maximum tolerable delay for each arc to infinity, as JASPER does not include arc delays. We assume the templates do not include any fixed components. We can complete the inputs to our problem using the rest of the input provided for an instance of JASPER. A solution without violation of capacity constraints for JASPER is then also a valid solution for our current problem and vice versa.

TABLE I
BINARY DECISION VARIABLES

Variable	Definition
$x_{c,v}$	1 iff an instance of c is mapped to v.
$m_{c,v,\mathrm{ver}}$	1 iff an instance of c is mapped to v with version ver.
$\delta_{c,v}$	1 iff $m_{c,v,\mathrm{ver}} \neq m^*_{c,v,\mathrm{ver}}$, i.e., an instance of c is added, removed, or switched to another version at v.
$l^{\mathrm{cpu}}_{c,v,\mathrm{ver},\mathrm{lev}}$, $l^{\mathrm{gpu}}_{c,v,\mathrm{lev}}$	Helper variable for calculating the CPU, GPU demand, which indicates whether the sum of data rates on the inputs of c on v is larger than or equal to the lower bound that defines the load level lev for version ver.
$u^{\mathrm{cpu}}_{c,v,\mathrm{ver},\mathrm{lev}}$, $u^{\mathrm{gpu}}_{c,v,\mathrm{lev}}$	Helper variable for calculating the CPU, GPU demand, which indicates whether the sum of data rates on the inputs of c on v is smaller than or equal to the lower bound that defines the load level lev for version ver.
$b^{\mathrm{cpu}}_{c,v,\mathrm{ver},\mathrm{lev}}$, $b^{\mathrm{gpu}}_{c,v,\mathrm{lev}}$	Helper variable for calculating the CPU, GPU demand, which indicates whether the sum of data rates on the inputs of c on v is in the range that defines the load level lev for version ver.
$r_{c,v,k}$	1 iff output k of the instance of ingress component c at v is activated.
$u_{a,v,v',l}$	1 iff the link is used for the path created for arc a with source and destination on v and v'.

The reduction can be performed in polynomial time in the size of the problem input. Therefore, our problem is NP-hard. From that, we can conclude that our problem is NP-complete.

IV. MIXED-INTEGER PROGRAM FORMULATION

In this section, we formalize the SPRING problem for heterogeneous services as a mixed-integer program (MIP). All constraints and objective functions in this formulation are linear or can be linearized. As the link and node resource demands are also specified using piecewise linear functions, we are dealing with a mixed-integer linear program.

Tables I and II show an overview of the binary and continuous decision variables in the MIP, respectively.

We define a preset constant $m^*_{c,v,\mathrm{ver}}$ to capture previous embeddings of components based on the input parameters; for every component c that was previously embedded into node v with version ver, represented by a tuple $(c, v, \mathrm{ver}) \in P_T$ (Section III-E), we set $m^*_{c,v,\mathrm{ver}}$ to 1. For all other components, nodes, and versions, we set it to 0. $\mathcal{M}$ represents a constant that is sufficiently large, used in the so-called Big-M formulations. $\mathcal{C}_{\mathrm{SRC}}, \mathcal{C}_{\mathrm{FIX}}, \mathcal{C}_{\mathrm{ING}} \subset \mathcal{C}$ represent the source, fixed, and ingress components, respectively. We indicate all the normal components that are not source or fixed components with $\mathcal{C}_N$.

A. Constraints

We assign fixed components and sources to their predefined locations (Constr. 2, 3). The data rate of each source is assigned to its output (Constr. 4).

$$\forall c \in \mathcal{C}_{\mathrm{SRC}}, \forall v \in V : x_{c,v} = \begin{cases} 1 & \exists (v, c, \mu) \in \mathcal{S} \\ 0 & \text{else} \end{cases} \tag{2}$$

$$\forall v \in V, \forall c \in \mathcal{C}_{\mathrm{FIX}} : \quad x_{c,v} = \begin{cases} 1 & \text{if } \exists (c, v) \in \mathcal{X} \\ 0 & \text{else} \end{cases} \tag{3}$$

TABLE II
CONTINUOUS DECISION VARIABLES

Variable	Definition
$\mathrm{cpu}_{c,v}$, $\mathrm{gpu}_{c,v}$	CPU, GPU demand of the instance of c if mapped to v.
$\mathrm{time}_{c,v}$	Processing time of the instance of c if mapped to v.
$p_{c,v,\mathrm{ver}}$	Potential CPU demand of c at v for version ver.
$g_{c,v}$	Potential GPU demand of c at v, defined for its GPU-accelerated instances.
$t_{c,v,\mathrm{ver}}$	Potential processing time of c at v using version ver.
$s_{c,v}$	Total CPU and GPU resource cost of c on v per time unit.
$\mathrm{in}_{c,v}$	Vector of length n_c^{in} of data rates at inputs of the instance of c at v, or an all-zero vector
$\mathrm{out}_{c,v}$	Vector of length n_c^{out} of data rates at outputs of the instance of c at v, or an all-zero vector
$o_{c,v}$	Vector of length n_c^{out} of potential data rates at outputs of the instance of ingress component c at v
$\mathrm{dr}^e_{a,v,v'}$	Data rate of the edge corresponding to an arc a that connects an instance of c at v to an instance of c' at v'.
$\mathrm{dr}^l_{a,v,v',l}$	Data rate on link l corresponding to an arc a that connects an instance of c at v to an instance of c' at v'.

$$\forall c \in \mathcal{C}_{\mathrm{SRC}}, \forall v \in V : \mathrm{out}_{c,v} = \begin{cases} \mu & \exists (v,c,\mu) \in \mathcal{S} \\ 0 & \text{else} \end{cases} \tag{4}$$

We track the added/removed/modified instances (Constr. 5). If an instance is created, the right number of inputs (Constr. 6) and outputs (Constr. 7) are created for it (we represent the k-th element of a vector w by $(w)_k$). At most one instance of each component can be mapped to a node (Constr. 8, 9).

$$\forall c \in \mathcal{C}_N, \forall v \in V : \delta_{c,v} = \begin{cases} m_{c,v,\mathrm{ver}} & \text{if } m^*_{c,v,\mathrm{ver}} = 0 \\ 1 - m_{c,v,\mathrm{ver}} & \text{if } m^*_{c,v,\mathrm{ver}} = 1 \end{cases} \tag{5}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall k \in [1, n_c^{\mathrm{in}}] : \quad (\mathrm{in}_{c,v})_k \leq \mathcal{M} \cdot x_{c,v} \tag{6}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall k \in [1, n_c^{\mathrm{out}}] : \quad (\mathrm{out}_{c,v})_k \leq \mathcal{M} \cdot x_{c,v} \tag{7}$$

$$\forall c \in \mathcal{C}_N, \forall v \in V : \sum_{\mathrm{ver} \in \mathrm{VER}} m_{c,v,\mathrm{ver}} \leq 1 \tag{8}$$

$$\forall c \in \mathcal{C}_N, \forall v \in V :$$

$$0 \leq |\,\mathrm{VER}\,| \cdot x_{c,v} - \sum_{\mathrm{ver} \in \mathrm{VER}} m_{c,v,\mathrm{ver}} \leq |\,\mathrm{VER}\,| - 1 \tag{9}$$

The data rate entering an instance determines its outgoing data rate (Constr. 10). The data rates are set only if the instance is mapped to a certain node. We assume all deployment versions for an instance have the same function for calculating the outgoing data rate. For ingress components, only one of the outputs can have a data rate (Constr. 11–13).

$$\forall c \in \mathcal{C}_N \setminus \mathcal{C}_{\mathrm{ING}}, \forall v \in V :$$

$$\mathrm{out}_{c,v} = \mathrm{fout}_c(\mathrm{in}_{c,v}) - (1 - x_{c,v}) \cdot \mathrm{fout}_c(\underline{0}) \tag{10}$$

$$\forall c \in \mathcal{C}_{\mathrm{ING}}, \forall v \in V :$$

$$o_{c,v} = \mathrm{fout}_c(\mathrm{in}_{c,v}) - (1 - x_{c,v}) \cdot \mathrm{fout}_c(\underline{0}) \tag{11}$$

$$\mathrm{out}_{c,v} = r_{c,v,k} \cdot o_{c,v} \tag{12}$$

$$\sum_{k \in [1, n_c^{\mathrm{out}}]} r_{c,v,k} = 1 \tag{13}$$

We assign data rate to each input of the instances on each node as the sum of data rates of overlay edges that end in that input (Constr. 14). Similarly, we assign data rate to the outputs of the instances (Constr. 15).

$$\forall c \in \mathcal{C}, \forall v \in V, \forall k \in [1, n_c^{\mathrm{in}}] :$$

$$(\mathrm{in}_{c,v})_k = \sum_{\substack{a \in \mathcal{A} \text{ ends in input } k \text{ of } \mathrm{src}_a(a), \\ v' \in V}} \mathrm{dr}^e_{a,v',v} \tag{14}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall k \in [1, n_c^{\mathrm{out}}] :$$

$$(\mathrm{out}_{c,v})_k = \sum_{\substack{a \in \mathcal{A} \text{ starts at output } k \text{ of } \mathrm{src}_a(a), \\ v' \in V}} \mathrm{dr}^e_{a,v,v'} \tag{15}$$

The data rates of the edges are mapped to network links, ensuring flow conservation over the path(s) (Constr. 16). The total delay of the used network links cannot exceed the maximum delay (Const. 17–19).

$$\forall a \in \mathcal{A}, \forall v, v_1, v_2 \in V :$$

$$\sum_{vv' \in L} \mathrm{dr}^l_{a,v_1,v_2,vv'} - \sum_{v'v \in L} \mathrm{dr}^l_{a,v_1,v_2,v'v} = \begin{cases} 0 & \text{if } v \neq v_1, v \neq v_2 \\ 0 & \text{if } v = v_1 = v_2 \\ \mathrm{dr}^e_{a,v_1,v_2} & \text{if } v = v_1, v_1 \neq v_2, a \in \mathcal{A}_{\mathrm{up}} \end{cases} \tag{16}$$

$$\forall l \in L : \mathrm{dr}^l_{a,v_1,v_2,l} \leq \mathcal{M} \cdot u_{a,v_1,v_2,l} \tag{17}$$

$$\forall l \in L : u_{a,v_1,v_2,l} \leq \mathrm{dr}^l_{a,v_1,v_2,l} \tag{18}$$

$$\sum_{l \in L} u_{a,v_1,v_2,l} \cdot d(l) \leq d_a^{\max} \tag{19}$$

The data rate on inputs of each instance determines the *minimum* resource demands of it. For selecting the right piece of the piecewise linear function, we determine the load level for every potential deployment version. For this, we compare the sum of all input data rates of the instance to the predefined upper and lower bounds for each load level (Constr. 20, 21). Exactly one load level is indicated as the right one (Constr. 22, 23). Based on the load level, we calculate the *potential* minimum CPU demand of each potential version (Constr. 24). We repeat the same process for determining the *potential* minimum GPU resource demands ($g_{c,v}$). We omit the corresponding constraints due to space limitations.

$$\forall c \in \mathcal{C} \setminus \mathcal{C}_{\mathrm{SRC}}, \forall v \in V, \forall \,\mathrm{ver} \in \mathrm{VER}, \forall \,\mathrm{lev} \in \mathrm{LEV} :$$

$$\mathrm{ub}_c^{\mathrm{lev}}(\mathrm{ver}) - \sum_{k \in [1, n_c^{\mathrm{in}}]} (\mathrm{in}_{c,v})_k \leq \mathcal{M} \cdot u^{\mathrm{cpu}}_{c,v,\mathrm{ver},\mathrm{lev}} \tag{20}$$

$$\sum_{k \in [1, n_c^{\mathrm{in}}]} (\mathrm{in}_{c,v})_k - \mathrm{lb}_c^{\mathrm{lev}}(\mathrm{ver}) \leq \mathcal{M} \cdot l^{\mathrm{cpu}}_{c,v,\mathrm{ver},\mathrm{lev}} \tag{21}$$

$$\forall c \in \mathcal{C} \setminus \mathcal{C}_{\mathrm{SRC}}, \forall v \in V, \forall \,\mathrm{ver} \in \mathrm{VER} :$$

$$\sum_{\mathrm{lev} \in \mathrm{LEV}} (l^{\mathrm{cpu}}_{c,v,\mathrm{ver},\mathrm{lev}} + u^{\mathrm{cpu}}_{c,v,\mathrm{ver},\mathrm{lev}}) = |\,\mathrm{LEV}\,| + 1 \tag{22}$$

$$\forall c \in \mathcal{C} \setminus \mathcal{C}_{\mathrm{SRC}}, \forall v \in V, \forall \,\mathrm{ver} \in \mathrm{VER}, \forall \,\mathrm{lev} \in \mathrm{LEV} :$$

$$0 \leq l^{\mathrm{cpu}}_{c,v,\mathrm{ver},\mathrm{lev}} + u^{\mathrm{cpu}}_{c,v,\mathrm{ver},\mathrm{lev}} - 2 \cdot b^{\mathrm{cpu}}_{c,v,\mathrm{ver},\mathrm{lev}} \leq 1 \tag{23}$$

$$p_{c,v,\mathrm{ver}} + \mathcal{M} \cdot (1 - b^{\mathrm{cpu}}_{c,v,\mathrm{ver},\mathrm{lev}}) \geq$$

$$\mathrm{fcpu}_c^{\mathrm{lev}}(\mathrm{ver}, \mathrm{in}_{c,v}) - (1 - m_{c,v,\mathrm{ver}}) \cdot \mathrm{ccon}_c^{\mathrm{lev}}(\mathrm{ver}) \tag{24}$$

Among the potential versions, only one version may be mapped on a potential location. The resource demands of the optimal versions of components (according to the objectives) are assigned as their final resource demand on the optimal nodes (Constr. 25, 26). Link and node resource consumption cannot be larger than the capacity (Constr. 27–29).

$$\forall c \in \mathcal{C}_N, \forall v \in V : \mathrm{cpu}_{c,v} = \sum_{\mathrm{ver} \in \mathrm{VER}} p_{c,v,\mathrm{ver}} \cdot m_{c,v,\mathrm{ver}} \quad (25)$$

$$\forall c \in \mathcal{C}_N, \forall v \in V : \mathrm{gpu}_{c,v} = g_{c,v} \cdot m_{c,v,\mathrm{ACC}} \quad (26)$$

$$\forall c \in \mathcal{C}, \forall v \in V : \sum_{c \in \mathcal{C}} \mathrm{cpu}_{c,v} \leq \mathrm{cap}_{\mathrm{cpu}}(v) \quad (27)$$

$$\forall c \in \mathcal{C}, \forall v \in V : \sum_{c \in \mathcal{C}} \mathrm{gpu}_{c,v} \leq \mathrm{cap}_{\mathrm{gpu}}(v) \quad (28)$$

$$\forall l \in L : \sum_{a \in \mathcal{A}, v, v' \in V} \mathrm{dr}^{l}_{a,v,v',l} \leq \mathrm{cap}(l) \quad (29)$$

The *potential* maximum processing time of each instance of each component is calculated using the given functions, if a version is mapped to a node (Constr. 30,31). The processing time of the selected version is assigned as the final processing time of it on the target node (Constr. 32).

$$\forall c \in \mathcal{C}_N, \forall v \in V, \forall \,\mathrm{ver} \in \mathrm{VER} :$$
$$t_{c,v,\mathrm{ver}} = \mathrm{fpt}_c(\mathrm{ver}, \sum_{k \in [1, n_c^{\mathrm{in}}]} (\mathrm{in}_{c,v})_k)$$
$$-(1 - m_{c,v,\mathrm{ver}}) \cdot \mathrm{fpt}_c(\mathrm{ver}, \sum_{k \in [1, n_c^{\mathrm{in}}]} (\mathrm{in}_{c,v})_k) \quad (30)$$

$$t_{c,v,\mathrm{ver}} \leq \mathcal{M} \cdot m_{c,v,\mathrm{ver}} \quad (31)$$

$$\forall c \in \mathcal{C}_N, \forall v \in V : \mathrm{time}_{c,v} = \sum_{\mathrm{ver} \in \mathrm{VER}} t_{c,v,\mathrm{ver}} \quad (32)$$

We calculate the total CPU and GPU resource cost of every embedded instance on their target nodes per time unit (Constr. 33). By multiplying this value and the processing time of each component, the total resource usage cost of each component on each node can be calculated.

$$\forall c \in \mathcal{C}_N, \forall v \in V :$$
$$s_{c,v} = \mathrm{cpu}_{c,v} \cdot \mathrm{cost}_{\mathrm{cpu}}(v) + \mathrm{gpu}_{c,v} \cdot \mathrm{cost}_{\mathrm{gpu}}(v) \quad (33)$$

B. Objectives

We define the following objective functions for the MIP:

- obj_1: Minimize the total compute resource cost

$$\text{min.} \sum_{c \in \mathcal{C}, v \in V} s_{c,v}$$

- obj_2: Minimize the total processing time

$$\text{min.} \sum_{c \in \mathcal{C}, v \in V} \mathrm{time}_{c,v}$$

- obj_3: Minimize network resource consumption

$$\text{min.} \sum_{a \in \mathcal{A}, v, v' \in V, l \in L} \mathrm{dr}^{l}_{a,v,v',l}$$

- obj_4: Minimize the number of added, removed, modified instances

$$\text{min.} \sum_{c \in \mathcal{C}, v \in V} \delta_{c,v}$$

Algorithm 1 Main procedure of the heuristic algorithm

```
1:  if ∃G_OL(T) with T ∉ 𝒯 then
2:      remove G_OL(T)
3:  for all T ∈ 𝒯 do
4:      if ∄G_OL(T) then
5:          create empty overlay G_OL(T)
6:      for all (c, v, λ) ∈ S_T and (c, v) ∈ X_T do
7:          if ∄i ∈ I_OL then
8:              create instance i ∈ I_OL
9:          set/update output data rate of i
10:     if a source instance i with no data rate exists then
11:         remove i
12:     for all i ∈ I_OL in topological order do
13:         c : component corresponding to i
14:         IN: sum of data rates on all inputs of i
15:         if n_c^in > 0 and IN = 0 then
16:             remove i and go to next iteration
17:         compute output data rates of i
18:         for all output k of i do
19:             set/update output data rate of i
```

To combine the benefits of using these objective functions, we define the following lexicographical combination of the objectives: min. $w_1 \cdot \mathrm{obj}_1 + w_2 \cdot \mathrm{obj}_2 + w_3 \cdot \mathrm{obj}_3 + w_4 \cdot \mathrm{obj}_4$. Ideally, the weights $w_1, \ldots, w_4$ should be selected such that the range of the values different objective can take do not overlap and each objective has a clear priority. The desired priority among these objectives depends on the use case.

V. Heuristic Approach

In this section, we present a scalable heuristic algorithm that finds fast solutions and can be used for large scenarios.

Algorithm 1 shows an overview of the main procedure. We describe the important steps in the rest of this section.

The templates can be sorted beforehand, e.g., according to the total input data rate from their sources. For new templates, we create an empty overlay (lines 4–5). We then process the source and fixed instances for the template (lines 6–9).

Setting the output data rate of an instance i results in creating/updating outgoing edges from the output(s) of i as well as the inputs of the instances where these edge are destined. For this, the algorithm must decide how many instances of which versions of the subsequent component need to be created on which nodes. We describe this with an example.

In Figure 5, while setting the output data of instances of S, the algorithm needs to create at least one instance of SRV. For every deployment version of SRV (in this example, SRV has only a VM deployment version), it looks for potential nodes. As one of objectives (described in Section III-G) is to minimize the number of added/removed instances, the algorithm takes a greedy decision; it tries to create an instance of SRV with the maximum possible input data rate. If the total outgoing data rate of S is higher than the upper bound of the

load level HIG for SRV, it creates an instance of SRV and sets its input data rate to this highest possible value. For the remaining data rate from S, it creates additional instances of SRV in the same way, until there is no more traffic left to be forwarded to a SRV instance.

At the same time, it selects the candidate nodes that can host the created instances. These nodes should have enough capacity and there should be a path to them from the node where S is located. The links over the path should have enough capacity and a total delay not larger than the maximum tolerable delay. Locations with an existing instance of SRV from a previous embedding are also considered. Among the candidate nodes we can now select the best option, considering the resulting processing time of the deployment version at that load level and resource usage cost on that node.

We then iterate over the instances of the overlay in a topological order (Line 12). That is, each instance i is processed only after all instances that have an edge to i (as specified in the template) have already been processed. The first instances that are processed are the instances after the source instance that are created while setting the output of the source instance in line 9 (instances of SRV in the previous example).

Next, we compute and propagate the data rates from outputs of the current instance towards other components (lines 17–19). In the previous example, this is the data rate towards DPI and CHE. If the data rate needs to be increased (i.e., if there are no previous embeddings of DPI or CHE, or if the previous data rate was less than the computed data rate at this step), we proceed in the same way as described for outputs of the source instance. If the data rate needs to be decreased, a similar process is required to select the most suitable instances of the subsequent components that should get a lower data rate. To limit the range of required modifications, we select an outgoing edge that has the smallest data rate larger than or equal to the data rate that we need to decrease.

If the current instance is an instance of an ingress component, we first calculate the most suitable embedding for all of its outgoing branches. Then, comparing the total cost of each branch (calculated as the total resource usage cost during the total processing time), the cheapest branch is added to the overlay and the rest of them are removed.

VI. Evaluation

We have implemented both the optimization and the heuristic approaches as Python programs. For solving the MIP, we have used the Gurobi Optimizer 8.0.1[2]. We have used the benchmarks for the Virtual Network Mapping Problem (VNMP)[3] to build the topology of our substrate networks. We present the results of two different experiments to compare the how the heuristic algorithm solves the problem compared to the MIP approach and to show the scalability of the heuristic.

To achieve results in a reasonable time, we have used a simple template (T_1) consisting of a source component and a DPI in different versions. We have set the resource demands and the processing times based on the findings of Araújo et al. [17], who have analyzed DPI VNFs with and without GPU-acceleration. We assume an ACC version performs 20 times faster than the VM. We assume other resources like memory, buffer, or disk space can be adjusted as needed similar to the compute resources. We have used the smallest substrate network from VNMP with 20 nodes and 44 links, with uniform capacities and resource costs over the network. We have set the GPU capacity of each node to 5 times less than its CPU capacity and the GPU usage cost per time unit on the same node to 50 times more than the CPU usage cost[4].

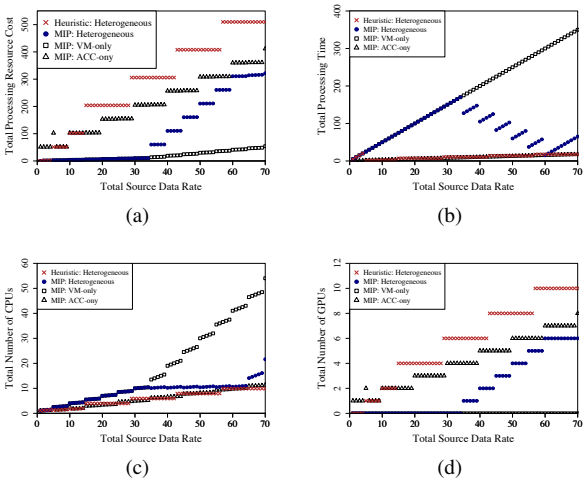


Fig. 6. Results of the first set of experiments with template T_1 including a source and a multi-version DPI

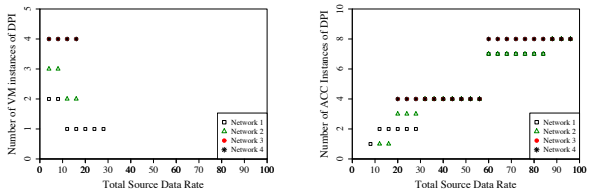
Fig. 6 shows the results of the first set of experiments. In these experiments, we have increased the data rate flowing from the only source instance of the template from 1 to 70. We have captured the values of different metrics for 4 cases: (i) Heuristic approach, considering both versions of DPI, (ii) MIP approach, considering both versions of DPI, (iii) MIP approach, considering only the VM version of DPI, and (iv) MIP approach, considering only the ACC version of DPI. To see the behavior of the algorithms in different load situations, we have embedded the template with different source data rates *without* considering its previous embeddings.

The heuristic approach starts selecting accelerated versions of DPI early on, because of its greedy decision process that tries to push as much of the input data rate as possible to the first instance it creates at each step. As the instances are created one by one without considering the whole template, the required instances in the next steps cannot be considered. For this reason, the heuristic creates embeddings that are more costly than the ACC-only experiments with the MIP approach (Fig. 6a). In exchange, the created templates have a

[2]http://www.gurobi.com/

[3]https://www.ac.tuwien.ac.at/files/resources/instances/vnmp

[4]These ratios *roughly* follow the Amazon EC2 On-Demand Pricing model. Concrete information about resource unit prices cannot be extracted from these models, as the pricing model is based on predefined instances with a certain group of resources reserved for them.



(a) (b)

Fig. 7. Results of the second set of experiments with the multi-structure template T_2 including four sources

very low processing time, making the approach favorable for time-sensitive services (Fig. 6b).

The MIP approach creates more balanced results, favoring low-cost solutions longer. The cost of the solutions lie between those of the VM-only and ACC-only experiments. Above the source data rate of 35, the MIP approach starts creating more ACC versions of the DPI (in addition to VM versions) as the load increases, which consume GPUs (Fig. 6d). This results in the gradual increase in the cost (Fig. 6a), decrease in the processing time (Fig. 6b), and only minor increase in the total number of CPUs (Fig. 6c).

In the second set of experiments, we show the heuristic results on larger substrate networks. We have used a multi-structure template (T_2), where the first option for deploying a DPI is a VM and the second option is an ACC version accompanied by an auxiliary VM. We have used Network 1 with 20 nodes and 44 links, Network 2 with 50 nodes and 124 links, and Network 3 with 100 nodes and 230 links (from VNMP instances). We have set the node and link capacities of Network 2–4 to 2.5, 5, and 10 times more than those of Network 1, respectively. The templates have 4 source locations from 4 distant locations in each substrate network, each with data rates increasing from 1 to 25. Because of the larger distances in larger networks and delay constraints of template arcs, each source location requires dedicated instances in Network 3 and 4, whereas the instances can be shared among sources in Network 1 and 2. As shown in Fig. 7, this results in a larger number of instances in Network 3 and 4. Results of Network 3 and 4 are overlapping. In all substrate networks, with lower source data rates, the first deployment option of the DPI (VM) is selected. As load increases, more and more of the second option of the DPI (ACC with an auxiliary VM) are created.

For the largest instances in these experiments, the heuristic finds a solution in less than 5 seconds. The MIP approach requires several minutes for the smallest instance and cannot find solutions to large instances in a reasonable time.

VII. Conclusion

In this paper, we have shown the feasibility of defining heterogeneous services, including components with different deployment options. We have used optimization and heuristic approaches for joint scaling, placement, and routing decisions for these services. Our approaches can create low-cost embeddings for low-load situations and with increasing load, switch to hardware-accelerated versions of service components for lower processing times. Using our approaches, different resource types in heterogeneous infrastructures can be used efficiently to get suitable resource costs and processing times.

Acknowledgment

We would like to thank Mr. Haitham Afifi for the discussions that helped progressing with our MIP formulation.

References

[1] N. F. S. de Sousa, D. A. L. Perez, R. V. Rosa, M. A. Santos, and C. E. Rothenberg, "Network Service Orchestration: A Survey," *arXiv preprint arXiv:1803.06596*, 2018.

[2] S. Dräxler, H. Karl, H. R. Kouchaksaraei, A. Machwe, C. Dent-Young, K. Katsalis, and K. Samdanis, "5G OS: Control and Orchestration of Services on Multi-Domain Heterogeneous 5G Infrastructures," in *2018 European Conference on Networks and Communications (EuCNC)*, June 2018, pp. 1–9.

[3] S. Dräxler, H. Karl, and Z. A. Mann, "Jasper: Joint optimization of scaling, placement, and routing of virtual network services," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2018.

[4] A. Fischer, J. F. Botero, M. T. Beck, H. De Meer, and X. Hesselbach, "Virtual network embedding: A survey," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, pp. 1888–1906, 2013.

[5] J. Li, N. Zhang, Q. Ye, W. Shi, W. Zhuang, and X. Shen, "Joint resource allocation and online virtual network embedding for 5g networks," in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, Dec 2017, pp. 1–6.

[6] A. Baumgartner, V. S. Reddy, and T. Bauschert, "Mobile core network virtualization: A model for combined virtual core network function placement and topology optimization," in *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*. IEEE, 2015, pp. 1–9.

[7] P. T. Endo, A. V. de Almeida Palhares, N. N. Pereira, G. E. Goncalves, D. Sadok, J. Kelner, B. Melander, and J. Mangs, "Resource allocation for distributed cloud: concepts and research challenges," *IEEE Network*, vol. 25, no. 4, pp. 42–46, July 2011.

[8] J. G. Herrera and J. F. Botero, "Resource allocation in nfv: A comprehensive survey," *IEEE TNSM*, vol. 13, no. 3, pp. 518–532, 2016.

[9] Z. A. Mann, "Interplay of virtual machine selection and virtual machine placement," in *Proceedings of the 5th European Conference on Service-Oriented and Cloud Computing*, 2016, pp. 137–151.

[10] E. Ahvar, S. Ahvar, N. Crespi, J. Garcia-Alfaro, and Z. Á. Mann, "NACER: a network-aware cost-efficient resource allocation method for processing-intensive tasks in distributed clouds," in *Proceedings of the 14th IEEE International Symposium on Network Computing and Applications*, 2015, pp. 90–97.

[11] M. Keller, H. Karl, and C. Robbert, "Template embedding: Using application architecture to allocate resources in distributed clouds," in *IEEE/ACM UCC*, 2014.

[12] S. Khebbache, M. Hadji, and D. Zeghlache, "Virtualized network functions chaining and routing algorithms," *Computer Networks*, vol. 114, pp. 95–110, feb 2017.

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

[14] S. Ahvar, H. P. Phyu, and R. Glitho, "CCVP: Cost-efficient Centrality-based VNF Placement and Chaining Algorithm for Network Service Provisioning," in *IEEE NetSoft*. IEEE, jul 2017, pp. 1–9.

[15] M. C. Luizelli, L. R. Bays, L. S. Buriol, M. P. Barcellos, and L. P. Gaspary, "Piecing together the NFV provisioning puzzle: Efficient placement and chaining of virtual network functions," in *IFIP/IEEE IM*, 2015, pp. 98–106.

[16] A. Mehta and E. Elmroth, "Distributed cost-optimized placement for latency-critical applications in heterogeneous environments," in *2018 IEEE International Conference on Autonomic Computing (ICAC)*, Sept 2018, pp. 121–130.

[17] I. M. Araújo, C. Natalino, A. L. Santana, and D. L. Cardoso, "Accelerating vnf-based deep packet inspection with the use of gpus," in *2018 20th International Conference on Transparent Optical Networks (ICTON)*, July 2018, pp. 1–4.