SDN Policy-Driven Service Chain Placement in OpenStack

Manuel Stein*, Michael Scharf† and Volker Hilt*
*Nokia Bell Labs, {first.last}@nokia-bell-labs.com
†Nokia, Email: michael.scharf@nokia.com

***Abstract*—Network functions virtualization requires automatic deployment and scaling of components. This raises the question of where to place instances of a function, for instance in the OpenStack cloud system. Data plane functions can forward large amounts of traffic. In this case, network-aware placement can avoid an inefficient use of host bandwidth, and a chain of functions can benefit from co-locating instances on a host. However, a practical challenge is that the bandwidth utilization or traffic demand matrix is not always known before the deployment of an instance. A promising remedy is to leverage existing Software Defined Networking (SDN) policies to derive connectivity weights between components.**

In this paper, we present this novel solution to the online instance placement problem. We have developed an extension of the OpenStack scheduler that uses SDN forwarding policies to rank potential hosts. For a given type of virtual machine, the corresponding forwarding policies can be retrieved from an SDN controller prior to the placement decision. Our prototype identifies potential communication peers and weighs the forwarding rules to prefer hosts that already run communication peers. We present heuristics for such weighing, and we also discuss limitations of the approach. A testbed implementation proofs that even in a simple example our solution can double the service chain throughput.

***Index Terms*—OpenStack, placement, service chain, SDN, NFV**

I. Introduction

Network Functions Virtualization (NFV) [1] evolves network services from appliances to cloud computing applications that can be scaled on-demand across distributed cloud locations, using virtual execution environments such as virtual machines (VMs), containers, or unikernels. Examples for virtualized network functions (VNF) are servers that terminate user traffic or functions on the traffic, e.g. routing, load-balancing, firewalls, echo cancellation, or traffic analysis. A forwarding path of traffic among a series of VNFs is called service chain. On-boarding such network functions to cloud computing removes the need to setup physical appliances.

One question is where to place VNF components. Given the need for elastic scaling, this is an online placement problem. State-of-the-art cloud computing solutions such as OpenStack [2] manage the computing resources. However, most cloud platforms do not take into account the network when determining where to place a new instance.

In NFV, large amounts of data plane traffic can traverse multiple VNFs within a service chain. If VNFs are spread across hosts, the host network interface could be traversed multiple times and become a bottleneck. If the host scheduler, e. g. in OpenStack, could take placement decisions based on instance communication, such traffic could be localized. Co-location of service chain functions reduces the need to traverse multiple network interfaces and also simplifies traffic isolation.

However, a fundamental challenge is to know the traffic demand between VNF instances *in advance*. The scheduler of a cloud computing system must place an instance before its traffic can be measured, and moving an instance later to another host is often impossible, since this could disrupt the service. Both the user traffic demand and the number of VNFs in resource pools can change over time, i. e., static bandwidth reservations won't be precise. It is in practice challenging to characterize the communication of a VM before it is actually deployed. The question is therefore how to optimize network-aware online placement decisions without requiring precise predictions of traffic patterns.

One potential source of information is the virtual network configuration in the cloud system, which is typically configured prior to launching instances. Advanced Software Defined Networking (SDN) controllers offer a rich set of policies how to interconnect instances, e. g., to separate virtual networks and to provide firewall functionality. These SDN policies include many hints on connectivity requirements, such as transport protocol ports, QoS configuration, and redirection rules that are specific to applications. To the best of the knowledge of the authors, the use of this information to optimize online placement of instances has not been studied so far.

This paper investigates how connectivity between VMs can be inferred from SDN policies, and how this information can be used a priori to heuristically improve the placement of a new VM. We demonstrate that by leveraging and correlating information that already exists in separate systems the network throughput of service chains can be significantly improved.

The remainder of this paper is structured as follows. Section II reviews the open source cloud computing software OpenStack and its instance scheduler. We also introduce an example for an advanced SDN controller for data centers. In Section III, we analyze how SDN policies can be considered during instance placement, and we introduce our prototype for policy-driven placement in OpenStack. Section IV reports initial measurement results from a testbed setup. Related work on network-aware placement is summarized in Section V. Finally, Section VI concludes this paper.

978-3-901882-89-0 @2017 IFIP

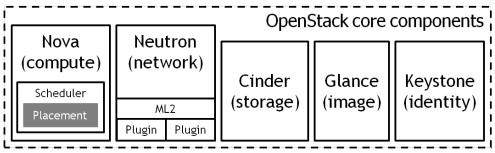

Fig. 1: Overview of important OpenStack components

II. Cloud Platforms and SDN

A. OpenStack cloud platform

Infrastructure-as-a-Service (IaaS) platforms offer computing, storage, and network resources on-demand. OpenStack [2] is widely used and is considered a promising solution for NFV infrastructures. OpenStack provides an ecosystem of core components for traditional IaaS and a number of additions for more sophisticated services. Figure 1 presents the OpenStack core component architecture of computing (*Nova*), networking (*Neutron*), block storage (*Cinder*), image service (*Glance*) and identity (*Keystone*). A cloud computing platform can be built by all or a subset of these components.

All components use Python and share a set of common functions that generalize authentication, messaging and remote procedure calls, database access, request context, and many more. Each component provides at least one agent that exposes the component's Application Programming Interfaces (APIs) based on Representational State Transfer (REST). The code is designed to allow for proprietary extensions by templates.

The OpenStack community releases a new version every six months. There is an ongoing effort to modularize resource tracking and scheduling in order to include resources other than compute hosts or to externalize host selection completely. This work uses the "Kilo" release published in April 2015.

B. Instance placement in OpenStack

When a new instance is started, the resource management in a cloud platform has to select an appropriate host. Because the primary target is to find sufficient available computing capacity, the selection is often regarded a host management task rather than networking or storage management. In OpenStack, host selection is part of compute resource management ("Nova"). The scheduler typically considers compute host resources, capacity, and availability. The placement logic selects the most suitable host to deploy each instance individually according to the requested resources by that instance only. As a result, instance placement is not based on the overall application deployment or its communication requirements.

In OpenStack, the commonly used placement logic is the *filter scheduler* [3]. The filter scheduler is a two-step process of *filtering* and *weighting* to find the most suitable host for a VM request. These two steps are also visualized in figure 2.

The logic starts with an instance request and a list of available hosts. The configured filters sequentially process the list and evaluate for each host in the remaining set whether it is able to accommodate the new instance. For example, some filters determine whether a node has enough residual compute resource capacity ("ComputeFilter", "RamFilter").

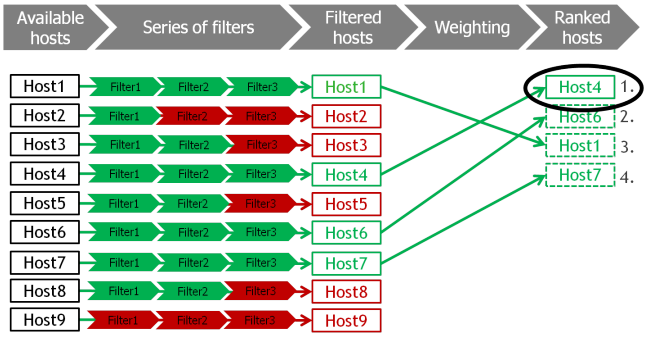

Fig. 2: Default instance scheduling in OpenStack "Kilo"

Host meta-data and instance request meta-data can be used as additional matching criteria. As such, host capabilities like SR-IOV or solid state disks (SSDs) can also be matched against requirements attached to the VM's flavor or image. In addition, affinity or anti-affinity filters exist.

The set of hosts that passes all configured filters is then processed by one or more weighters. For example, the *BaseHostWeigher* template in the host scheduler agent performs the host ranking. A weighter assigns a normalized weight to each host. Multiple weights can be combined to a weighted sum. Based on the weight score, the scheduler ranks the list of valid hosts, whereby the largest weight identifies the best suitable host, as illustrated in Fig. 2. In the default configuration, the filter scheduler only uses memory weighting ("RAMWeigher") that ranks by most residual Random Access Memory (RAM) capacity. Affinity or anti-affinity weighting is also available, but it can only use a fixed weight for peer instances, and therefore it can not be used to differentiate between different communication patterns per peer instance.

C. SDN policy system

The *Neutron* component OpenStack enables the use of a Software Defined Networking (SDN) solution to provide the connectivity between instances. There are several open source and closed source SDN controllers that integrate into cloud computig platforms. In this paper, we consider the *Nuage Virtualized Services Platform* (VSP) as an example for a widely used data center SDN solution. Other SDN platforms provide similar intent-based connectivity policies, and our approach is also applicable in those cases.

The Nuage VSP provides a policy directory (*Virtual Service Directory*), SDN controllers (*Virtual Service Controller*) and OpenVSwitch-based software switches (*Virtual Routing Service*). Policies are stored and retrieved from the directory. Upon deployment of an instance, they get compiled to forwarding rules that are configured on the switches. The policy directory exposes a REST-ful API that one can access using a Python software development kit (*Virtualized Service Platform SDK*). The SDK provides notifications that push changes in the policy directory to notification subscribers. This enables clients to learn and process all SDN policies for a VM before they actually are created by OpenStack.

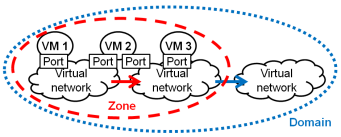

Fig. 3: Example for an SDN policy system

As depicted in figure. 3, the SDN policies distinguish hierarchically between *zones* and *domains* to enable the setup of multi-tiered virtual networks. Instances can be assigned to them, and forwarding rules can be defined to control which VM is allowed to communicate with whom. The policies are described by the typical notion of *ingress* (forwarding traffic into the network) and *egress* (forwarding traffic out of the network). In order to have traffic forwarded from a source to a destination, a pair of matching ingress/egress policies is required. Each of the SDN policies applies to a *location* and a *network*. *Ingress* policies describe traffic from an origin *location* into a *network* and *egress* policies describe traffic out of some *network* into a destination *location*.

Table I summarizes this terminology. Both location and network identifiers can be address-based, group-based, or hierarchical. An example for address-based classifiers are IP address ranges or autonomous systems, while group-based classifiers define sets of endpoints. Classifiers can be hierarchically organized into zones and domains. While such policy classifiers identify endpoints, policy criteria can further restrict to specific traffic characteristics, namely Ethernet type, IP protocol, DSCP, and source/destination port. Note that the policy criteria do not include source/destination addresses, because these come with the endpoint identified by classifiers of a policy. From a high-level point of view, the SDN policy system is therefore a sophisticated combination of firewall and Quality-of-Service (QoS) rules for multi-tier environments.

D. Challenges for network-aware placement in OpenStack

The architecture and resource management in OpenStack and its host selection process result in significant challenges for optimizing the placements of service chaining in NFV.

The first limitation is that the OpenStack host selection process is performed for each single instance individually, based on a snapshot of the available host resources. OpenStack is designed for online per-instance placement. Each VM is managed as individual resource. Resource allocations are reported by hosts independent from the selection process. This design limits the implementation complexity and evades the risk of blocking, and the filtering and weighting method also scales. But in this software design a coordinated optimization of placing several instances is very challenging.

	Source	**Destination**
Ingress	Location	Network
Egress	Network	Location

TABLE I: Ingress/egress SDN policy viewpoints

Secondly, the architectural split between network and compute management limits the use of non-computing resource metrics. In OpenStack the networking and storage subsystems do not report neither metrics nor configuration to the host manager, which is responsible for selecting the host of an instance. Also, the internal workflow used by OpenStack prevents the use of actual resource utilization as a metric. Reports on the actual utilization would raise scalability issues. Without redesigning the Nova scheduler, placement algorithms can only use static resource allocation data.

A third challenge is that OpenStack Neutron itself has only limited networking capabilities. Neutron can deal with virtual networks, but the API is not designed for service chaining use cases, which may e. g. require per-flow routing. The so-called security groups are the closest approximation to flow descriptions, and there is also an optional Group-Based Policy component in OpenStack. Actual deployments of OpenStack may rely on a separate SDN solution offering more functionality.

Finally, there is the inherent difficulty to predict the communication between instances prior to actually deploying them.

III. SDN Policy-Driven Scheduling

A. Concept of SDN policy-driven placement

Tke key idea of SDN policy-driven scheduling is to leverage existing information about expected connectivity between instances during the placement. Generally, more expressive policies provide more hints. Basic cloud virtual network models only describe subnets and endpoints. This model does not provide much information that could be leveraged for improved placement. This limitation can easily be seen in trivial examples, such as application-layer load balancing. A virtual network model would have the load balancer share a virtual network with all service instances in order to distribute requests, but the instances themselves, despite residing in the same network, don't forward traffic to each other. It would be wrong to assume a strong affiliation among the instances just because they use the same address space or the same virtual network. Instead, intent-based forwarding policies in SDN can express more specifically the intent of communication between a load balancer and a service instance.

SDN policies are typically atomic, i. e., separate rules exist for ingress/egress directions and specify the connectivity of an endpoint only to a group or a network but independent from other endpoints. Address-based and group-based abstraction require the policies to be compiled to find out which traffic would be forwarded to whom. In order to deduce from the policies potential instance affinities, the allowed traffic has to be translated into weights that quantify the affiliations.

It is not until the deployment request that VM subnet and group affiliations determine the tenant and provider policies that apply to the VM interfaces. To identify policies for ranking, host selection requires SDN policies to be set up prior to the VM deployment request. Still, the SDN controller would not compile the policies until the VM starts up. This late binding of policies requires policy-driven placement to analyze

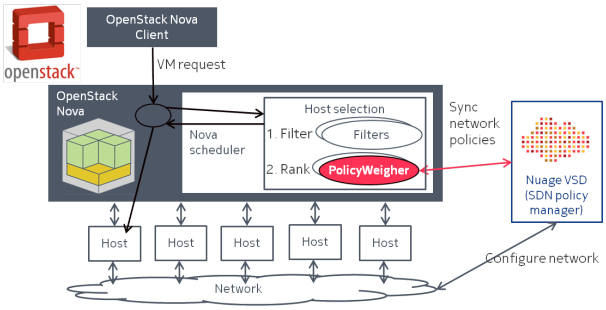

Fig. 4: Prototype architecture for policy-driven placement

the applicable network policies in advance for hints to rank potential hosts with respect to already deployed instances.

In the heuristics used in this paper, the combined traffic that can be expected between the endpoints of a new instance and endpoints of existing instances always gives a positive weight towards co-location. However, this is not always perfect. Instances are often also distributed for reliability, e. g., in a database cluster. For example, an analysis of the SDN policies may conclude to co-locate all peers in a data-replicating application. But then a failure of a host would be detrimental to the reliability concept. So, affinity ranking needs to be complemented by anti-affinity constraints or weights that prevent co-location when necessary. This can be integrated in a multi-objective resource scheduler that employs other sources for anti-affinity (e. g., interference, reliability).

B. Architectural design options

There are three different architectures where the SDN policy compilation and host ranking could be located. One approach would be to implement this logic as part of the application orchestration, which is external to both host management and the SDN solution. This requires the orchestration to keep track of deployed instances and means to control host selection, e. g., by attaching the weights to the deployment request. Such a system architecture is out of scope of this paper as it targets OpenStack only. But this solution may be implemented in systems that maintain a global coordinated scheduling.

Second, the host ranking could be a new service of the SDN controller, which would then evaluate a proposed list of suitable hosts for a new VM. The controller would then have to perform a what-if analysis to determine all networks and groups that the VM will participate in. A standardized interface for providing ranking queries is the Application-Layer Topology Optimization (ALTO) [4].

Finally, the SDN-policy based ranking can be integrated with host management. This least disruptive solution requires timely access to the SDN policies. We implement this approach in our prototype leveraging the Nuage VSP software development kit (SDK). In the following, we proof that it is feasible to derive host ranking weights from SDN policies.

C. Prototype implementation

Figure 4 shows the workflows of our prototype *PolicyWeigher*. Upon initialization, it connects with the Nuage VSD using the SDK to subscribe to push notifications. Upon changes in the SDN topology or the policy set, our PolicyWeigher receives the update and applies it to our cached topology and policy structure. Instance placement requests are created by Nova clients through the RESTful OpenStack Compute API [5], either through "server" (instance) creation or through update requests that would spawn or relocate an instance. The Nova scheduler agent is handed the instance request, retrieves a list of all hosts, processes filters and then hands the remaining candidate host list to our *PolicyWeigher*.

Here, our software can rank the candidate host list with its deployed instances. Once applicable policies are identified, we calculate weights for all potential links to the new instance as described in the subsection on endpoint ranking. When multiple existing instances share a host, we summarize their scores to calculate the total host weight. The result is a ranking of hosts according to where the new VM may communicate with the broadest variety of flows. Once the weights for the host candidate list have been calculated, the weigher normalizes these and applies a weight multiplier that can emphasize the ranking of the *PolicyWeigher* among other weighers, e. g., the *RAMWeigher* or the *DiskWeigher* (cmp. Fig. 2).

D. Endpoint ranking from SDN policies

Our solution focuses on cloud applications or service chains that are composed of more than one instance and that operate on the user data plane, i. e., they process large amounts of traffic. When a new instance joins the deployment, our ranking heuristic will try to provide a preference towards existing instances. Inherently no preference exists if the instance is alone in a service chain. When no networking preferences are taken into account, the OpenStack default *RAMWeigher* would favor a host with the most residual memory, tending towards a *greedy* host selection. The objective of endpoint ranking with network awareness is to turn the host selection into to a *best-fit* with respect to expected communication.

This requires a new weight, which we calculate from SDN policies. Each SDN policy comes with a set of criteria. Unlike simple firewall rules, SDN policy criteria can apply to any set of VMs. In order for traffic to be actually forwarded from one endpoint to another, both an ingress and an egress policy are required that apply to both endpoints respectively. A combination of matching ingress and egress policy criteria describes flows. In this paper, we calculate a minimum criteria match M from each pair of applicable ingress/egress policies that describes traffic that would actually be forwarded. For example, when one ingress policy's criteria allows all IPv4 flows from a source location into the network of the destination and an egress policy allows only IPv4 HTTP flows from the network of the origin to arrive at the destination location, they would jointly effectively only forward HTTP flows from the source to the destination endpoint.

For every directional link between two endpoints, we calculate the match M of every applying policy pair. Different ingress/egress policy pairs can shadow each other, e. g., if one match allows all Ethernet traffic, any more specific matches

Criteria	Ether type	DSCP	IP proto	src port	dst port
Range	2^{16}	2^{6}	2^{8}	2^{16}	2^{16}

TABLE II: Matching criteria ranges

are shadowed. If a pair doesn't have matching criteria, it does not contribute. Two matches can not partially overlap, i.e., they can either be equal or one flow is a super-set of the other.

For every directional link to/from the VM we calculate a weight from the fraction of allowed traffic, e. g., when the total match set forwards a wide range of traffic, the link receives a high score and when policies restrict to a smaller, more specific subset of traffic, the link receives a lower score. Criteria shown in table II show the value range. We calculate the total weight of the link by summing the fractions of each policy pair.

$$weight_{link} = \sum_{M} \prod_{crit \in M} \frac{value_{crit}}{range_{crit}} \quad (1)$$

For instance, assume a match set has three entries that allow a different IPv4 tcp destination port each, then the weight is the product of $1/2^{16}$ ether types (IPv4), any DSCP value, $1/2^{8}$ protocols (TCP), any $2^{16}/2^{16}$ source ports, and $3/2^{16}$ for the three allowed destination ports.

An alternative to the product of fractions is to transform the value range to weights that reflect the importance of given values. For instance, a TCP destination port 80 (HTTP) could be given a higher importance than UDP destination ports 67+68 (DHCP), because more traffic would be expected, or because DHCP in particular would not be considered to trigger large amounts of user plane traffic.

The complexity of the weighting grows with the number of endpoints in a deployment and the number of ingress/egress policies that apply to them. One potential concern is the temporal effect of the ranking onto the host selection algorithm, because the weight search complexity is super-linear. Hence, it is best to be implemented as part of the policy directory. A comprehensive evaluation of the scalability of our approach is left for further study.

IV. Evaluation

A. Functional evaluation

We have set up a testbed that uses OpenStack and Nuage VSP as shown in figure 4. For functional evaluation, we place short service chains onto our testbed. The exemplary service chain consists of an Apache web server [6] for an application and HA proxies [7] as transparent VNF services. User transactions are created using the curl-loader tool [8] to benchmark the throughput of a deployment.

The scenario consecutively places two service chains (green and red) of 3 instances each onto a group of four hosts, which makes it necessary to share at least one host. We run the scenario once with the default Nova scheduler configuration that uses a greedy host allocation. Then we run the scenario again with our PolicyWeigher ranking activated, which co-locates affiliated instances of a service chain, provided that the same host has enough resources.

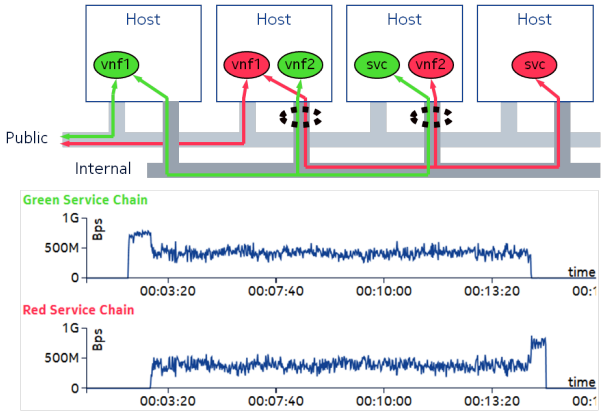

Fig. 5: Default placement scenario

The default placement result in figure 5 shows the co-location of instances of both chains. Traffic forwarding from the red *vnf1* to *vnf2* and again to the web server *svc* of the red chain causes flows to share the host network interface with the green *vnf2* and the green web server *svc*. As a result, the first scenario run yields a throughput of half the host interface speed (500 Mbit/s) when both service chains are active.

Policy-driven placement achieves co-location of all three instances of each service chain as depicted in figure 6. After the placement of the green service chain, the first instance of the red service chain is deployed on a new host because policy weighing would not find any forwarding rules that allow traffic between the separate chains. Even though our hosts would have sufficient capacity to each host all 6 instances of the two chains, the default ranking tends towards greedy allocation, so a new host is allocated for the first instance of the new chain. The remaining instances are again co-located with the first one due to the policy rules that forward traffic along the chain. As a result, each chain can serve user traffic at the full host network interface speed (1 Gbit/s).

This simple scenario shows how our *PolicyWeigher* ranking can avoid that independent service chains compete by calculating a weighted preference to co-locate instances.

B. Benefit evaluation

Theoretical studies have already shown that network-aware placement can have substantial benefits in much larger and more complex network scenarios (see Section V).

The quantitative benefit of the service chain performance depends on the chain model and eventually on the accuracy of the SDN configuration. SDN policies may vary in how specifically they describe the traffic. Policies could forward all traffic between two endpoints, which would be interpreted in our scheme as a strong affiliation – even though the instances may barely communicate in reality. Our solution will therefore not in all cases improve the performance.

The benefit also depends on service chain length and traffic indirection among VNFs, as not all flows may traverse the same chain. There is further potential for optimization if points of indirection (e. g. load balancers) can be adapted to forward

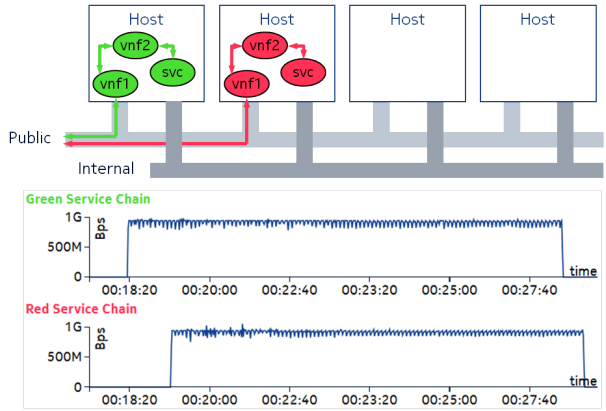

Fig. 6: Policy-driven placement scenario

traffic to a co-located instance, rather than equally distributing traffic to both co-located and remote instances. An evaluation of these approaches is left for further study.

V. Related Work

Addressing the networking requirements of cloud applications has been extensively researched (cmp. [9] [10] [11]). For instance, the localization of an application's VM traffic matrix formulated in [12] is a quadratic assignment problem, which can be addressed by heuristics. The distribution of an NFV application formulated in [13] is a combination of general assignment and facility location problem with an approximation. But such graph embedding solutions cannot easily be mapped to the per-VM host selection logic used by OpenStack. Online VM placement has been studied as a generalized multi-resource online allocation problem [14] but without focusing on chaining of instances. The authors of [15] present a system that needs to profile applications and their generated traffic before improving the network-aware placement. Barcelo et al. [16] formulate the resource allocation as a minimum cost mixed-cast flow problem with less complexity but yet embeds the full service graph.

Many published algorithms focus on offline optimization of placement for known workloads. These capacitated or minimum cost problem formulations require a model of the network demand prior to the deployment. In contrast, real cloud platforms have to decide instantaneously which node shall host the next instance. And the scheduler has in fact only limited information to make a decision.

There are few publications that actually study the scheduling algorithms used by OpenStack or other comparable platforms. A comparison of different scheduling policies [17] uses a simulation tool instead of the real OpenStack scheduler. Another study [18] evaluates the behavior of the OpenStack scheduler with black box measurements without considering its internal realization. There is also other related work such as [19], but this study is based on an outdated implementation of the scheduler. In addition, there are also many ongoing research activities towards improving the OpenStack scheduler for NFV infrastructures. For instance, the authors of [20] present a solution for multi-objective resource scheduling, but the evaluation is limited to simple scenarios.

VI. Conclusion

This paper presents how Software Defined Networking (SDN) policies can be used to improve placement of service chains, e. g., in Network Functions Virtualization (NFV) solutions. The existing instance placement logic in cloud computing platforms has limitations if instances have to deal with large amounts of network traffic. As a solution, we design a new scheduler that includes a weighing algorithm for SDN policies. Our heuristics leverage hints that SDN policies provide about the connectivity needs and network demand of new instances. We have developed a prototype for SDN-driven placement in the OpenStack platform. Our functional evaluation shows promising improvements since our approach can separate chains that compete for network resources.

References

[1] ETSI, "Network functions virtualisation," http://portal.etsi.org/NFV/NFV_White_Paper.pdf, 2012.
[2] OpenStack, "Web site," http://www.openstack.org, 2016.
[3] OpenStack, "Nova Filter Scheduler," http://docs.openstack.org/developer/nova/filter_scheduler.html, 2015.
[4] R. Alimi, R. Penno, Y. Yang, S. Kiesel, S. Previdi, W. Roome, S. Shalunov, and R. Wound, "Application-Layer Traffic Optimization (ALTO) Protocol," RFC 7285, 2014.
[5] OpenStack, "Compute API," http://developer.openstack.org/api-ref-compute-v2.1.html, 2016.
[6] Apache Software Foundation, "HTTP Server," http://httpd.apache.org/.
[7] W. Tarreau, "HAProxy," http://www.haproxy.org/, 2015.
[8] R. Iakobashvili and M. Moser, "curl-loader," http://curl-loader.sourceforge.net/, 2012.
[9] N. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization," *Comput. Netw.*, vol. 54, no. 5, pp. 862–876, 2010.
[10] B. Jennings and R. Stadler, "Resource management in clouds: Survey and research challenges," *Journal of Network and Systems Management*, pp. 1–53, 2014.
[11] K. Oberle, M. Kessler, M. Stein, T. Voith, D. Lamp, and S. Berger, "Network virtualization: The missing piece," in *Proc. Intelligence in Next Generation Networks (ICIN)*, 2009.
[12] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *Proc. IEEE INFOCOM*, 2010.
[13] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, "Near optimal placement of virtual network functions," in *Proc. IEEE INFOCOM*, 2015.
[14] F. Hao, M. Kodialam, T. V. Lakshman, and S. Mukherjee, "Online allocation of virtual machines in a distributed cloud," in *Proc. IEEE INFOCOM*, 2014.
[15] K. LaCurts, S. Deng, A. Goyal, and H. Balakrishnan, "Choreo: Network-aware task placement for cloud applications," in *Proc. ACM IMC*, 2013.
[16] M. Barcelo, J. Llorca, A. M. Tulino, and N. Raman, "The cloud service distribution problem in distributed cloud networks," in *Proc. IEEE ICC*, 2015.
[17] B. Hu and H. Yu, "Research of scheduling strategy on OpenStack," in *Proc. CLOUDCOM-ASIA*, 2013.
[18] O. Litvinski and A. Gherbi, "Experimental evaluation of OpenStack compute scheduler," *Procedia Computer Science*, vol. 19, pp. 116–123, 2013.
[19] H. Lindgren, "Performance management for cloud services: Implementation and evaluation of schedulers for OpenStack," Master Thesis, KTH, 2013.
[20] M. Yoshida, W. Shen, T. Kawabata, K. Minato, and W. Imajuku, "MORSA: A multi-objective resource scheduling algorithm for NFV infrastructure," in *Proc. APNOMS*, 2014.