© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Enabling SDN Hypervisor Provisioning through Accurate CPU Utilization Prediction

Nemanja Đerić[1], Amir Varasteh[1], Amaury Van Bemten[1], Andreas Blenk[1,2] and Wolfgang Kellerer[1]

[1] Chair of Communication Networks, Department of Electrical and Computer Engineering, Technical University of Munich, Germany

[2] Communication Technologies, Faculty of Computer Science, University of Vienna, Austria

***Abstract*—Providing predictable performance to tenants is mission critical for network hypervisors. As a hypervisor acts as an intermediary between tenants controllers and the physical infrastructure, its resources (e.g., CPU, RAM) should be provisioned and allocated carefully. Initially, we demonstrate that state-of-the-art CPU prediction approaches are not suitable for provisioning network hypervisor CPU resources, since they predict only the mean CPU utilization. However, provisioning the resources with a mean value can significantly degrade the forwarding performance of a network hypervisor. In this article, we present a novel approach which provisions network hypervisor CPU resources efficiently, while avoiding performance degradation. We take three steps to achieve our goal: *(i)* conducting a profound measurement campaign to determine what is the minimum amount of CPU resources that needs to be allocated to a network hypervisor in order to have no performance degradation; *(ii)* revealing the key properties of virtual networks that affect the CPU utilization; *(iii)* designing a precise CPU prediction model. Using randomly generated virtual networks and arbitrary physical topologies, we show that our prediction model exhibits an average relative error of around 4%. Further, our evaluations indicate that provisioning the CPU resources of a network hypervisor based on the proposed prediction model does not degrade the hypervisor forwarding performance. Utilizing our approach, network operators can minimize their resources consumption while still providing predictable and undegraded forwarding performance to tenants.**

***Index Terms*—network virtualization, network hypervisor, isolation, measurement, modeling, control plane**

I. Introduction

A. Context: Provisioning Network Hypervisor Resources

Network virtualization strives to enable the coexistence of different network operating systems on the same physical infrastructure [1]. This opens interesting advantages over non-virtualized environments: faster innovation through parallel deployment or easier testing of new network algorithms on the real infrastructure [2]–[5]. *Software-defined networking (SDN)* frames a particularly interesting case for *network virtualization:* tenants can bring their own SDN controllers in order to control their virtual SDN networks. The integral component realizing network virtualization is the *network hypervisor* [6] (NH). It is logically located between the tenants SDN controllers and the physical infrastructure [6] (e.g., forwarding devices/switches). It translates all the control plane messages (e.g., switch forwarding rules) exchanged by the tenants SDN controllers and the physical switches. Additionally, an NH inspects the translated messages in order to verify that the tenants are not violating the agreed virtualization policies (e.g., allowed forwarding behavior of data plane devices). Following the general trend of softwarization and the today's high demand for flexibility [7], most NHs are realized as software instances (e.g., Java programs) running on commodity hardware [1]. Therefore, we can consider an NH as a (complex) virtual network function (VNF) – it uses hardware resources such as CPU or RAM in order to process control plane messages. Thus, the performance of an NH directly depends on the available physical hardware resources (i.e., available number of CPU cores). Therefore, it is crucial to carefully provision the resources of NH, as its processing performance impacts the performance of *virtual networks (VNs)*.

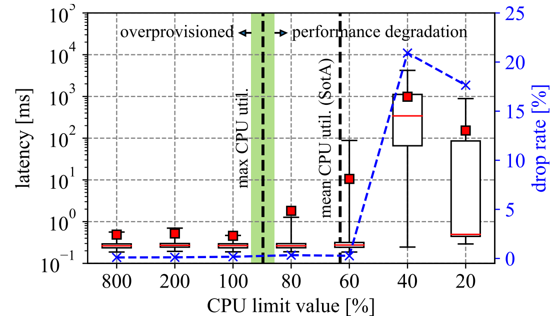


Fig. 1: Impact of the allocation of CPU resources of an NH on the control plane message latency and loss. Dashed black lines show the maximal and average observed CPU utilization during an unconstrained run, where all of the cores where allocated to network hypervisor (CPU limit is 800 %). The server running an NH has in total 4 physical cores, thus 8 hyperthreads, making the maximal CPU utilization of 800 %.

B. Motivation: Predictable Virtual Network Performance

Different strategies exist to allocate hardware resources to hypervisors. One existing strategy simply over-provisions hypervisors [6], [8] — a clearly too expensive strategy in terms of resource consumption. Another strategy is to derive hypervisor performance models [9], [10] and to provision resources accordingly. However, as we will show, state-of-art performance models fall short in terms of accuracy and precision.

In order to illustrate this, we evaluate the control plane processing latency of an NH for varying amount of allocated CPU resources (Fig. 1). In this scenario, allocating around 90 % capacity of a single CPU core represents an ideal resource allocation decision. In Fig. 1, this is shown as the highlighted area around the left dashed line (i.e., green area). Allocating

fewer resources, i.e., *under-provisioning* the CPU resources, increases the control plane message latency by up to three orders of magnitude and the loss of control plane messages by $20\,\%$[1] — a clearly unacceptable performance degradation. On the other hand, *over-provisioning* the CPU resources does not yield performance benefits: the latency stays below $1\,ms$ for $100\,\% - 800\,\%$[2] of CPU capacity. However, as the figure illustrates, over-provisioning leads to a waste of resources and is actually not needed when having a precise performance model. Furthermore, Fig. 1 also shows why state-of-the-art CPU prediction approaches [9], [10] are not suitable for provisioning. They simply predict the mean CPU utilization (in this scenario $60\,\%$), which results in under-provisioned NH. An ideal NH provisioning system allocates the least amount of CPU resources so that no performance degradation occurs; in this case, the ideal allocation lies around $90\,\%$ of one CPU core. Furthermore, state-of-the-art prediction models are also too simplistic: they base their CPU prediction solely on the control plane message rate [9], [10]. Thus, these models ignore the potential impact of (virtual) network parameters and dynamic configuration changes (e.g., the number of VNs or a changing VN topology in case of varying demands or, e.g., failures). In this article we tackle the challenge to derive generalizable performance models that allow for precisely determining such CPU provisioning in various scenarios.

C. Contributions: Network Hypervisor Performance Model

In this article, we present a network hypervisor performance model that reduces over-provisioning while avoiding performance degradation. Our solution is scenario-agnostic and is, hence, capable of supporting physical topologies following a wide variety of graph models and a wide variety of VN requests (e.g., in terms of the number of (virtual) nodes, links and hosts). We take three steps to achieve our goal: we conduct a profound measurement campaign to *(i)* determine what is the minimum amount of CPU resources that has to be allocated to an NH in order to have no performance degradation; *(ii)* reveal the key properties of VNs that affect the CPU utilization; *(iii)* design a CPU prediction model. Indeed, deriving a precise performance model based on measurements is a challenging task: for instance, the physical topologies and VN configurations can vary greatly [11]. Hence, measuring the resource consumption of an NH for all possible cases is not feasible. Therefore, determining what parameters to evaluate and what values to select is crucial for designing precise and accurate prediction model. Finally, we validate and evaluate our model and compare its accuracy with *state-of-the-art* approaches on 3 randomly generated and 3 realistic topologies for a wide variety of generated VNs requests. In contrast to *state-of-the-art*, we demonstrate that provisioning the CPU resources with the presented model does not incur a significant control plane performance degradation.

[1]In case the resources are limited to 20%, an NH throttles the TCP connections in order to reduce the total amount of received messages. Hence, in this case processing time and latency are better compared to a scenario when the limit is 40%.

[2]Note that $800\,\%$ means a dedicated reservation, i.e., pinning of 8 threads.

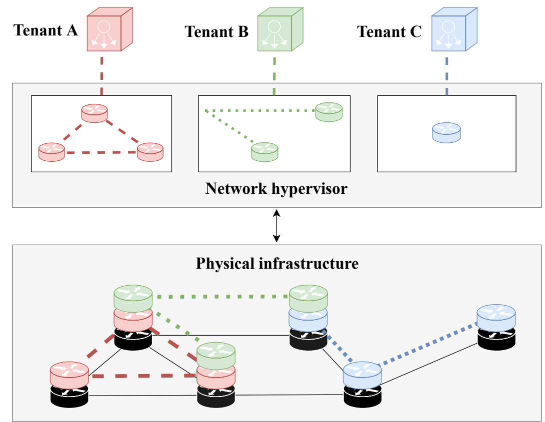


Fig. 2: Overview of network virtualization. A network hypervisor acts as a proxy between the tenants controllers and the physical infrastructure. A hypervisor can provide different virtualization policies to different tenants, i.e., in this case different levels of topology abstraction.

D. Organization

The rest of this article is organized as follows. Sec. II introduces background concepts and network virtualization terminologies. Sec. III outlines our benchmarking procedure, the measurement setup and its results. The NH CPU prediction model is developed in Sec. IV and thoroughly evaluated in Sec. V. Finally, Sec. VI presents the related work and Sec. VII concludes the paper and discusses remaining open issues.

II. Background: Network Virtualization

In this section, we firstly present a short background regarding network virtualization in SDN, where we analyze various virtualization functionalities. Furthermore, we also provide insights on the basic implementation concepts commonly used when designing an NH in order to understand which VN parameters affect NH CPU workload.

A. Virtualization in SDN

Virtualization of SDN networks enables the *network as a service* (NaaS) model. Tenants can request network resources (e.g., a custom virtual topology with bandwidth requirements) to a provider. Through network virtualization, the provider allows different tenants to share a common physical infrastructure. The concept is illustrated in Fig. 2. Each tenant is given a *virtual network* (VN), i.e., a set of virtual (interconnected) switches and hosts, to which it can connect its own custom *SDN controller*, or *controller*. Through their controller, tenants can fully control the data plane forwarding behavior in their VN and steer the traffic as they desire. In order to provide such a service, the provider uses a *network hypervisor* (NH) which acts as a proxy between the controllers of the tenants and the physical infrastructure. When a tenant requests a new VN, a *VN embedding* algorithm is responsible for mapping this VN to the physical infrastructure and for reserving data plane resources (e.g., bandwidth) accordingly [11]–[13]. Furthermore, in the data plane, tenants are differentiated by their *flowspace*, where a *flowspace* is a subset of all possible OpenFlow (OF) [14] matching fields. If the tenants flowspaces

are not overlapping, all the data plane traffic can be easily mapped to the corresponding tenant. For instance, we can define a flowspace of a tenant A (see Fig. 2) as the *10.0.0.1/24* subnet with full port access on physical switches highlighted in red color (they are interconnected with dashed lines).

An NH is also responsible for ensuring that all tenants are indeed using their VNs according to the agreed virtualization policies (*security feature*). This is achieved by inspecting the control plane messages and potentially rewriting them during the translation operation. Furthermore, some NHs provide additional features targeting simplification of the data plane management to tenants. For instance, certain NHs provide topology abstraction [15], [16], control plane isolation [6], or full flowspace usage [15]. Naturally, there are many different ways to implement these functions, making the resource prediction modeling implementation-dependent [17]. In this article, we use *FlowVisor (FV)* and we focus on evaluating the most basic and crucial *security feature* that all NHs have to implement. Furthermore, we also focus on provisioning the CPU resources for a flow embedding task[3] as it is a cornerstone of remote network control, and many applications are realized using only this feature [13].

B. Message Inspection/Processing in FlowVisor

FlowVisor (FV) [6] implements *partial topology abstraction* and *full port abstraction*. *Partial topology abstraction* can be defined in the following way: besides the requested end-point physical switches, the intermediate switches on the paths between the end-point switches are also shown to the tenants controllers, while all other physical switches are hidden. This corresponds directly to the red dashed VN in Fig. 2 (i.e., tenant A), where the physical switches are mapped with *one-to-one* configuration to the virtual switches. *Full port abstraction* means that only the physical ports containing the tenants hosts and interconnecting the VN are shown, while all other existing physical ports are hidden or abstracted away. In order to achieve such functionalities, each control plane message has to be processed by the NH. In OF [14], the forwarding behavior of a switch is modified with *FlowMod Add* message[4], hence, the corresponding message is one of the most complex to inspect. For instance, match and action fields contained in every *FlowMod Add* message have to be checked with respect to the agreed virtualization policies. As we focus on provisioning the resources for flow embedding task, in the following we explain the FV's processing pipeline of *FlowMod Add* in more detail.

Upon a reception of *FlowMod Add* message, FV firstly checks the contained action set. For instance, the corresponding message might be trying to add a rule which forwards the traffic to a port which is not part of the tenant's *flowspace*. Thus, we suspect that the number of virtual ports could have an impact on the processing workload of FV. If the contained action set is not violating the agreed policies, FV then inspects the match. This is done by intersecting the match with the *flowspace* and evaluating if the tenant has a permission to match on those fields. If a tenant is using additional field (not included in its *flowspace*), FV rewrites them. Similarly, we suspect that type of matching can have an impact. Thus, if we use port based matching (common to L2 and L3 forwarding applications), the required resources might scale again with the number of virtual ports. Finally, the message is forwarded to the targeted physical switch. Moreover, the size of topology could have an impact on the total processing workload, as it can affect the lookup time for determining the correct physical switch destination. Furthermore, the topology size and edge density directly impact the total number of ports in the network, thus, potentially affecting the required workload for inspecting the match and action fields of *FlowMod Add*. Although FV implements most of the look ups with hashmap (scales with $\mathcal{O}(1)$), some look ups use linked lists (scales with $\mathcal{O}(n)$), thus the lookup and workload could be affected by the aforementioned parameters.

[3]In OF, flow embedding is achieved with *FlowMod Add* message.

[4]In OF 1.0 specification [18] *FlowMod Add* is called OFPT_FLOW_MOD.

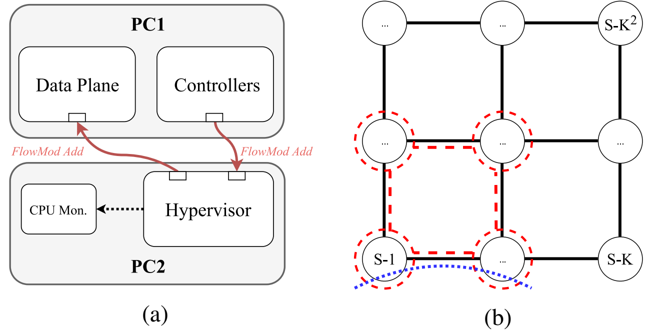


Fig. 3: (a) Measurement setup consisting of 2 interconnected PCs, and (b) illustrations of the physical data plane grid topology of dimension $k = 3$ (black solid lines), with example grid VN of dimension $k_v = 2$ (red dashed lines), and a possible flow request spanning over two virtual and physical switches (blue dotted line).

III. Hypervisor Benchmarking

In this section we firstly introduce our measurement setup (Sec. III-A) and present the considered VN parameters (Sec. III-B). The parameters are based on the insights presented in the background section. Further, in Sec. III-C we present our benchmarking procedure, while the results of our measurement campaign are presented in Sec. III-D.

A. Measurement Setup

In order to better understand how different VN configurations (e.g., VNs with different amount of virtual switches) influence network hypervisor resource utilization, initially we conducted our measurements on a smaller setup. Having such a setup allows us to have a more controlled environment, thus, the impact of various parameters was easier to investigate. The measurement setup is depicted in Fig. 3a. It consists of two PCs equipped with Intel quad-core i7-7700 CPUs, 16GB of RAM, and running Ubuntu 14.04 LTS.

The first PC (*PC1*) *(i)* runs *mininet* [19] to emulate a physical infrastructure, and *(ii)* runs the tenants controllers as multiple *Ryu* [20] instances. The controllers generate control plane messages with the goal of embedding a certain number of flows per second (described in Sec. III-C).

TABLE I: Evaluation parameters.

Parameter	Notation	Values
number of tenants	t	2, 3, 4, 5
physical topology size	k	4, 9, 16, 25
per-tenant flow request rate	r_i	90, 180, 280, 400
per-tenant flow length	l_i	2, 3, 4, 5, 6, 7
per-tenant virtual topology size	v_i	4, 9, 16, 25
Per-tenant number of virtual ports	p_i	1, 2, 3, 4, 5

The second PC (*PC2*) runs FV as the network hypervisor. The logging and *FlowMod Add* state keeping features, which are not necessary for the normal operation of the NH, are disabled. The CPU utilization of *PC2* is sampled every 0.1 second (lower values are not recommended) by a CPU monitor implemented with the Python *psutil* [21] library. CPU utilization is measured in percents of a single-threaded core: since the PC has four cores with hyper-threading, CPU utilization ranges from 0% to 800%. The messages sent by the controllers are received, translated, and forwarded by FV towards the corresponding data plane switches emulated by *mininet*.

The amount of available resources allocated to an NH is controlled with *cpulimit* tool. The *cpulimit* tool is process-based, meaning that if a specified process exceeds the allowed CPU resource consumption, it uses SIGSTOP and SIGCONT POSIX signals in order to throttle the process accordingly.

The physical data plane and tenants controllers run on the same PC. To ensure that this does not affect the measurements, we ensure that the PC is always underutilized during the measurements.

B. Evaluated Parameters and Scenario

We focus on evaluating the impact of flow embedding tasks on the required NH CPU resources; our control plane traffic between tenant controllers and their virtual networks consists of *FlowMod Add* messages only. This is a standard choice as flow embedding is *(i)* a paramount functionality of the remote control of networks, and *(ii)* many applications, e.g., industrial applications with strict QoS requirements messages [13], can be implemented solely with *FlowMod Add* messages. In order to establish one flow between two hosts, tenant's controller generates one *FlowMod Add* message towards each switch on the chosen shortest virtual path. We consider that tenants add rules matching on physical input port, and unique destination IP addresses, while the action is to forward (output) to a certain port. Input and output ports are determined by the shortest path calculation. The NH receives the corresponding messages and processes them as described in Sec. II.

Based on the insights gathered in the background section (see Sec. II), we consider the following evaluation parameters and reason about their usage (see Tab. I):

- *Number of tenants t:* For each tenant, the NH has to maintain additional TCP/OF connections from the virtual switches, i.e., from the hypervisor, to the tenant controllers, store the isolation and abstraction policies, and potential state variables.
- *Physical topology size k:* The NH appears as the controller to all physical switches. Hence, an NH must keep one TCP/OF connection towards each physical switch. Thus, the number of switches k in the physical topology can potentially affect the required CPU resources.
- *Per-tenant flow request rate r_i*: Each tenant adds flows to its VN with a pre-defined rate r_i (with uniform arrival distribution). Increasing the rate linearly increases the number of *FlowMod Add* messages on both interfaces of the hypervisor; the message has to be received from the tenant controllers (interface one) and forwarded towards the physical switches (interface two). As a consequence, the rates might conceivably increase CPU utilization [9], [10], [17].
- *Per-tenant path-flow length l_i*: The longer the path of a flow is, the more messages have to be processed and translated by the NH (one per physical switch). We define the path-flow length as the number of switches between the end hosts of the flow. FV does not provide any topological abstraction: even if the tenants only request to manage two physical switches, they might have to also control all switches connecting the two physical switches.
- *Per-tenant virtual topology size v_i ($v_i \leq k$):* FV guarantees no control policy violation. Hence, every *FlowMod Add* message received by FV has to be inspected in order to ensure that the tenants are only modifying the switches in their VNs. Thus, with more virtual switches, the larger the virtual switch list is, which could affect the workload.
- *Per-tenant number of virtual ports p_i:* Before forwarding *FlowMod Add* messages to the physical switches, the hypervisor translates virtual port numbers to physical port numbers and makes sure that a tenant is using only physical ports attached to its VN. Having a larger number of virtual ports can hence increase the lookup time, in turn affecting the workload of the NH.

In our scenario, we assume that each tenant has one host attached to each switch which is mapped to one virtual port. Furthermore, each tenant also requests the additional virtual ports in order to interconnect its virtual grid network. *Note:* In order to vary the number of virtual ports p_i, we use a different method, we allow tenants to also attach the virtual ports (hosts) dedicated to other tenants, thus increasing the amount of virtual ports.

Chosen Parameter Values (Tab. I). Current *state-of-the-art* SDN-enabled carrier grade switches are being shipped with small flow table size and they cannot handle high control plane traffic rates [22], [23]. For instance, the 10G forwarding device PICA P-3290 can only handle up to around 1000 rule/flow updates per second [23]. Therefore, in this paper we consider similar parameter values as we want to demonstrate that our solution is capable of supporting carrier grade SDN-enabled hardware. For example, if we consider 5 tenants, where each tenant has a flow request rate of 400 per second, in the worst case, one physical switch could experience up to 2000 update messages per second. This value is around 2x higher compared to the supported rate of PICA P-3290. Furthermore, the maximal considered sizes the physical (grid)

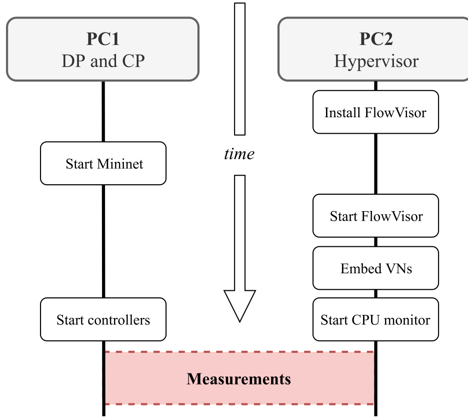


Fig. 4: Benchmarking procedure. PC1 emulates the data plane and the tenants SDN controllers, while PC2 is running the virtualization layer and CPU monitoring script.

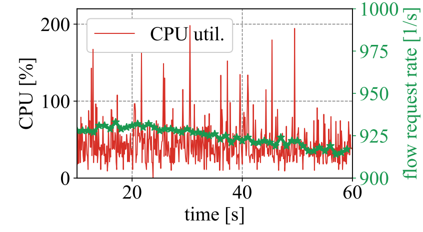


Fig. 5: Observed time series of CPU utilization (red line with left y-axis) during one measurement run with the following parameters: $t = 5$, $r_i \sim 184$, $k = 4 \times 4$, $v_i = 4 \times 4$, $l_i = 4$ and $p_i = 1$. Furthermore, the green scattered line with the right axis represents time series of the total observed flow request rate per second during the aforementioned run.

topology and VNs is consistent with common topologies such as Internet2 [24] and Nobel EU [25].

C. Measurement Procedure

Fig. 4 depicts the general measurement procedure. A measurement scenario is defined by the set of evaluation parameters values defined in the previous subsection (see Sec. III-B). For one measurement configuration, the measurement runtime is 60 seconds and it is repeated 10 times for statistical significance (unless stated otherwise). Furthermore, we discard the observed samples during the first 10 seconds of the run, in order to avoid any potential transient phase due to initial switch connections or transport connection stabilization. Exploring all possible combinations of our six evaluation dimensions is time-wise infeasible. For instance, if we vary every parameter five times, the total measurement time would amount to a few months. Therefore, we perform the measurements only for certain scenarios with manually chosen parameter values, selected with the goal of inferring parameter scaling dependencies.

During our measurement campaign we observed that running the measurements one after each other could lead to software aging effects [26], [27], i.e., the CPU utilization of FV could increase over successive runs of the same scenario. Hence, before each run, FV is completely reinstalled. That is, all configuration files, logs, and the database are deleted, and FV is rebuilt. Finally, all remaining processes from previous runs are killed and the memory cache is cleared.

Firstly, the data plane topology is started based on the measurement scenario. FV is then booted and the tenants VNs are embedded with the corresponding requirements based on the chosen configuration of the measurement scenario. The flowspace of tenants corresponds to different $/16$ subnets and all the requested virtual switches with their virtual ports. We consider grid topologies for both the physical and VNs because they are easy to scale up or down with non-random parameters. Occasionally, embedding large VN can overload the NH, resulting in a software crash. Such runs are repeated.

After all VNs are embedded, the CPU monitor is initialized and the tenants controllers are started. Before starting the measurement, we ensure that all controllers finished the initial `OF` handshake procedures with all of the requested virtual switches (i.e., with FV). Furthermore, we also ensure that all tenants SDN controllers finished generating their corresponding path request list. That is, each tenant initially generates randomly 100 paths, where each path is generated with Dijkstra's shortest path algorithm [28] between two randomly selected end hosts. The path is defined as a set of switches on the path, with the corresponding ports. In certain measurement cases, all paths are supposed to have the same length. Thus, before adding a path to the corresponding path request list, we simply repeat the path generation procedure until we obtain a path with a desired length. During the runtime, a flow request is then defined by generating *(i)* a unique destination IP address within the tenant's flowspace and by *(ii)* randomly selecting a path from the path request list. This is done in order to avoid heavy path computations during the measurement runtime, as it could affect the measurement precision.

After the initialization procedure, the controllers (each tenant has one) start adding flows with rate r_i[5]. The generation of flows is uniformly spaced, and the flows are generated based on the aforementioned path request list and unique IP address. In this section, we assume that all tenants have the same rate, virtual topology size and number of virtual ports.

If the controller sends multiple `OF` messages as one TCP segment, this reduces the total workload of the NH, in contrast to sending one `OF` message per TCP segment [8]. However, as the flow generation is uniformly spaced in the time, merging of multiple TCP segments almost never occurs on the controllers side.

D. Measurement Results

This section reports the results of our measurement campaign. Firstly, we evaluate the stability and repeatability of our measurements, i.e., we evaluate if *FlowMod Add* generation rates are stable and if repeating one measurement scenario produces the same results (as the already observed run). Afterwards, we evaluate the impact of the considered VN parameters on the observed CPU utilization.

1) Measurements Stability: Fig. 5 shows the measured CPU utilization time series for the complete duration of one run,

[5]The rate of flow addition is configured in *Ryu* through simple *sleep* commands. This is quite imprecise but, as shown in Fig. 5, is stable enough. Hence, we define the r_i as the mean rate *actually* generated by *Ryu* and obtained through post-processing of the traces.

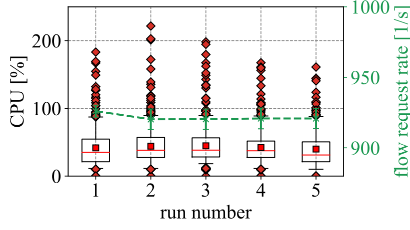
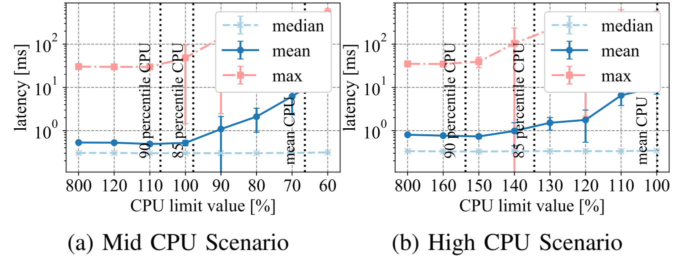


Fig. 6: Distributions of measured CPU utilization for five measurement runs with identical evaluation parameters, i.e., $r_i \sim 300$, $k = 5 \times 5$, $v_i = 4 \times 4$, $l_i = 4$ and one host per virtual switch. The boxplots whiskers correspond to the 5% and 95% percentiles. The green line shows the mean total *FlowMod Add* rate and its standard deviation.



(a) Mid CPU Scenario (b) High CPU Scenario

Fig. 7: Impact of limiting the CPU resources allocated to NH on the observed control plane processing time latency (median, mean and maximal) for two different measurement cases. Every run with different CPU allocation limit is repeated 5 times, and the mean values of these 5 runs are shown with their corresponding confidence intervals.

alongside with the total flow request rate $\sum_{i=1}^{n} r_i$ generated by the controllers. All flows have the same length, thus the amount of *FlowMod Add* messages per second received by *FV* is constant and directly correlated with the flow request rate. However, even though the flow generation is uniformly spaced and stable (the maximum variance is in range of a few percents), we observe that CPU utilization exhibits high variability, with multiple extreme peaks. For instance, the minimum observed CPU utilization is close to $0\,\%$, the maximal is around $198\,\%$, while the mean is $45\,\%$. During runtime, in order to avoid blocking the TCP socket/connection, *FV* places the received messages in a queue, which is then periodically cleared. Hence, the workload oscillates with time, suggesting that predicting the exact CPU utilization in one specific time instance of one run is hardly possible.

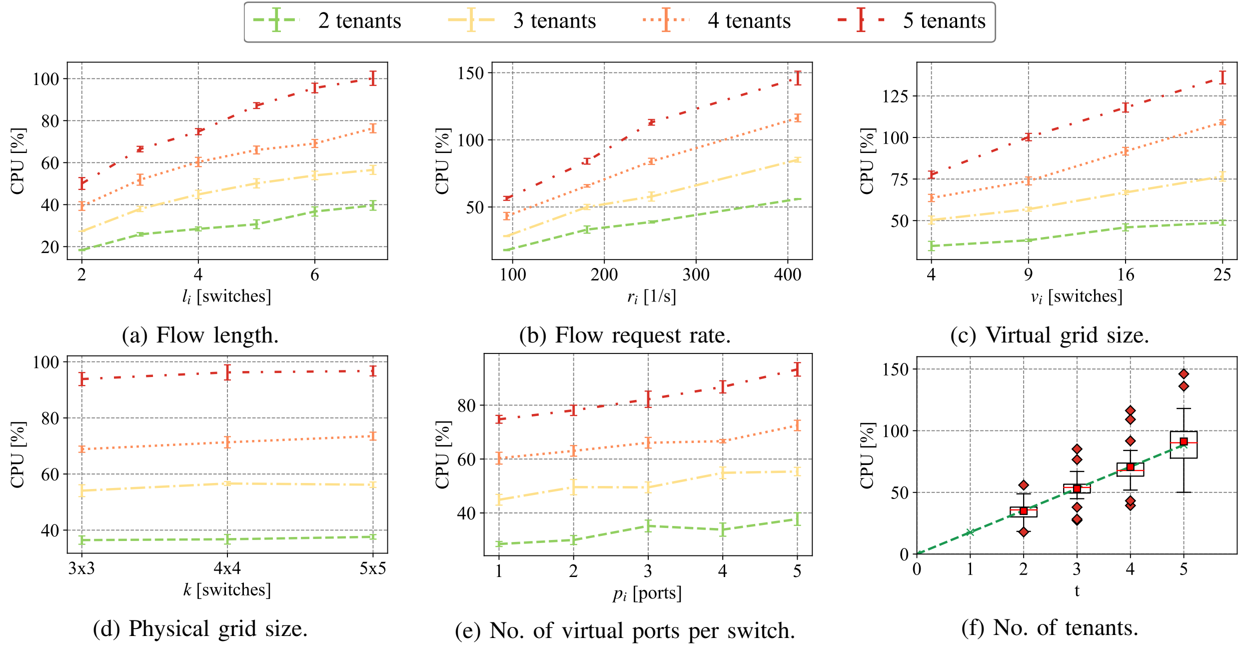
Although the observed CPU utilization within one run varies, repeating the same measurement run multiple times with the same parameters produces almost identical CPU utilization distributions. For instance, in Fig. 6, the mean observed CPU utilization of all five measurement runs falls within the range of 39%–45%. This indicates that modeling and predicting the statistical properties (e.g., median) of an NH CPU utilization profile is indeed feasible.

2) Allocating a Sufficient Amount of Resources: Precisely allocating hypervisor resources is only needed in case a CPU limitation truly affects the network performance, e.g., NH forwarding latency. Accordingly, for precise performance modeling, it is important to find a point, i.e., statistical value, which is used for allocating hypervisor resources. Ideally, such point would minimize the amount of allocated CPU resources while avoiding performance degradation. In order to determine it, we evaluate two randomly generated scenarios, first scenario has higher CPU requirements while the second one has lower. Fig. 7 shows the impact of limiting the available CPU resources of the NH on the control plane processing time for the aforementioned two scenarios. Furthermore, statistical properties (mean, 85th percentile and 90th percentile) of the measured CPU utilization profile of an unconstrained run (i.e., run when all of the resources are allocated to the NH, CPU limit is 800%) are shown as dashed vertical lines. Firstly, provisioning the CPU resources based on the observed mean (*state-of-the-art* approach) or 85th percentile incurs a significant increase of both mean and max latency. On the other hand, provisioning with 90th percentile does not produce the same impact, i.e., the latencies stay the same as in unconstrained or baseline scenario (i.e., CPU limit is 800%). Therefore, we advocate to use the **90th percentile** as a profound match: the 90th percentile offers a good trade off between performance predictability and resource allocation overhead. *Note:* Different use cases might have different performance requirements, thus using any other percentile value could also pose as a valid solution – the chosen value only defines a trade off between performance and resource consumption.

From now on, we evaluate how the aforementioned evaluation parameters (see Sec. III-B) affect the 90th percentile CPU utilization (with CPU utilization we refer to 90th percentile CPU utilization), and we strive to learn the scaling dependencies in order to generate a prediction model capable of supporting arbitrary topologies and randomly generated VN requests. Fig. 8 shows the observed CPU utilization values of an NH for all considered parameters. Each value is a mean of 10 runs (for each run we observe 90th percentile CPU utilization) and the corresponding confidence intervals.

3) Flow Length: Fig. 8a shows the impact of the flow length on the NH 90th percentile CPU utilization. Since we disabled the state keeping feature, the messages are processed independently. Thus, the total number of *FlowMod Add* messages increases linearly with the flow length; in turn the CPU utilization increases linearly with the *FlowMod Add* message rate. For example, for 5 tenants adding 184 flows per second, the measured CPU utilization increases with the path length from around $50\,\%$ to around $100\,\%$ linearly.

4) Flow Rate: Similarly, the *FlowMod Add* message rate and the CPU utilization increases linearly with the flow request rate (see Fig. 8b). For instance, on average, for 5 tenants with path lengths of 6, embedding 5×90 flows per second requires only around $56\ \%$ CPU resources, while embedding 5×434 flows per second requires around $146\,\%$. As both of these parameters impact the *FlowMod Add* message rate per tenant in the same manner, the impact of the flow rate and flow length is almost the same. Since all other parameters are the same (e.g., VN size), the slopes of both curves are indeed approximately equal: $0.0109\,\%$/(*FlowMod*/s) for flow length and $0.0113\,\%$/(*FlowMod*/s) for the flow rate. Moreover, the flow request rate and the corresponding flow lengths can be used to determine the total number of translated *FlowMod Add* messages by the hypervisor.



(a) Flow length. (b) Flow request rate. (c) Virtual grid size.

(d) Physical grid size. (e) No. of virtual ports per switch. (f) No. of tenants.

Fig. 8: (a)–(f) Impact of different evaluation parameters on the NH 90th percentile CPU utilization on a physical grid topology for 2 to 5 tenants. Plots show mean observed CPU utilization values of 10 runs along with the 95% confidence intervals assuming uniform distributions. The following parameters are presented: (a) path length with $r_i = 184$, $k = 4 \times 4$, $v_i = 4 \times 4$ and $p_i = 1$, (b) flow request rate with $k = 5 \times 5$, $v_i = 4 \times 4$, $l_i = 5$ and $p_i = 1$, (c) virtual topology size with $r = 430$, $k = 5 \times 5$, $l_i = 3$ and $p_i = 1$, (d) physical topology size with $r = 430$, $v_i = 3 \times 3$, $l_i = 3$ and $p_i = 1$, (e) number of virtual ports dedicated for connecting the hosts per switch with $r = 184$, $k = 4 \times 4$, $v_i = 4 \times 4$, $l = 4$. (f) shows the data of (a)–(f) as box plots for the different number of tenants (5% and 95% percentile whiskers are used).

5) Physical and VN Size: As suspected, increasing the VN size does indeed affect the CPU utilization (see Fig. 8c). For instance, embedding 5×434 flows per second (5 tenants) with a length of 3 in a 2×2 virtual grid network generated CPU utilization of around 78 %, while repeating the same measurement in 5×5 grid lead to 136 % CPU utilization. Interestingly, the CPU utilization does not scale linearly with the total number of virtual switches but with its square root. Although this might sound counter-intuitive (as typically lookup scalings are $\mathcal{O}(n)$ for a list or $\mathcal{O}(\log(n)), \mathcal{O}(n\log(n))$ for tree-like data structures), increasing the virtual grid dimension also increases the average node degree (i.e., number of virtual ports), which also has an impact of CPU utilization. Contrary to our expectations from Sec. III-B, the size of the underlying physical topology does not produce a significant impact on the CPU utilization (see Fig. 8d). For instance, with 3 tenants, the observed CPU utilization increased only from ~ 54 % (for 3×3 physical grid) to around ~ 56 % (for 5×5 physical grid).

6) Number of Virtual Ports: The previous measurements are based on the assumption that each tenant has one host connected to each virtual switch (requires one virtual port). Thus, coupled with topology knowledge, the total number of virtual ports per each tenant's switch is fully defined. However, other physical topologies and VN configurations can have a different number of virtual ports per switch. To incorporate the topology impact in our measurements, we investigate the impact of the number of virtual hosts/ports per switch. To this end, we increase the total number of hosts per tenant on each switch from 1 to 5, in turn increasing the number of virtual ports. We observe a non-negligible linear impact (see Fig. 8e). For instance, the observed mean CPU utilization for 5 tenants is increased from around 74 % (for 1 host per virtual switch) to around 93 % (for 5 hosts per virtual switch). This occurs since the NH inspects every *FlowMod Add* in order to investigate if the tenants are indeed using only their corresponding *flowspaces*.

7) Number of Tenants: Fig. 8f presents all of the data from Figs. 8a–8e merged and sorted based on the total number of tenants. Furthermore, the plot shows the linear regression of the 90th percentile values for each number of tenants. We observe that the intercept of the regression line is around 0 %. Although the number of tenants in some cases could have a non-linear impact (see $p_i = 2$ on Fig. 8e), we also highlight that the deviation is not drastic. Overall (see Fig. 8f), for small number of tenants, the impact can be simplified and assumed additive. That means that the increase in CPU utilization observed for each additional tenant is always the same. In particular, that means that, for predicting the CPU utilization for a multi-tenant scenario, we can simply compute the impact of each tenant independently and add the resulting CPU values.

8) ***Summary:*** While CPU utilization is by nature a highly variable metric, we have seen that our measurement procedure always obtains stable results. We have shown that the impact of the considered evaluation parameters on the 90th percentile CPU utilization can be approximated by simple functions (models). This suggests that a precise modeling of the 90th percentile of CPU utilization is indeed possible. Further, the size of the physical topology seems to have no major effect on the resource consumption, hence, suggesting that the model

can be physical-topology-agnostic for smaller sized networks. In the following section (see Sec. IV-B), we will use these measurements and observed scaling dependency in order to generate a CPU performance model capable of supporting arbitrary physical networks and a wide variety of VN requests.

IV. Hypervisor CPU Prediction Model

In this section, based on the results of our measurement campaign, we present our NH CPU utilization prediction model. We use 90th percentile CPU utilization as it provides a good trade off between performance and resource consumption (see Sec. III-D2). Firstly, we fit our comprehensive grid measurements with a linear model. Then, we extend it with a port scaling factor in order to generalize its applicability to arbitrary physical topologies and randomly generated VNs.

A. Model for Grid Topologies

The results of Sec. III-D show that the CPU utilization of a FV scales with *(i)* the *FlowMod Add* message rate (as witnessed by the impact of flow length and flow request rate), *(ii)* the total number of virtual switches, and *(iii)* the number of virtual ports per switch. Furthermore, for a small number of tenants, *(iv)* the impact of this parameter can be assumed to be additive (see Sec. III-D7): the contribution of each additional tenant to the CPU utilization is equal to its contribution as a single tenant. As the number of virtual ports is used to represent the topology impact (e.g., edge density), we will only consider it when extending our model for arbitrary topologies (Sec. IV-B). We define our initial CPU prediction model as:

$$f_1(r, v, n) = \sum_{i=1}^{n} \left(c_0 + c_1\ r_i^{FM} \sqrt{v_i}\right), \tag{1}$$

where n is the number of tenants, r_i^{FM} is the total control plane *FlowMod Add* rate, which is fully defined with the flow request rate and the corresponding path lengths, (i.e., if we assume that the total number of different paths of a tenant i is X_i, then $r_i^{FM} = \sum_{j=1}^{X_i} r_j^i \cdot l_j$). The variable v_i gives the total number of virtual switches of a tenant i, and c_0 and c_1 are fitting coefficients. We multiply the parameters (r_i^{FM} and $\sqrt{v_i}$) since the required resources for processing each *FlowMod Add* message depend on the configuration of a VN (see Sec. II and Sec. III-D). To illustrate, if no messages are sent ($r_i = 0$ or $l_i = 0$), the size of a VN (i.e., parameter $\sqrt{v_i}$) should not have a significant impact on the CPU utilization.

Using a regression model minimizing the mean square error and the measurement data presented in Figs. 8a–8c, we obtain the coefficients $c_0 = 6.46$ and $c_1 = 2.84 \times 10^{-3}$. The cumulative distribution of the absolute fitting error and the corresponding median are shown in Fig. 9. The median of the absolute error is around $3.41\,\%$ and the maximum error is $12.4\,\%$ (*for scenario*: num. tenants = 5, $r = 430$, $k = 5 \times 5$, $v_i = 3 \times 3$, $l_i = 3$ and $p_i = 1$), witnessing the fitting accuracy.

B. Model Extension for Arbitrary Topologies

The initial model f_1 (Eqn. 1) assumes a physical grid topology as well as virtual grid networks. However, in practice,

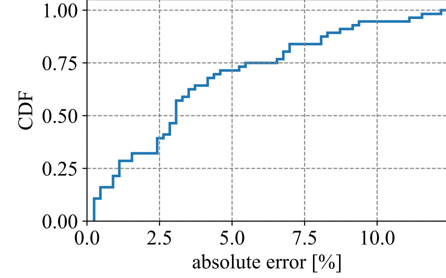


Fig. 9: Cumulative distribution function (CDF) of the absolute fitting error of the grid measurement data with a linear based model (Eg. 1).

other topologies are possible. For instance, tree-like topologies used in data center networks, ring-like topologies used in industrial networks and wide-area network topologies such as those from the Topology Zoo [29] have less dense structures. The average node degree of a ring topology is always 2, while the average node degree of a grid with 9 nodes is 2.67 [6].

We have seen that the number of virtual ports affects the NH CPU utilization (see Fig. 8e). As the average number of virtual ports of a VN directly corresponds to the average node degree of its topology, we use this notion to extend our model to arbitrary topologies. We linearly scale the grid-based CPU prediction with a per-tenant port scaling factor ϕ_i. We use a linear scaling factor ϕ_i because of the linear dependency observed in Fig. 8e. The extended model can then be formulated as:

$$f_2(r, v, n) = \sum_{i=1}^{n} \phi_i\left(c_0 + c_1\ r_i \sqrt{v_i}\right). \tag{2}$$

The intuition for defining the per-tenant port scaling factor ϕ_i is similar to the cross-multiplication rule in elementary arithmetic. We first divide the per-tenant grid-based prediction (based on Eqn. 1) with a parameter representing the average node degree in a grid topology (i.e., with p_i^e). This division "removes" the grid aspect in the prediction (details in Sec. IV-B1). Afterwards, again on a per-tenant basis, we multiply the newly obtained predictions with the average node degree of each tenant's virtual topology (i.e., with p_i). Mathematically, using a tuning parameter α, the per-tenant port scaling factor ϕ_i can be represented as:

$$\phi_i = \left(\frac{p_i}{p_i^e}\right)^{\alpha}, \tag{3}$$

where p_i is the average node degree of tenant's i VN:

$$p_i = \frac{\sum_{j=1}^{v_i} a_j^i}{v_i}, \tag{4}$$

where a_j^i is the number of virtual ports of virtual switch j.

1) Calculating p_i^e: The parameter p_i^e removes the grid aspect from the prediction. It tries to compute the average node degree of a grid topology with similar properties as the requested VN topology by a tenant. If a tenant requested a VN where the total number of nodes is a *square number*, i.e., the root of the total number of virtual switches is an integer, i.e., $y_i = \sqrt{v_i}, y_i \in \mathbb{Z}^+$ (e.g., $\sqrt{4} = 2$), an equivalent grid

[6] Excluding additional ports for connecting hosts to the virtual switches.

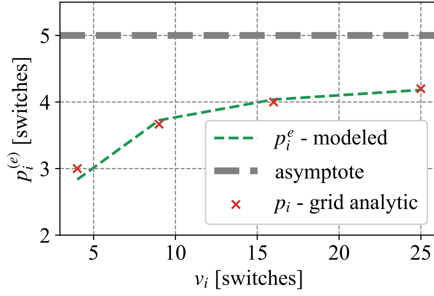


Fig. 10: Computation of p_i^e based on v_i. Values are interpolated (green dashed line) from the average number of virtual ports per switch for grid topologies (red crosses) and converge to 5 (thick dashed gray asymptote) as internal nodes in a grid topology connect to 4 other nodes and one host.

VN is simply $y_i \times y_i$ grid. Thus, it is easy to calculate the average node degree of the corresponding grid VN equivalent. For instance, if a tenant requested a VN with $v_i = 4$ nodes, the grid-like equivalent is a 2×2 grid, which has 12 virtual ports, thus $p_i^e = 12/4 = 3$.

If the total number of nodes of a requested VN is not a square number, we determine p_i^e based on the linear fitting of the average node degree of various VN sizes which were used in the measurements (i.e., 2×2, 3×3, 4×4, 5×5). The fitting is shown in Fig. 10. The red crosses correspond to the grid topologies used in the measurements. In this case, p_i^e simply corresponds to the average number of virtual ports per virtual switch for a grid topology of dimension $\sqrt{v_i}$. For intermediate values, since there is no direct grid-like VN equivalent, we simply fit the total number of virtual ports in the network piece-wise linearly and divide it by the number of nodes. This is shown by the green dashed line. Note that the values used for linear fitting are based on our measurement scenarios, which means that each virtual node also has an additional virtual port for connecting one virtual host to it. That is why the p_i^e values converge to 5, as shown by the thick gray dashed line.

2) Calculating α: As observed in Sec. III-D6, increasing the number of virtual ports also increases the observed CPU utilization linearly. However, the increase is not directly proportional, i.e., doubling the average number of virtual ports does not double the observed CPU utilization. Therefore, we introduce an exponential tuning parameter α, and we use it to improve the fitting of our scaling parameter ϕ_i to our measurements. To calculate α, we use the measurement data from Fig. 8e and we calculate α while minimizing the mean square error. We obtain $\alpha = 0.281$. Hence, replacing parameters in Eqn. 2 with their real values, we obtain our final model.

V. Model Evaluation

Having established our model, we now evaluate its accuracy using topologies and VNs that were not used during the measurements. This should allows us to quantify how the model can adapt to arbitrary physical topologies and randomly generated VNs. First, we introduce our evaluation scenario and explain how we generate a wide variety, i.e., random VN requests (Sec. V-A). Second, we describe the models used as comparison baseline (Sec. V-B). Finally, in Sec. V-C, we evaluate the precision of our CPU prediction model, and we also evaluate the effect of provisioning the resources with our model on the NH processing latency.

TABLE II: Physical topologies considered and their number of nodes and edges and their density. As a comparison, 5×5 and 6×6 grids have densities of 1.6 and 1.67 respectively.

Topology	# Nodes	# Edges	Density
Ring30	30	30	1
Internet2	34	42	1.23
NobelEU	28	41	1.46
Watts-Strogatz	30	60	2
Erdos-Reny15	30	63	2.1
Erdos-Reny30	30	136	4.53

TABLE III: Distribution of parameters for the final evaluation. $U(x, y)$ denotes a uniform distribution between x and y.

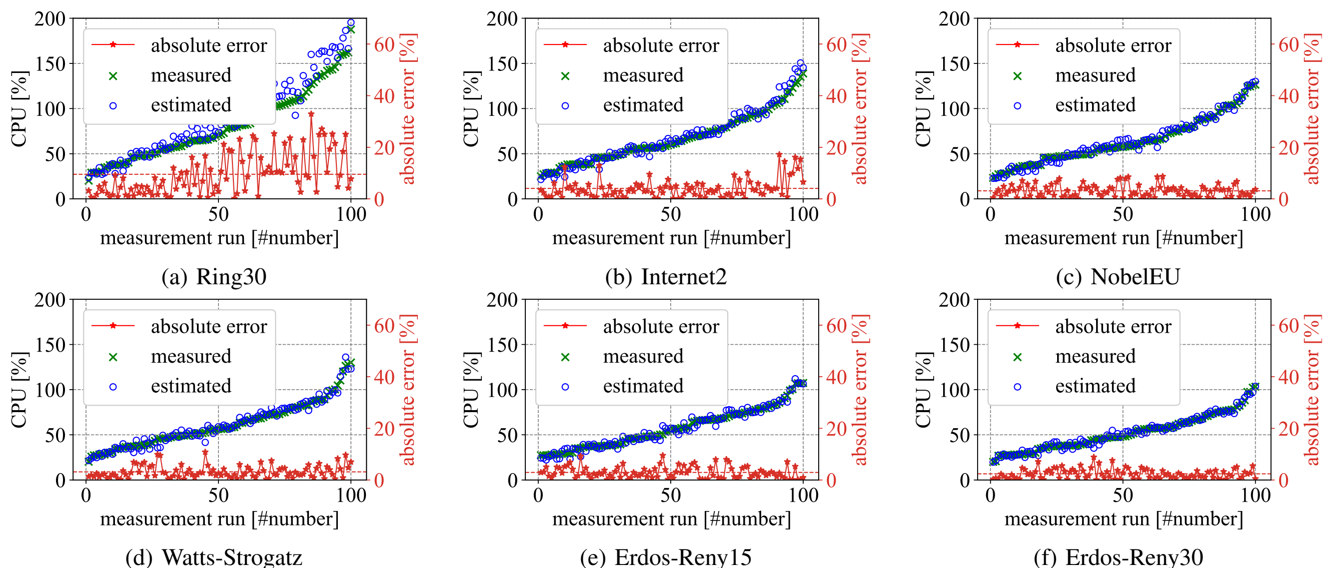
Scenarios	VN size [#nodes]	Flow rate [1/s]
1 – 20	$U(2, 25)$ – *Full*	$U(200, 600)$ – *Full*
21 – 40	$U(13, 25)$ – *Big*	$U(400, 600)$ – *High*
41 – 60	$U(2, 13)$ – *Small*	$U(400, 600)$ – *High*
61 – 80	$U(2, 13)$ – *Small*	$U(200, 400)$ – *Low*
81 – 100	$U(13, 25)$ – *Big*	$U(200, 400)$ – *Low*

A. Scenario

1) Topologies: We consider six physical topologies (Tab. II): two existing wide-area network topologies (Internet2 [24] and Nobel EU [25]), a typical industrial network topology (a 30 node ring) and three randomly generated network topologies: two Erdos-Reny [30] models with 30 nodes and an edge probability of 15 % and 30 %, and a Watts-Strogatz [31] model with 30 nodes with an average degree of 4. The selected topologies have a wide range of different edge densities (defined as the number of edges divided by the number of nodes), supporting the choice of these topologies as a representative set.

2) VN Requests: We consider that the total number of tenants during one run is static, and it ranges from 2 to 5. For each topology, and for each number of tenants, we define 100 different measurement scenarios. One measurement scenario is generated based on the *(i)* per-tenant VN size distribution and, *(ii)* a per-tenant flow request rate distribution. Depending on the scenario number, we use different parameter values as listed in Tab. III (as in Sec. III). This is done since using the full ranges as in the measurement section will lead to VN requests with different requirements. However, in multi-tenant cases, the average of all requirements of all tenants converges to expected mean values. Therefore, in order to evaluate more extreme cases, we consider cases for which all tenants may request only *big* or *small* networks, and only *high* or *low* flow request rates (as in Tab. III). As a consequence, we build four combinations, making a total of 5 different cases, with 20 scenarios each.

3) VN Connectivity: Based on a given VN size, edges in the VN are generated as follows. One node is randomly selected.



(a) Ring30 (b) Internet2 (c) NobelEU (d) Watts-Strogatz (e) Erdos-Reny15 (f) Erdos-Reny30

Fig. 11: (a)–(f) Estimated and measured 90th percentile CPU utilization for 100 randomly generated scenarios with 2-5 tenants using the considered topologies (i.e., Internet2, NobelEU, Ring, Erdos-Reny and Watts-Strogatz). The scenarios are sorted based on the measured CPU utilization in the ascending order. The absolute error is shown as red starred line, while the mean observed error is shown as horizontal red dashed line.

Out of all its neighbors, a new node is randomly selected and connected. Out of all the neighbors of existing nodes, a node is randomly selected and connected. The procedure stops as soon as enough nodes are connected. Finally, the per-tenant flow request rate is generated based on the values in Tab. III. We reuse the same flow generation procedure as in the measurement section.

B. Model Variations & Baseline

State-of-the-art VNF CPU resource prediction models generate their estimate while considering different input parameters [32], [33]. However, apart from the message rate, these parameters (e.g., IP source address, TCP destination port) often do not have an impact on the CPU utilization of an NH. Thus, if we directly apply these approaches for provisioning the CPU resources of an NH, they would generate their estimate solely based on the control plane message rate. Similarly, *state-of-the-art* NH CPU performance models [9], [10], [17] only consider the total control plane message rate as their input parameter for estimating the CPU utilization of an NH. Therefore, as our baseline, we use a linear model based on the total control plane message rate [9], which does not take into account the properties of the VNs. The model is defined as follows:

$$f_3(r,n) = c_o^B + c_1^B \sum_{i=1}^{n} r_i. \quad (5)$$

The coefficients c_0^B and c_0^B are obtained by a least mean square error fitting using the same measurement data set presented in Fig. 8; the coefficients are $c_0^B = 11.96$ and $c_0^B = 13.86 \times 10^{-3}$.

Furthermore, in order to evaluate separately the effect of the number of virtual switches and ports, we also consider our original f_1 model, which does not include the effect of virtual ports.

C. Evaluation Results

First, we evaluate the accuracy of our proposed model. We show that, even for arbitrary topologies and randomly generated VNs, the prediction error remains low (8–9 % on average). Finally, we compare the provisioning performance of our proposed prediction model, and the impact on the processing latency. Furthermore, we also compare the achieved accuracy and provisioning performance to the two baseline models.

1) Prediction Accuracy: Fig. 11 shows, for each physical topology, the measured and predicted CPU utilization for 100 randomly generated scenarios. It can be observed that the proposed model accurately predicts CPU utilization, even for randomly generated virtual topologies and unknown physical topologies. For instance, the mean absolute prediction errors per topology fall between 2.4–9.6 %, while the highest observed error was over 30 %. The maximal absolute error is observed for the ring topology, as it exhibits the most different characteristics compared to the grid (e.g., significantly lower edge density). Due to this difference, the maximal measured CPU values for the ring topology exceeded the maximal grid-based values, which were used for model fitting (more detailed explanation is in the next paragraph). For example, for the ring topology, we observed CPU values reaching up to 200%, while in the case of a grid topology the maximal values were around 150%. Thus, the predicted values were outside of the modeled range, thus the performance suffered the most in this case.

2) Topology Analysis: We observe that different topologies require different amount of CPU resources. For instance, *Ring30* requires the highest amount of CPU resources (up to $\sim$ 180 %) while *Erdos-Reny30* requires the least (up to $\sim$ 110 %). We can explain this in the following way: randomly generated VNs on physical topologies with a lower edge density (e.g., ring) typically have longer paths compared to the ones generated on more dense topologies. For instance, a

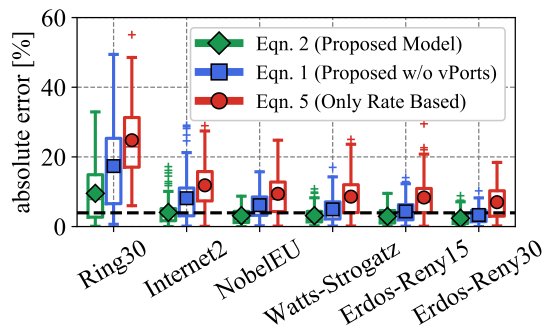


Fig. 12: Box plots of the prediction accuracy of the three models f_2 (Eqn. 2, proposed model), f_1 (Eqn. 1, proposed model without port scaling factor) and f_3 (Eqn. 5, only rate based model - *SotA*). The dashed gray line indicates the average fitting error of the grid measurements, thus, applying the model on different topologies with randomly generated requests on average introduces a slight penalty, i.e., the average error increases by a few percents.

VN with 4 virtual switches on a ring topology is always a line with a maximal path length of 4. On the other hand, a grid VN with the same amount of nodes (i.e., 2×2) on a physical grid topology has a maximum path length of 3. Therefore, for less dense topologies, a larger number of *FlowMod Add* messages are needed, in turn leading to higher CPU utilization. Thus, the observed and predicted mean CPU utilization values in Figs. 11b–11d are highly correlated with the density of the corresponding topologies (see Tab. II).

3) Mean Error and Baseline Comparison: Fig. 12 shows the prediction error of our proposed model f_2 (Eqn. 2) and of the two baseline models f_1 (Eqn. 1, proposed model without port scaling factor) and f_3 (Eqn. 5, only rate based), all models predict 90th percentile CPU utilization. The highest absolute errors are observed for the ring topologies as it differs the most from the measurement grid topology. For instance, the mean prediction error achieved for the ring topology is slightly higher compared to other topologies, i.e., around $9.5\,\%$ compared to $2.4\,\% - 4.1\,\%$ for other topologies. Furthermore, the baseline prediction models (i.e., Eqn. 1, and Eqn. 5) produce considerably higher prediction errors, as the mean prediction per topology varies between $3\,\%$ and $25\,\%$. This confirms our original motivation: certain NH functions depend on configurations of the tenant VNs — processing the same message in different network settings requires a different amount of resources.

4) Sources of Error: Fig. 13 shows heatmaps of the measured absolute error for the proposed model and the baseline models for all topologies and for different VN sizes. The proposed model performs quite constantly, having an overall mean absolute prediction error of around $4\,\%$. As the baseline models do not include all affecting parameters (i.e., total number of virtual switch and ports), they fail to perform in extreme cases. For instance, the rate baseline model (Eqn. 5, only rate based) reaches a mean absolute error of $40\,\%$ for small networks, which can lead to significant unpredictability of the performance perceived by tenants.

As the baseline (Eqn. 5, only rate based) was fitted based on the grid measurement data where VN sizes vary from 4 to 25 virtual switches, we would expect that it performs the best for middle sized VNs (e.g., with around 12-13 nodes). However, as it can be seen in Fig. 13c, this is not the case, as the baseline performed the best for virtual topologies with a higher number of nodes. This stems from the fact that (on average) our tree-like VN generation procedure produces VN requests with a lower amount of virtual ports compared to the grid VNs used in the measurement section. This makes CPU prediction of the baseline higher for all VN sizes, hence, the precision becomes worse for middle-sized VNs, and the best for larger VNs (see Fig. 13c). Our proposed model includes a per-tenant port scaling factor, hence, it can in general mitigate this effect among all VN sizes.

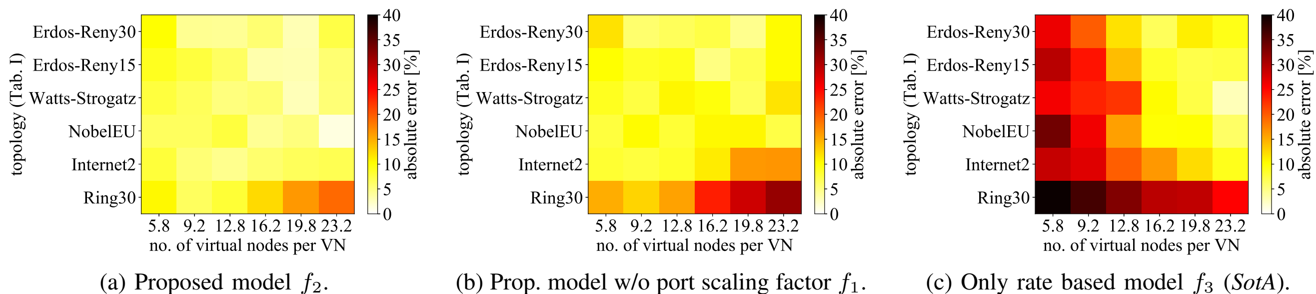
Overall, the rate-based solutions can perform decently in static and fixed environments, i.e., with fixed number of static tenants (and fixed VN configurations) generating constant control plane load. However, novel communication networks are envisioned to be dynamic and flexible, i.e., a tenant should be able to request arbitrary VNs (with a desired configuration) at any time for a certain duration. In contrast to our proposed solution, *state-of-the-art* approaches do not include all the crucial parameters (e.g., the error can reach over 60% depending on VN sizes), therefore, these solutions do not seem suitable for flexible and dynamic networks.

5) Effect of Evaluation Parameters: The proposed CPU prediction model depends on several input parameters: number of tenants, total number of virtual switches, the flow request rate, and the number of ports per virtual switch for each tenant. In order to evaluate whether the performance deviates with some of the considered parameters, we uniformly bin the data, and we show the mean absolute prediction error along with standard deviation and maximum value in Fig. 14. Overall, we can observe that the mean and maximal absolute error is higher when the measured CPU utilization is higher. There are two reasons behind this. Firstly, we show absolute errors, thus an absolute error of 10% at 100% is more highlighted on the figure compared to an absolute error of 5% at 50% (while relative errors are the same). Secondly, in case of a higher CPU utilization, the ring topology exhibits significantly higher absolute error compared to the other topologies (e.g., see Fig. 11a and Figs. 11b-11f). Thus, this significant error increase caused by extrapolating skews a bit data on Fig. 14. Further, since all these parameters are correlated (e.g., higher rates produces higher CPU utilization), we can observer the same trend for all parameters except the number of tenants.

D. Impact on Latency

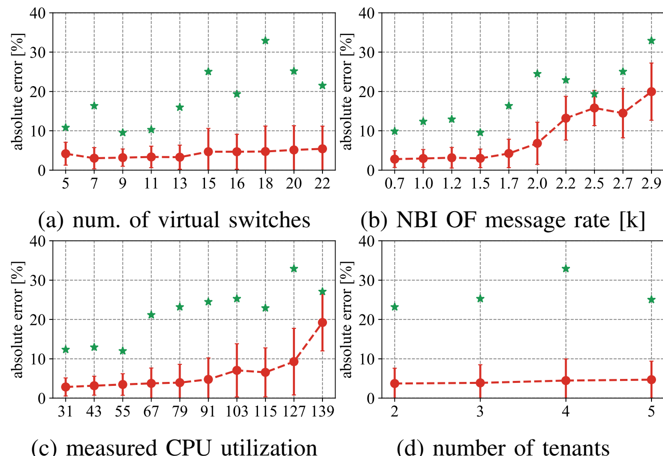
To evaluate the impact of CPU provisioning on the processing latency of the NH, we generate 100 scenarios per total number of tenants for Internet2 and Erdos-Reny15 topologies in the same manner as explained in Sec. V-A. Each measurement scenario is then run 4 times with 4 different provisioning strategies while tracking the processing latency profiles. We consider the following 4 provisioning strategies.

1) *Over-provisioning.* All physical resources are dedicated to the NH. This strategy is very resource inefficient but is expected to provide the best overall processing performance.
2) *Mean CPU Estimate.* The resources are provisioned based on the observed mean CPU utilization during the



(a) Proposed model f_2. (b) Prop. model w/o port scaling factor f_1. (c) Only rate based model f_3 (*SotA*).

Fig. 13: Impact of the number of virtual switches per tenant and topology type on the mean observed relative error for (a) proposed model and (b) the baseline. The date is separated in 6 uniformly created bins, and the mean values of each bin are shown.



(a) num. of virtual switches (b) NBI OF message rate [k]

(c) measured CPU utilization (d) number of tenants

Fig. 14: Absolute error dependency on (a) the average number of virtual switches per VN, (b) the total *FlowMod Add* rate, (c) the measured CPU utilization, and (d) the total number of tenants. The observed data on figures (a)–(c) is pre-processed by binning it into 10 equally sized bins. For each bin, mean and standard deviation are shown as error bars, while green stars indicate the corresponding maximum.

over-provisioned run. This corresponds to the *state-of-the-art* CPU performance models [9], [10].

3) *Proposed Model.* The resources are provisioned with the proposed improved model, i.e., with f_2 (Eqn. 2).
4) *Proposed Model with Additional Margin.* As our proposed model is not perfect (average fitting error is around 4%), it occasionally underestimates the CPU utilization, in turn, potentially increasing the processing time of an NH. In order to compensate for underestimation and non-perfect *cpulimit* precision, we also consider provisioning strategy based on the proposed model with an additional margin. For an additional margin we use the maximal fitting error, which is $12.4\,\%$ (see Fig. 9). Thus, the resources are provisioned with $f_2 + 12.4\,\%$. This approach ensures that we always provision the resources with a slightly higher value, thus, the impact on latency should always be negligible.

For each measurement run, we record the mean, median, maximal, and 90th percentile latencies. Fig. 15 shows the latency profiles achieved by the strategies for the two topologies and the four aforementioned statistical properties. As already shown in the introduction, provisioning with the mean CPU estimate (2nd set of box plots) is not sufficient as it incurs a big overall latency increase. Provisioning with the proposed model (3rd set of box plots) does not increase the median latency, however, the mean is increased as the maximal latencies are increased significantly. For instance, the maximal latency increases around 3 times, reaching values of around 100 ms. This is happening when the model underestimates the required CPU resources for a given measurement scenario. Thus, in certain time instances, the NH exceeds the allocated CPU resources, in turn triggering *cpulimit* to throttle the corresponding process.

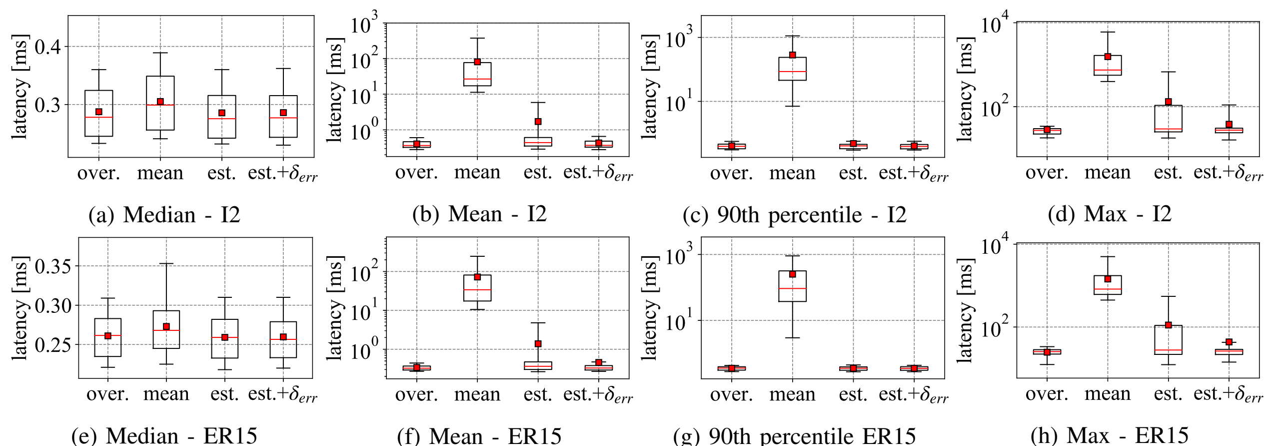
Provisioning the CPU resources with an additional margin (4th set of box plots) achieved almost the same processing performance as the over-provisioned case. For instance, the mean maximal latencies increased only from 24-29ms to 38-44ms. The mean and median latencies stayed the same. We can conclude that provisioning with an additional margin provides huge resource savings (in the most naive case $710\,\%$ less CPU capacity) while having an acceptable and still predictable impact on the processing performance for virtual network tenants.

VI. Related Work

Our approach for provisioning the resources of an NH is based on a carefully designed measurement based CPU prediction model. Since an NH can be considered as a VNF, we firstly describe *state-of-the-art* Virtual Networking Function (VNF) and NH resource prediction models, while highlighting their shortcomings. Furthermore, as our model is based on comprehensive NH measurements, subsequently, we cover existing works dealing with the benchmarking of NHs.

A. Resource Prediction Models

In order to reduce excessive power consumption in cloud computing systems through dynamic resource scaling, many authors focused on designing accurate VNF resource prediction models [32]–[37]. Although an NH can be considered as a VNF, these approaches cannot be directly applied. For instance, they often consider different input parameters (e.g., IP source address, TCP destination port etc.) [32], [33] which do not affect NH CPU utilization. Or they are based on already observed CPU samples [35]–[37]. However, as the VN configurations can change over time (e.g., number of virtual switches), the prediction performance of these models would also suffer.



(a) Median - I2 (b) Mean - I2 (c) 90th percentile - I2 (d) Max - I2

(e) Median - ER15 (f) Mean - ER15 (g) 90th percentile ER15 (h) Max - ER15

Fig. 15: (a)–(f) Impact of 4 different NH CPU provisioning strategies on the observed message processing time of an NH. For each run, four different statistical properties are considered: median, mean, 90th percentile, and maximal latency. As we have 400 different scenarios, each box plot represents 400 observed values of a certain statistical property.

On the other hand, there are also a few NH specific CPU prediction models [9], [10], [17]. However, there are two problems with these approaches. Firstly, they predict the average NH CPU utilization only. Using the average CPU utilization only does not count for potential variability in the overall CPU utilization. This can result in a significant performance degradation of the perceived tenant performance (as shown in Sec. I). Secondly, the prediction is only based on the control plane message rate. Thus, the algorithm cannot react to changes in the VN configuration parameters, e.g., the number of virtual switches or hosts. Furthermore, as these parameters have a significant impact on the utilization of NHs [8], the prediction performance would suffer in dynamic scenarios. On the contrary, our model predicts the 90th percentile of the CPU utilization, which does not incur a forwarding performance degradation on average. Moreover, our model accounts for performance-critical parameters such as the number of virtual switches or ports.

B. Hypervisors Benchmarks

Network hypervisor benchmarks have so far either focused on exploring the processing time of various control plane messages [6], [8], [15], [38]–[43] or on measuring the CPU utilization of NHs in various different settings [6], [8]–[10], [17]. In contrast to our benchmarks, evaluating how to predict NH CPU requirements based on different parameters, e.g., number of VNs, has been ignored so far.

Furthermore, some of the aforementioned studies suggest that the CPU utilization of NH and the processing time is only correlated with a subset of VN parameters, e.g., the number of tenants [9], [10] or the number of virtual and physical switches [8]. However, these studies consider only very basic combinations of virtual and physical network parameters and settings, e.g., a single-switch topology [9], [10], only a line topology [17], [40] or two-port switches [8]. Therefore, the impact of arbitrary topologies (in terms of the number of switches and the interconnecting links) is not considered. We perform comprehensive NH performance benchmarks (including latency and CPU) on various different and realistic physical and virtual network topologies. In particular, we investigate a multitude of impact factors. Furthermore, we tailor the benchmarks with the goal of detecting and learning the impact of various physical and VN parameters in a fast and efficient manner.

VII. Conclusion

Correctly provisioning the resources available to a *network hypervisor* (NH) is crucial for ensuring stable and predictable network performance for tenants. Yet, the state-of-the-art does not offer adequate solutions as they neglect bursty NH workloads or the impact of dynamic *virtual network* (VN) changes. Thus, in this article, with the goal of provisioning NH resources for flow embedding scenario, we design an accurate CPU prediction model based on comprehensive measurements that generalizes for different substrate topologies and virtual network requests. The proposed model exhibits high prediction accuracy as the mean average prediction error is overall around 4 %. Provisioning the resources with the corresponding model produces only a slight increase of tail latencies. For instance, on average, the maximal processing time increased from around 25ms to around 44ms. With our new model, it becomes possible to minimize the resource consumption or improve the overall utilization through accurate prediction of the required CPU resources of an NH.

From a more general point of view, we believe that our work sheds light on potential other applications where similar predictions are necessary. For example, softwarization of networking functions (e.g., firewall, NAT) and new networking architectures (e.g., SDN) increase the total number of deployed VNFs running in software on commodity hardware. Applying the same procedure to learn the performance profiles of various VNFs could potentially produce enormous savings while provisioning the resources in a centralized cloud where these functions run.

Future Work. The amount of required resources for normal operation of a VNF may be affected by many different parameters: e.g., number of clients within a firewall-protected network, number of flows a NAT has to process

etc. Manually learning what parameters have an effect, and which are the most affecting ones for various VNFs (with different implementation architectures) can be time-consuming and cost-inefficient. Therefore, new solutions are needed that are capable of learning the most affecting parameters in an automated and online manner.

Furthermore, in this article we focus on providing a solution capable of provisioning network hypervisor CPU resources in smaller environments. However, it is still not clear how our solution scales, and if it is generalizable. Thus, one of our future steps is to deploy our solution on larger setups in an automated manner.

Acknowledgment

This work has been funded in part by the European Union's Horizon 2020 research and innovation program (grant agreement No 647158 - FlexNets), in part by the Bavarian Ministry of Economic Affairs, Regional Development and Energy as part of the project 5G Testbed Bayern mit Schwerpunktanwendung eHealth and in part by DFG under the grant numbers KE1863/6-1 and KE1863/8-1. This work reflects only the authors' view and the funding agency is not responsible for any use that may be made of the information it contains.

References

[1] Andreas Blenk, Arsany Basta, Martin Reisslein, and Wolfgang Kellerer. Survey on network virtualization hypervisors for software defined networking. *IEEE Communications Surveys & Tutorials*, 18(1):655–685, 2016.

[2] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtala, Daniel Obenshain, Dmitri Perelman, and Yee Jiun Song. Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services. pages 635–651, 2016.

[3] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4visor: Lightweight Virtualization and Composition Primitives for Building and Testing Modular Programs. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 98–111, New York, NY, USA, 2018. ACM. event-place: Heraklion, Greece.

[4] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 58–72, New York, NY, USA, 2016. ACM. event-place: Florianopolis, Brazil.

[5] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs HÃűlzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 183–197, New York, NY, USA, 2015. ACM. event-place: London, United Kingdom.

[6] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, 1:132, 2009.

[7] Ibrahim Afolabi, Tarik Taleb, Konstantinos Samdanis, Adlen Ksentini, and Hannu Flinck. Network slicing and softwarization: A survey on principles, enabling technologies, and solutions. *IEEE Communications Surveys & Tutorials*, 20(3):2429–2453, 2018.

[8] A. Blenk, A. Basta, W. Kellerer, and S. Schmid. On the Impact of the Network Hypervisor on Virtual Network Performance. In *2019 IFIP Networking Conference (IFIP Networking)*, pages 1–9, May 2019.

[9] Christian Sieber, Arsany Basta, Andreas Blenk, and Wolfgang Kellerer. Online resource mapping for sdn network hypervisors using machine learning. In *NetSoft Conference and Workshops (NetSoft), 2016 IEEE*, pages 78–82. IEEE, 2016.

[10] Christian Sieber, Andreas Obermair, and Wolfgang Kellerer. Online learning and adaptation of network hypervisor performance models. In *Integrated Network and Service Management (IM), 2017 IFIP/IEEE Symposium on*, pages 1204–1212. IEEE, 2017.

[11] NM Mosharaf Kabir Chowdhury, Muntasir Raihan Rahman, and Raouf Boutaba. Virtual network embedding with coordinated node and link mapping. In *IEEE INFOCOM 2009*, pages 783–791. IEEE, 2009.

[12] Andreas Fischer, Juan Felipe Botero, Michael Till Beck, Hermann De Meer, and Xavier Hesselbach. Virtual network embedding: A survey. *IEEE Communications Surveys & Tutorials*, 15(4):1888–1906, 2013.

[13] Amaury Van Bemten, Nemanja Ðeric, Johannes Zerwas, Andreas Blenk, Stefan Schmid, and Wolfgang Kellerer. Loko: Predictable latency in small networks. In *Proceedings of the 15th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT). ACM*, 2019.

[14] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[15] Ali Al-Shabibi, Marc De Leenheer, Matteo Gerola, Ayaka Koshibe, Guru Parulkar, Elio Salvadori, and Bill Snow. Openvirtex: Make your virtual sdns programmable. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 25–30. ACM, 2014.

[16] Yoonseon Han, Thomas Vachuska, Ali Al-Shabibi, Jian Li, Huibai Huang, William Snow, and James Won-Ki Hong. Onvisor: Towards a scalable and flexible sdn-based network virtualization platform on onos. *International Journal of Network Management*, 28(2):e2012, 2018.

[17] Nemanja Ðerić, Amir Varasteh, Arsany Basta, Andreas Blenk, and Wolfgang Kellerer. Sdn hypervisors: How much does topology abstraction matter? In *2018 14th International Conference on Network and Service Management (CNSM)*, pages 328–332. IEEE, 2018.

[18] OpenFlow Switch Specification. Version 1.0. 0 (wire protocol 0x01). *Open Networking Foundation*, 2009.

[19] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[20] Ryu SDN Framework Community. Ryu sdn framework. https://osrg.github.io/ryu/, 2017. Accessed: 2020-07-21.

[21] G Rodola. Psutil package: a cross-platform library for retrieving information onrunning processes and system utilization. *Google Scholar*.

[22] Maciej Kuzniar, Peter Peresini, and Dejan Kostic. What you need to know about sdn control and data planes. Technical report, 2014.

[23] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. What you need to know about sdn flow tables. In *International Conference on Passive and Active Network Measurement*, pages 347–359. Springer, 2015.

[24] David Hock, Matthias Hartmann, Steffen Gebert, Michael Jarschel, Thomas Zinner, and Phuoc Tran-Gia. Pareto-optimal resilient controller placement in sdn-based core networks. In *Proceedings of the 2013 25th International Teletraffic Congress (ITC)*, pages 1–9. IEEE, 2013.

[25] S. Orlowski, M. Pióro, A. Tomaszewski, and R. Wessäly. SNDlib 1.0–Survivable Network Design Library. In *Proceedings of the 3rd International Network Optimization Conference (INOC 2007), Spa, Belgium*, April 2007. http://sndlib.zib.de, extended version accepted in Networks, 2009.

[26] Yennun Huang, Chandra Kintala, Nick Kolettis, and N Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 381–390. IEEE, 1995.

[27] Sachin Garg, Aad Van Moorsel, Kalyanaraman Vaidyanathan, and Kishor S Trivedi. A methodology for detection and estimation of software aging. In *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*, pages 283–292. IEEE, 1998.

[28] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[29] Simon Knight *et al.* The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765 –1775, October 2011.

[30] Paul Erdös and Alfréd Rényi. On random graphs, i. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.

[31] Duncan J Watts and Steven H Strogatz. Collective dynamics of small-world networks. *nature*, 393(6684):440, 1998.

[32] Houda Jmila, Mohamed Ibn Khedher, and Mounim A El Yacoubi. Estimating vnf resource requirements using machine learning techniques. In *International Conference on Neural Information Processing*, pages 883–892. Springer, 2017.

[33] Albert Mestres, Eduard Alarcón, and Albert Cabellos. A machine learning-based approach for virtual network function modeling. In *2018 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, pages 237–242. IEEE, 2018.

[34] Kalika Suksomboon, Masaki Fukushima, Shuichi Okamoto, and Michiaki Hayashi. A dilated-cpu-consumption-based performance prediction for multi-core software routers. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 193–201. IEEE, 2016.

[35] Rashid Mijumbi, Sidhant Hasija, Steven Davy, Alan Davy, Brendan Jennings, and Raouf Boutaba. Topology-aware prediction of virtual network function resource requirements. *IEEE Transactions on Network and Service Management*, 14(1):106–120, 2017.

[36] Rashid Mijumbi, Sidhant Hasija, Steven Davy, Alan Davy, Brendan Jennings, and Raouf Boutaba. A connectionist approach to dynamic resource management for virtualised network functions. In *2016 12th International Conference on Network and Service Management (CNSM)*, pages 1–9. IEEE, 2016.

[37] Lun Tang, Xiaoyu He, Peipei Zhao, Guofan Zhao, Yu Zhou, and Qianbin Chen. Virtual network function migration based on dynamic resource requirements prediction. *IEEE Access*, 7:112348–112362, 2019.

[38] Yoonseon Han. A framework for development, operations, and management of sdn-based virtual networks.

[39] Dmitry Drutskoy, Eric Keller, and Jennifer Rexford. Scalable network virtualization in software-defined networks. *IEEE Internet Computing*, 17(2):20–27, 2013.

[40] G. N. Nurkahfi, A. Mitayani, V. A. Mardiana, and M. M. M. Dinata. Comparing flowvisor and open virtex as sdn-based site-to-site vpn services solution. In *2019 International Conference on Radar, Antenna, Microwave, Electronics, and Telecommunications (ICRAMET)*, pages 142–147, 2019.

[41] H. Jin, G. Yang, B. Yu, and C. Yoo. Talon: Tenant throughput allocation through traffic load-balancing in virtualized software-defined networks. In *2019 International Conference on Information Networking (ICOIN)*, pages 233–238, 2019.

[42] AL-Badrany Zainab and Mohammed Basheer Al-Somaidai. Latency evaluation of an sdn controlled by flowvisor and two heterogeneous controllers. In *International Conference on New Trends in Information and Communications Technology Applications*, pages 3–16. Springer, 2020.

[43] Andreas Blenk, Arsany Basta, and Wolfgang Kellerer. Hyperflex: An sdn virtualization architecture with flexible hypervisor function allocation. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 397–405. IEEE, 2015.

Nemanja Đerić received his B.Sc. degree in electrical and computer engineering in 2014 from University of Belgrade, while he received M.Sc. degree in communication engineering from Technical University of Munich in 2016. Afterwards, he joined the Chair of Communication networks at Technical University of Munich, where he is currently pursuing Ph.D. degree and working as a Research and Teaching Associate. His current research interests are Software Defined Networking (SDN) and Network Virtualization (NV).

Amir Varasteh received his M.Sc. degree in Information Technology from Sharif University of Technology, Tehran, Iran in September 2015. He joined the Chair of Communication Networks at Technical University of Munich (TUM) as a PhD candidate in February 2017. His current research focuses on Network Function Virtualization, Software-Defined Networks, and Mobile Edge Computing.

Amaury Van Bemten received the B.Sc. degree in engineering, in June 2013, and the M.Sc. degree in computer science and engineering, in June 2015, both from the University of Liège, Liège, Belgium. He joined the Chair of Communication Networks at the Technical University of Munich (TUM) as a PhD candidate in September 2015. His current research is focused on data plane isolation techniques for predictable latency in software-defined networks.

Andreas Blenk received the Dr.-Ing. degree (Ph.D.) from the Technical University of Munich in 2018 with distinction. He joined the Chair of Communication Networks with the Technical University of Munich in 2012, where he is currently a Senior Researcher. Since March 2019, he is also a Senior Research Fellow at the Communication Technologies group of the Faculty of Computer Science of the University of Vienna. His research is focused on self-driving, flexible and predictable virtual and software-defined networks, as well as data-driven networking algorithms.

Wolfgang Kellerer (M′96 – SM′11) is a Full Professor with the Technical University of Munich (TUM), heading the Chair of Communication Networks at the Department of Electrical and Computer Engineering. Before, he was for over ten years with NTT DOCOMO's European Research Laboratories. He received his Dr.-Ing. degree (Ph.D.) and his Dipl.-Ing. degree (Master) from TUM, in 1995 and 2002, respectively. His research resulted in over 200 publications and 35 granted patents. He currently serves as an associate editor for IEEE Transactions on Network and Service Management and on the Editorial Board of the IEEE Communications Surveys and Tutorials. He is a member of ACM and the VDE ITG.