Online learning and adaptation of network hypervisor performance models

Christian Sieber, Andreas Obermair, Wolfgang Kellerer
Chair of Communication Networks
Department of Electrical and Computer Engineering
Technical University of Munich, Germany
Email: {c.sieber, wolfgang.kellerer}@tum.de

***Abstract*—Software Defined Networking (SDN) paved the way for a logically centralized entity, the SDN controller, to excerpt near real-time control over the forwarding state of a network. Network hypervisors are an in-between layer to allow multiple SDN controllers to share this control by slicing the network and giving each controller the power over a part of the network. This makes network hypervisors a critical component in terms of reliability and performance. At the same time, compute virtualization is ubiquitous and may not guarantee statically assigned resources to the network hypervisors. It is therefore important to understand the performance of network hypervisors in environments with varying compute resources.**

In this paper we propose an online machine learning pipeline to synthesize a performance model of a running hypervisor instance in the face of varying resources. The performance model allows precise estimations of the current capacity in terms of control message throughput without time-intensive offline benchmarks. We evaluate the pipeline in a virtual testbed with a popular network hypervisor implementation. The results show that the proposed pipeline is able to estimate the capacity of a hypervisor instance with a low error and furthermore is able to quickly detect and adapt to a change in available resources. By exploring the parameter space of the learning pipeline, we discuss its characteristics in terms of estimation accuracy and convergence time for different parameter choices and use cases. Although we evaluate the approach with network hypervisors, our work can be generalized to other latency-sensitive applications with similar characteristics and requirements as network hypervisors.

I. Introduction

Software Defined Networking (SDN) is transforming the way we think about networking. While traditional networks consist mostly of proprietary devices of a single vendor, SDN promises network devices with open and programmable interfaces. The networking devices are reduced to simple forwarding devices and, ideally, this allows a network operator to choose freely between interchangeable devices of different vendors. Furthermore, this programmability enables the separation of the control and data plane of the network by moving control decisions to a logically centralized controller. The literature refers to the controller as *SDN controller* or *Network Operating System* (NOS) [1]. By pushing forwarding rules to the devices and polling statistic counters, a NOS can monitor and reconfigure the network as a whole. OpenFlow [2] is a wide-spread SDN protocol for pushing rules to devices and retrieving statistics.

Network virtualization in SDN is the idea of introducing a virtualization layer between the NOS and the network devices through network hypervisors (NHV). A NHV is a piece of software which allows several *tenants* to share the control over one physical data plane. To do so, the operator of a tenant's NOS first requests a slice of the network, which for example can be a subset of the traffic defined by a VLAN header, and afterwards provides the NHV with the IP address of his controller. The NOS will be able to push forwarding rules which apply to its slice of the traffic. But the NOS will not be aware of other traffic in the network, as the NHV isolates the control decisions from the different tenants.

The network virtualization layer intercepts and translates all control decisions and statistic messages in the network, which makes it a critical part of the infrastructure with high requirements on availability and delay. Therefore, understanding the performance of NHVs in different environments is of importance. The central questions of this paper are: Which model describes the performance of a NHV best? How can this model be trained at runtime? And if there is a change in available resources, how can this be detected and the model consequently be adapted to the new environment? This is important, as an overloaded NHV will increase the delay for network reconfigurations and network statistic updates. One way to solve this are offline benchmarks of every NHV instance in every deployed environment. However, this approach is not scalable to a larger number of platforms and hypervisors.

In [3], we introduce the concept of learning performance models of network hypervisors at runtime based on measured utilization and message counters. In this paper we go a step further and extend our approach to environments with fluctuating resource availability. There are multiple reasons for dynamic resource assignments in virtualized environments. For example, a dynamic resource increase can happen with vertical auto-scaling. There, the infrastructure increases the assigned resources when an application hits a predefined threshold. A temporary decrease in assigned resources can happen for example when higher priority tasks are scheduled. Virtual machine migration techniques are also known for decreasing resources to speed up the incremental state transfer between two physical machines.

The contribution of this work is described as follows. First, we fine-tune a previously proposed network hypervisor

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

performance model and introduce a novel model which allows more detailed estimations. Second, we propose an online machine learning pipeline to train the performance models and adapt the models in case of resource fluctuations. Third, we evaluate the pipeline by exploring the parameter space and give guidelines for different use cases.

The proposed pipeline can be deployed as part of an autonomous orchestration layer which keeps track of current usage and capacity of running NHV instances. This allows load-balancing of network tenants based on their control message rate and that way prevent over- and underutilization of the NHV instances.

This paper is structured as follows. We first give the background and related work of this area of research in Section II. In Section III we present a formalized model of the discussed scenario. In Section IV we introduce the proposed learning pipeline, the performance models and the chosen machine learning techniques. In Section V we discuss the experimental evaluation methodology and in Section VI we present the results of the evaluation. In Section VII and VIII we conclude by deducing deployment guidelines from the findings and give an outlook on future work in this area.

II. Background & Related Work

Next, we discuss the state of the art in the area of network hypervisors and application resource demand prediction/estimation in cloud environments.

In our previous work [3], we introduce the concept of learning performance models of NHV at runtime. For this, we first developed the benchmarking framework hvbench [4] and setup two different NHV (FlowVisor [1] and OpenVirteX [5]) in our testbed in different physical environments, e.g. with different number of CPU cores. Afterwards, we emulated a Poisson message arrival process with increasing message rates and applied an online machine learning approach to compare the learning performance of three different models. The results show that a negative exponential model is able to learn the performance of the network hypervisors fast and with a low prediction error.

In [6], [7], [8], [9], the authors discuss and survey SDN network virtualization in general and the network hypervisor placement problem (HPP) specifically. HPP targets the amount of needed hypervisor instances and the placement of the hypervisor instances inside the network. In [10], [11], the control plane latency and monitoring overhead of different SDN network hypervisor architectures is evaluated.

In [12], [13], [14], the authors show that machine learning techniques can accurately predict future database query resource requirements based on previously monitored queries. But these approaches are not applicable to our problem as per-message resource consumption and hardware details are not available.

A related area of research is the performance prediction and classification of applications through online machine learning from the perspective of the infrastructure provider. In [15], the authors propose and compare different approaches for per-application resource usage prediction. In [16], [17], [18], the authors propose machine learning pipelines to provide medium-term resource demand predictions and elastic resource scaling. [19] provides an extensive survey on anomaly detection in cloud environments in general. But anomaly detection on performance models is not considered in [19].

This work extends the state of the art by tackling the dynamic budget estimation problem from the point of view of an NHV orchestrator with limited control and information about the underlying virtual resources and the resources assigned to the NHV. The challenge here is to not only learn an accurate performance model, but also to recognize when the NHV is subject to an increase or decrease in available resources. Furthermore, false positives should be avoided while at the same time be sensitive enough to adapt the model fast to the new amount of resources.

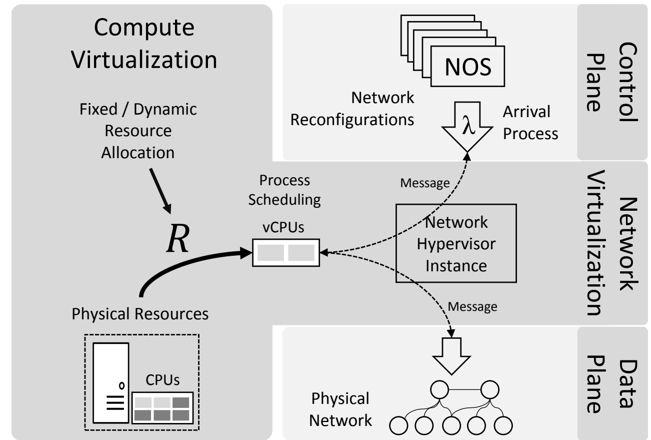


Figure 1. System model overview. The figure illustrates the relationship between compute and network virtualization. A process scheduler on the physical machine assigns R resources to the network hypervisor instance which uses the resources to process and, if necessary, forwards messages from the network's control and data plane.

III. System Model

Next, we formalize the relationship between the physical resources, the virtualized NHV instance and the control plane messages. Figure 1 depicts the abstract system model. Table I summarizes the nomenclature. The system model consists of a compute- and a network control-centric part. The figure shows that from the perspective of the compute on the left, the network hypervisor instance is a process consisting of one or multiple threads running inside a container, e.g. Docker, or a virtual machine, e.g. KVM. The container or virtual machine with the hypervisor inside is assigned resources R by the host's process scheduler based on a fixed or dynamic scheduling strategy. For example, out of six physical CPUs, the scheduler assigns the NHV two CPUs, so called *virtual* CPUs or *vCPUs*, which the NHV is allowed to utilize each up to 50 %. In our work, we assume a dynamic scheduling strategy where the resource provider assigns resources dynamically between an upper limit R_{max} and a lower guaranteed limit of R_{min}. ΔR

Table I
NOMENCLATURE & VARIABLES

Nomenclature	
$fm, ot(ot = \{fr, er, sp, sf, po\})$	Message types.
λ	Total arrival rate of messages.
$\lambda_{ot} = w_{ot} \cdot \lambda$	Arrival rate of ot messages.
$\lambda_{fm} = w_{fm} \cdot \lambda$	Arrival rate of fm messages.
$w_{ot}, w_{fm}, w_{ot} + w_{fm} = 1$	Message type distribution in λ.
$R, R_{min}, R_{max}, \Delta R$	Resources assigned to NHV.
B	Total message budget of the NHV.
$\rho \| 0 \leq \rho \leq 1$	Instantaneous resource utilization.
$\rho_{max} \| \rho_{max} \leq 1$	Max allowed system utilization.
$\rho()$ or $\rho(\lambda) = \rho$	Performance model.

denotes a change of resource allocation. This paper considers compute resources (CPU) as the only limited resource.

From the perspective of the network control, the NHV instance is part of a virtualization layer which consists of one or multiple NHV instances in between the network's control and data plane. This virtualization layer allows multiple NOS to share the control of the physical network. In OpenFlow, a NOS can control the network by adding or updating forwarding rules in the devices using flow modification messages. Furthermore, it can request the current state of the network, such as throughput, by sending statistic request messages. Each of the reconfiguration or state requests has to pass through a NHV instance, which uses its assigned resources to process the messages. We assume a negative exponential (*Poisson*) message arrival process with an average rate λ.

Some OpenFlow message types are computational more complex than others, e.g. simple echo requests compared to flow modifications. For our evaluation, we consider the OpenFlow message types *feature request* (*fr*), *echo request* (*er*), *port stats* request (*sp*), *flow stats request* (*sf*), *packet out* (*po*) and *flow modification* (*fm*). During our experiments with OpenVirteX and FlowVisor, we noticed that flow modifications exhibit a much larger cost per message than each of the other message types. Hence, for the remainder of this paper we summarize $\{fr, er, sp, sf, po\}$ as *other* (ot) and denote the combined message rate as λ_{ot}.

IV. PROPOSED LEARNING PIPELINE

In the following we discuss the proposed learning pipeline. The objective of the learning pipeline is to estimate the maximum performance in terms of message rate λ the NHV instance can process with the currently assigned resources R. This maximum rate is also denoted as the message budget B. Furthermore, it must detect and adapt the reported message budget to resources fluctuations ΔR. The adaption can be gradual, by decreasing the importance of older samples compared to more recent samples. This approach does not require to detect a ΔR. But the adaptation can also by rapid, by implementing a ΔR-detection, which detects a ΔR and, if necessary, discards the now invalid model.

We first give a general overview of the elements in the pipeline, including the ΔR-detection based on a Support Vector Machine (SVM). Afterwards we discuss the performance

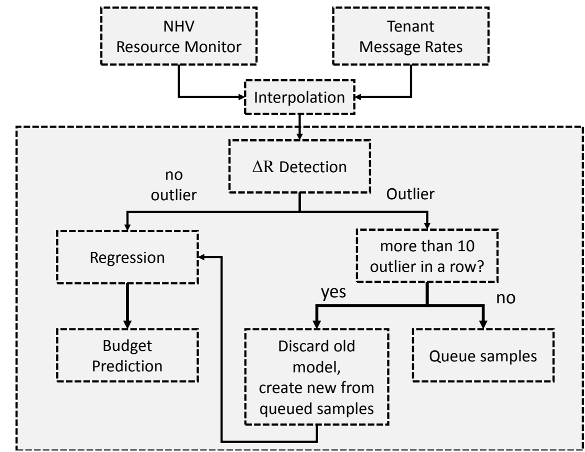


Figure 2. Proposed learning pipeline. Input parameters are the resource utilization and tenant message rate counters. Output is the estimation of the current message rate budget. Outlier detection is used to invalidate the current model in case of large ΔR.

model based on the overall message rate λ from our previous work and how we adapt it for this work. Subsequently we discuss the sample weighting function for the gradual adaptation. At the end of this section we discuss an extended performance model which distinguishes different message types (λ_{fm}, λ_{ot}).

Figure 2 presents the proposed pipeline. There are two input data sources. One is the NHV resource monitor which measures the exact amount of cumulative CPU time used by the NHV. The second one is the cumulative message counters provided by the hypervisor which denote how many messages are processed by the NHV per message type and per tenant. Resource usage and message counter values are provided unsynchronized with a frequency of 1 Hz each.

The samples are then processed as follows. The first step is combining the asynchronously collected samples of cumulative resource usage $\Sigma\rho$ and message counters $\Sigma\lambda$ by linear interpolation. For this, two cumulative resource usage samples before and after the point in time t of a message rate sample $s^t_{\Sigma\lambda}$ are taken, $s^{<t}_{\Sigma\rho}$ and $s^{>t}_{\Sigma\rho}$. Afterwards, the cumulative resource usage sample $s^t_{\Sigma\rho}$ for the message rate sample $s^t_{\Sigma\lambda}$ is linearly interpolated. As a result, we have a sample $(s^t_{\Sigma\lambda}, s^t_{\Sigma\rho})$ of the message counter with the approximate *cumulative* resource usage for those messages at time t. Finally, the instantaneous resource usage s_ρ for the message rate at t, i.e. s_λ, is calculated by element-wise subtraction $(s^t_{\Sigma\lambda}, s^t_{\Sigma\rho}) - (s^{t-1}_{\Sigma\lambda}, s^{t-1}_{\Sigma\rho}) = (s_\lambda, s_\rho)$.

In the second step, the ΔR-detection checks if the current performance model is still valid and not invalidated by a change in resources ΔR. To do so, it calculates the difference between the sample (s_λ, s_ρ) and the current performance model $\rho()$ by $\rho(s_\lambda) - s_\rho$. On this error, it applies a one-class support vector machine (SVM) [20] for outlier detection. The SVM is configured with a radial basis function kernel (RBF), $\gamma = 0.1$ as kernel parameter and we consider 10 % of the training data as outliers ($\nu = 0.1$). Then, if none of the samples in the last $T_{thres} = 10$ seconds fit to the current model, we assume a big resource change ΔR and invalidate the current

model. If no resource change is detected, the sample is added to the previous samples of the current model and the model is re-trained with updated sample weights where old samples become less important than the newer samples. A maximum of 120 samples, equal to the last 120 s, are stored and older samples are discarded. The idea behind this is, that if there is a small ΔR which can not be detected by the ΔR-detection, we instead adapt the model over time using the sample weight function and the limited memory of 120 s. The training of the model uses orthogonal distance regression (ODR) [21] which considers errors in the data in both dimensions, i.e. in the measurement of the resource usage and the message counters.

In the following we introduce the performance model $p()$, the sample weighting function $w(t)$, the extended performance model $p_{ext}()$ and the budget B in detail.

A. Performance Model

We define $\rho(\lambda)$ as the (injective) relationship between the message arrival rate λ and the induced utilization of the NHV instance. For example, $\rho(10000) = 1$ describes a resource usage of 100 % of the resources R assigned to the NHV by the physical resource scheduler at message rate of 10000. We denote $\rho^{-1}()$ as the inverse of $\rho()$ which translates resource usage to λ. The message rate an instance can process at specific maximum utilization ρ_{max} is denoted as budget B:

$$B = \rho^{-1}(\rho_{max}) \tag{1}$$

We set $\rho_{max} = 0.90$ for the evaluation. Accordingly, the budget is defined as $B = \rho^{-1}(0.90)$.

In [3] we previously compared different NHV performance models for $\rho()$. The results show that a negative exponential performance model can describe the performance of a NHV accurately. Hence, we choose the following negative exponential model from [3] for this work:

$$\rho(\lambda) := \Theta_c \cdot (1 - e^{-\Theta_a \cdot \lambda}) + \Theta_b \tag{2}$$

While in general the model yields good results, it exhibits unstable behavior in cases where a majority of the samples report a low utilization, e.g. $\leq 15\,\%$. This comes from the fact that there are many possible regression results for the coefficients Θ_b and Θ_c which fit well to low utilization samples but do not allow accurate estimation of the budget at ρ_{max}. Hence, based on training experiments, we fixed $\Theta_b = 0$ and $\Theta_c = 1.6$.

B. Gradual Adaptation & Weight Function

If the resources R assigned to the NHV instance do change, but the ΔR-detection does not detect it, the performance model has to gradually adapt over time. We perform the gradual adaptation of the model by reducing the importance of each sample in the regression based on its age. In Eq. 3 we define a function $w(t)$ which describes a negative exponential decay process. t denotes the age of the sample in seconds. The negative exponential decay process was chosen based on preliminary experiments.

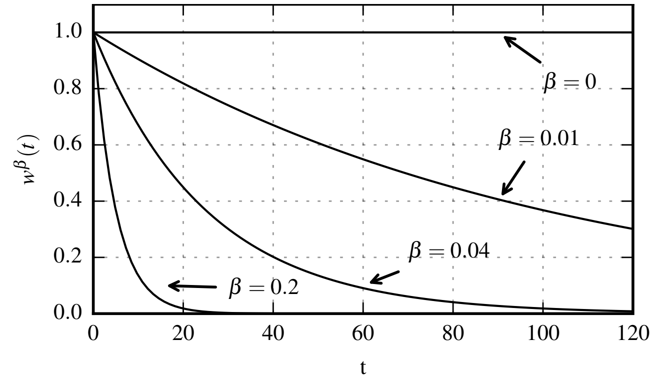


Figure 3. Sample weight function $w(t)$ (Eq. 3) describes the important of each sample in the regression depending on the age t of the sample in seconds. Parameter β describes how fast the weights of the samples decrease. The figure illustrates w for $\beta = [0.0, 0.2, 0.04, 0.01]$.

$$w^\beta(t) := e^{-t \cdot \beta} \tag{3}$$

Figure 3 illustrates w for four different values of β. For $\beta = 0$, w becomes $w(t) = 1$, which makes the weight of a sample independent of its age and all 120 samples are equally important in the regression. For $\beta = 0.2$, the importance of a sample decreases fast so that after approximatively 5 seconds, a sample has half of the weight of a new sample. In this paper, we evaluate the impact of the weight function on the adaptation for 15 different values between 0.0 and 0.2 ($\beta = [0.0, 0.015, .., 0.2]$).

C. Extended Performance Model

The performance model discussed so far only considers the relationship between the total message rate λ and the resulting utilization ρ. Hence, in cases when it is desired to differentiate the cost between *fm* and *ot* message rates, the simple model is not sufficient. Next, we introduce an extended model which considers λ_{fm} and λ_{ot} separately.

The following equations define the extended performance model. The model is made up of separate equations for λ_{fm} (Eq. 4) and λ_{ot} (Eq. 5) with each consisting of two parts, a linear part and a negative exponential part. The two parts are added together using the parameters A, B, C and D:

$$\rho_{ext,fm}(\lambda_{fm}) = A \cdot (\Theta_A \cdot \lambda_{fm}) + B \cdot \Theta_B \cdot (1 - e^{-\Theta_C \cdot \lambda_{fm}}) \tag{4}$$

$$\rho_{ext,ot}(\lambda_{ot}) = C \cdot (\Theta_D \cdot \lambda_{ot}) + D \cdot \Theta_E \cdot (1 - e^{-\Theta_F \cdot \lambda_{ot}}) \tag{5}$$

Eq. 6 describes the total utilization as sum of the utilization induced by fm and ot message rates:

$$\rho_{ext}(\lambda_{fm}, \lambda_{ot}) = \rho_{ext,fm}(\lambda_{fm}) + \rho_{ext,ot}(\lambda_{ot}) \tag{6}$$

The parameters A, B, C and D allow for fine-tuning the performance model between linear and negative exponential behavior. From preliminary experiments for this work we concluded, that this gives the best possible result, as depending on the platform, hypervisor and message type (fm or ot),

the performance behavior varies between linear and negative exponential behavior. $B = 0.7$ and $D = 0.7$ provided the best balance between linear and negative exponential model in our experimental environments. Therefor we set $B = 0.7$ and $D = 0.7$ in the evaluation. Following the constrains $A + B = 1$ *and* $C + D = 1$ we set $A = 0.3$ and $C = 0.3$.

To learn the parameters Θ_A, Θ_B, Θ_C and Θ_D from the samples, we use the following two equations (Eq. 7) and (Eq. 8). Two independent ODR regressions are used to learn the coefficients of both equations.

$$\rho_{ext,lin}(\lambda_{fm}, \lambda_{ot}) = (\Theta_A \cdot \lambda_{fm}) + (\Theta_D \cdot \lambda_{ot}) \quad (7)$$

$$\rho_{ext,exp}(\lambda_{fm}, \lambda_{ot}) = \Theta_B \cdot (1 - e^{-\Theta_C \cdot \lambda_{fm}}) + \Theta_E \cdot (1 - e^{-\Theta_F \cdot \lambda_{ot}}) \quad (8)$$

V. Evaluation Methodology

In the following we present the methodology used for the evaluation of our proposed architecture. We first present a general overview of our experimental set-up. Afterwards we discuss how we evaluate the accuracy of the budget estimation. At the end of this section we show the message arrival process we use to emulate virtual tenant networks and their messages.

A. Experimental Set-up

Figure 4 describes the experimental set-up consisting of the benchmark *hvbench* [22], a simple emulated network, the NHV resource monitor *hvmonitor* [22] and a message bus which distributes the measurement samples to the learning pipeline (implemented with *kafka* [23]). The pipeline in turn outputs the model for the budget estimation. A control component adjusts the overall tenant message rates and message type mixes based on a random walk process (upper left corner). Furthermore, it adjusts the assignment of R to the NHV process. For our set-up we choose to implement the resource R assignment by scaling the CPU frequency of the NHV host through the Linux CPU governor. For example, the experiment is started with assigning the maximum possible resources $R_{max} = 3.2\ GHz$ to the NHV process. After 60 s, R is reduced to $\frac{R_{max}}{2} = 1.6\ GHz$. For this, the *performance* CPU governor has to be activated so that the Linux kernel always scales the CPU to the maximum configured CPU frequency.

During the experiment, *hvmonitor* queries the total time the process used the CPU ($s^t_{\Sigma\rho}$) with an accuracy of 10 ms and sends it to the message bus. Furthermore, the simple emulated network answers all OpenFlow messages it receives from the NHV with static, preconfigured, responses. Additionally, *hvbench* reports the message counters ($s^t_{\Sigma\lambda}$) to kafka. The following hardware and software set-up is used for the evaluation presented in this paper: Intel(R) Core(TM) i5-3470 CPU with a maximum frequency of 3.20 GHz, 8 Gb RAM and Ubuntu 14.04.4 LTS. *FlowVisor* is used as hypervisor in version 1.4.0, *hvbench* and *hvmonitor* in version 0.1.0.

B. Budget Estimation Error

The budget estimation error ϵ is defined as the difference between the true budget, denoted as *ground truth*, and the estimated budget. We use offline benchmarks to determine

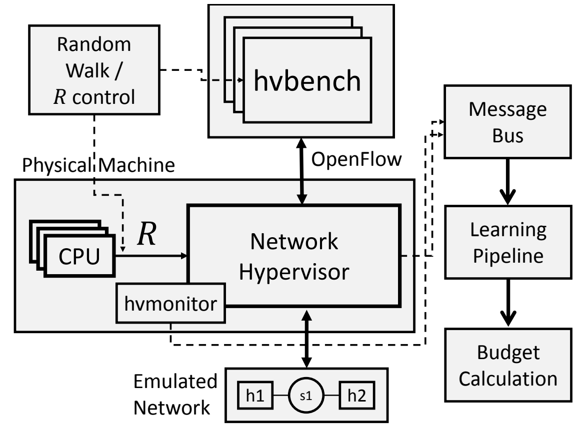


Figure 4. Experimental set-up consisting of the benchmark *hvbench*, a simple emulated network (2 hosts, 1 switch), the NHV resource monitor *hvmonitor* and a message bus which distributes the measurement samples to the learning pipeline, which in turn outputs the model for the budget calculation.

an approximation of the true budget of a specific NHV on a specific hardware platform. For this, we linearly increase the message rate λ in small steps until the monitor component reports an average utilization of ρ_{max}. Based on the results of the offline benchmark, we can define the relative estimation error ϵ as: $\epsilon = \frac{|GroundTruth - B|}{B}$

C. Message Arrival Process

We configure *hvbench* with a Poisson message arrival process and we use two random walk processes to model the change of λ and (w_{ot}, w_{fm}) over time. At the beginning of the experiment run we set λ to $\rho^{-1}(0.5)$ based on the ground truth. Then, each 5 s we use random number generator to decide to keep λ constant with a chance of 40 % or change it with a chance 60 % according to the following rule:

$$\lambda = \begin{cases} \lambda \cdot 0.9 & \text{if } \rho(\lambda) > 0.5 \\ \lambda \cdot 1.1 & \text{if } \rho(\lambda) < 0.5 \end{cases}$$

When the message type distribution random walk process is activated, we adjust (w_{ot}, w_{fm}) also every 5 s. With a chance of each $\frac{1}{3}$, we either decrease or increase w_{fm} by 10 %. With a chance of $\frac{1}{3}$, we keep w_{fm} constant. w_{ot} is updated accordingly ($w_{ot} + w_{fm} = 1$).

VI. Evaluation

Next we evaluate the estimation accuracy of the proposed pipeline by using the methodology defined in the previous section. The main performance metrics here are the relative budget estimation error ϵ and the model convergence time after a change in resources. If not otherwise stated, the experiments were conducted in the described test-bed (Section V). For space reasons we focus on the results for FlowVisor as network hypervisor. The results for OpenVirtX do not considerable differ from the presented ones.

A. Budget Estimation Error without ΔR

In the following we discuss the budget estimation error ϵ for different combinations of β and constant available resources R. We evaluate β in the range of $[0, 0.2]$ with a step size of 0.015 and R in the range of $[1.6, 3.2]$ with a step size of 0.1. The result of a combination of β and R is presented as median over 20 runs. For each run, after a warm-up phase of 120 s, a sample is taken each second for a period of 180 s and the error averaged over all samples. We use the random walk described in Section V-C for λ. w_{ot} and w_{fm} we keep constant at $w_{ot} = 0.5$ and $w_{fm} = 0.5$. Figure 5 illustrates the mean estimation error over the evaluated parameter space of β and R. The results show that on average the estimation error is 5.8 % with a standard deviation of 2.5 %. Furthermore, we conclude from the figure that the estimation error depends on R. For example, while for $R = 1.6$ we observe ϵ to be $3.8\,\%$ on average, ϵ for $R = 3.0$ is on average $9.5\,\%$. The lowest error can be observed for $R = 1.6$ with $\epsilon = 2.6\,\%$ Additionally, the figure suggests a minor correlation between the estimation error and parameter β. However, the maximum (Pearson) correlation we observe is for $R = 2.6$ with 0.18.

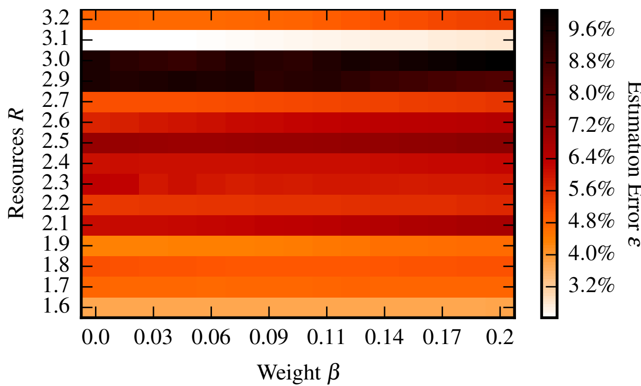


Figure 5. Mean estimation error for $R = [1.6, 3.2]$ and $\beta = [0, 0.2]$ for constant R. $R = 1.6$ exhibits the lowest error with $\epsilon = 2.6\,\%$. $R = 3.0$ with $\epsilon = 9.5\,\%$ the highest. No correlation between e and β.

Two main conclusions can be drawn from the figure. First, we observe only an insignificant influence of β in the evaluated parameter range. Therefore β can be chosen freely in the evaluated range in cases when R is constant. Second, the error depends on R and ranges between 2.6 % and 9.5 %. Next, we discuss the influence of parameter β on the convergence time in scenarios where R varies over time.

B. Convergence Time after ΔR

In compute environments with dynamic assigned resources R, it is important for the learning pipeline to quickly adapt to changes of R. We focus on the use case of a sudden decrease of R and measure the time period between the decrease and the point in time the model reaches a relative error of less than 10 % again. First we illustrate the convergence time by example for a gradual adaptation, afterwards we explore the parameter space with and without ΔR-detection.

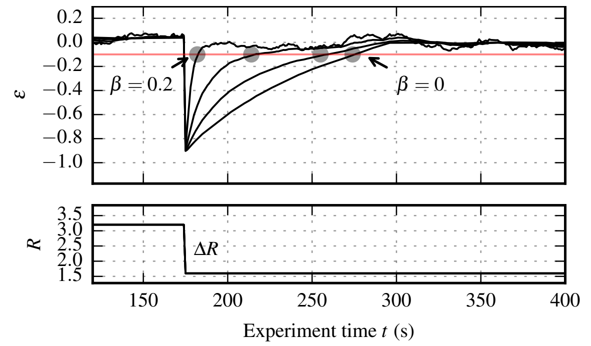


Figure 6. Convergence time after a resource change $\Delta R = 1.5$ for $\beta = [0.0, 0.0143, 0.0429, 0.2]$. The horizontal (red) line marks an low error threshold of $\epsilon = -0.1$. Circles mark the time when the model adapted.

Figure 6 illustrates the convergence time by example for four different values of β, with a decrease in CPU frequency of 1.4 GHz ($\Delta R = 1.4$) and a relative error threshold of 10 % ($\epsilon = 0.1$). A training phase with a duration of 125 s is omitted in the figure. At 175 s into the experiment, the available resources are decreased from 3.0 GHz to 1.6 GHz. Two observations can be made from the figure. First, up to 175 s into the experiment all four model instances can estimate the available budget with a high accuracy of $\epsilon \leq 0.04$. Second, the choice of β has a significant influence on the convergence time. For $\beta = 0.2$, which only considers recent samples, the model adapts rapidly (7 s) to the new resources. However, as few samples have a strong influence on the regression, we observe a higher variance in the figure. For $\beta = 0.0$ it takes 99 s to reach the low-error threshold and no variance is visible.

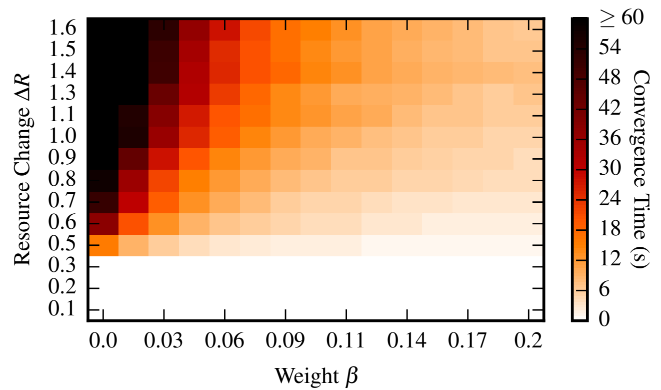
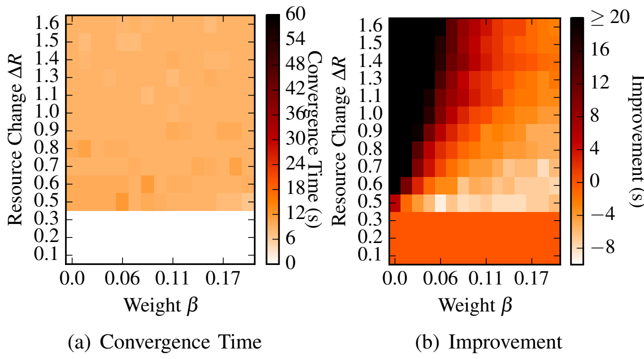


Figure 7. Evaluation of convergence time **without ΔR-detection** for different ΔR and β. β and ΔR are divided in 15 equally-spaced values. Convergence times increases with decreasing values of β and increasing values for ΔR.

Next, we evaluate the convergence time *without* ΔR-detection for different values of the sample weight parameter β and the amplitude of resource change ΔR. Figure 7 depicts the relationship between $\beta = [0, 0.2]$ and $\Delta R = [0.1, 1.6]$ with a step size of 0.015 and 0.1, respectively, and the convergence time. Each value for a combination of ΔR and β is presented as the median of 20 runs on a grey-scale. Note that all values of a convergence time $\geq 60s$ are displayed in black. The figure shows that with increasing β, i.e. with a faster decay of the



(a) Convergence Time

(b) Improvement

Figure 8. Evaluation of convergence time **with ΔR-detection** for different ΔR and β. β and ΔR are divided in 15 equally-spaced values. Convergence times are homogeneous with minor variations for almost all parameters.

weight of a sample, the model converges to a state of low estimation error faster for most combinations of ΔR and β. Furthermore, the figure illustrates that the model converges slower if the amplitude of the resources change ΔR is larger. For small ΔR, i.e. $\Delta R = \{0.5, 0.6\}$, a low error threshold is reached in less than 4 s, while a combination of large values of ΔR and small values of β can result in an average convergence time of up to 95 s.

The findings of Figure 7 motivate the need for a sophisticated ΔR-detection to accelerate convergence in cases of large ΔR and low β. With activated ΔR-detection, if a change is detected, the sample buffer is flushed and the model retrained with the outlier samples and any subsequent samples as described in Section IV. Figure 8(a) depicts the relationship between β, ΔR and the convergence time *with* ΔR-detection. Figure 8(b) illustrates the difference between Figure 7 and 8(a). Compared to the case without ΔR-detection, we observe a decrease in convergence time for 46 % of the investigated combinations of β and ΔR, mostly in the upper-left triangle of the figure. On average, the decrease is 16.6 s. Furthermore, the standard deviation over all combinations decreases from 21.2 s to 3.9 s From the figure we conclude, that with ΔR-detection a wider range of values for β can be selected while maintaining a low convergence time after ΔR.

However, the figure also illustrates that the convergence time can increase, especially for high values of β and low ΔR found in the lower-right triangle of the figure. On average, the increase is 3 s. This is due to the way outliers are treated in the pipeline. After an outlier is detected, the outlier sample is stored in a separate buffer and not used for the regression. When we observe ten outliers in a row, we signal a detected ΔR and discard the current performance model and use the outlier buffer to learn a new model. But if we do not observe ten outliers in a row, the samples in the outlier buffer are discarded. Hence, there are less samples available to learn from in cases where an *undetected* change in resources happened and this increases the convergence time. This effect could be mitigated by increasing the sensitivity of the SVM. However, increasing the sensitivity increases the chance of falsely detected resources changes and unnecessary re-learning.

C. Extended Performance Model

The extended performance model enables the estimation of the message budget per message type. In the following section we discuss the accuracy of the extended performance model. For the evaluation of the extended model we use two random walk processes for the message generator, one random walk process which controls the overall mean message rate (λ) and one which controls the allocation of how many messages of each type are send (λ_{fm} and λ_{ot}). The allocated resources R are constant. Nevertheless, ΔR-detection is turned on to simulate the full proposed pipeline. Each experiment run consists of a warm-up phase of 400 s where no model training was performed. After the warm-up phase, 600 samples are collected at 1 Hz and used in the training of the model. Hence, for the evaluation we take a snapshot of the extended model coefficients at 1000 s into the experiment. Offline benchmarks provide accurate ground truth. β is fixed to 0.0.

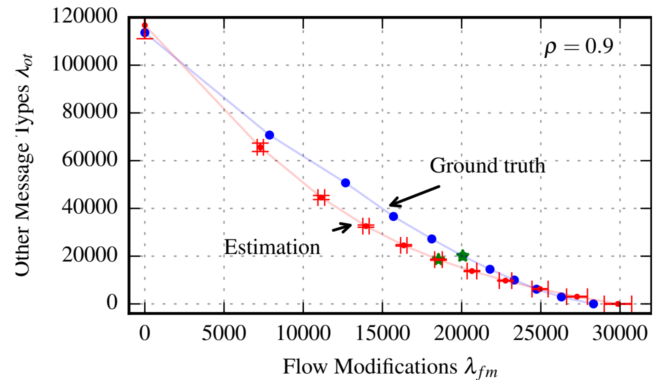


Figure 9. Budget estimation accuracy of different message type distributions using the extended model. The blue dots depict the ground truth from offline benchmarks. Red dots show the estimation results of 21 random experiment runs. The two green stars highlight the 0.5 allocation, i.e. $\lambda_{fm} = \lambda_{ot}$.

Figure 9 presents the results. The (blue) markers illustrate offline benchmarks, i.e. the ground truth. For the offline benchmark we selected 11 allocations of λ_{fm} and λ_{ot} with $(\lambda_{fm}, \lambda_{ot}) \in [(a \cdot \lambda, (1-a) \cdot \lambda) | \forall a \in [1, 0.9, .., 0]]$, denoted as Υ. Each allocation is presented as a distinct marker. For example, the upper left dot represents a composition of $\lambda_{fm} \approx 1 \cdot 115000 = 115000$ and $\lambda_{ot} = (1-1) \cdot 115000 = 0$. The confidence intervals of the offline benchmarks are omitted as they would not be visible on the presented scale. The links between the (blue) markers are for better readability and do not represent measurements.

The estimation result of 21 random experiment runs per allocation are shown in red. For the estimation we use the coefficients of the extended model at 1000 s into the experiment run to approximate $\rho_{ext}^{-1}(0.9)$ for the allocations of λ_{fm} and λ_{ot} defined in Υ. The confidence intervals for the estimations are indicated with error bars in the direction of λ_{fm} and λ_{ot}. For most evaluated allocations we observe stable estimation results and therefore most of the confidence intervals are hardly visible. From the figure we conclude, that for allocations with a share of ≥ 0.5 in favor of λ_{fm}

Table II
DEPLOYMENT GUIDELINES

Use Cases	Static R, Stable distribution	Dynamic R and/or unstable distribution
Overall budget sufficient	Simple model, $\beta \geq 0.1$	Normal model with SVM $\beta \leq 0.1$
Budget estimation per message type required	Simple & ext. model, accept cost per type inaccuracy	If unstable distribution, extended model. If stable distribution, simple & ext. model.

($\lambda_{ot} \leq 20.000$ to $\lambda_{fm} \geq 20.000$), the extended model can accurately estimate the message budget. For shares ≥ 0.5 in favor of λ_{ot}, it becomes less accurate and varies between underestimating, e.g. at $\lambda_{ot} = 40.000$ to $\lambda_{fm} = 15.000$, and overestimating the utilization, e.g. where λ_{fm} is close to 0. Confidence intervals increase if either λ_{fm} or λ_{ot} dominates λ, but in general are small.

VII. SUMMARY & DISCUSSION

The evaluation results show that the choice of model, pipeline configuration and parameters can be optimized depending on the use case. Next, we summarize the results and discuss how to achieve optimal budget estimation results for different use cases.

In Section VI-A we discuss the general budget estimation error with constant resource assignment. The results show that the sample weight parameter has no significant effect on the estimation error in the evaluated parameter range and that the error ranges between 2.6 % and 9.5 %.

In Section VI-B we evaluate the convergence time after a change in assigned resource with and without ΔR-detection. For this we measure the time it takes for the pipeline to accurately estimate the current budget after ΔR. Without ΔR-detection, for minor ΔR and a large value of β, we observe a fast convergence, whereas large changes and a low value of β show low convergence time. With ΔR-detection, the results exhibit a homogeneous convergence time in the evaluated parameter range. On the one side, we conclude that the ΔR-detection speeds up the convergence time considerable after a large change in resources assignment and low β. On the other side, the ΔR-detection slows down the convergence time for minor changes.

The evaluation of the extended model in Section VI-C shows that accurate estimation of the resource consumption per message type is possible. In particular, the results shows that the asynchronously collected message counters and resource usage samples are sufficient for the regression to learn the model coefficients in scenarios where the message type distribution is not constant.

Next, we discuss the following questions: When do I choose the extended model and which values do I use for the parameters? Based on the evaluation, we identify three criteria which dictate the choice of model and parameters. First, is it enough to know the overall budget or do I need the budget per message type? Second, do I expect the available resources R to be constant or are they likely to change frequently? Third, is the message type distribution of the incoming messages mostly stable or is it likely to be unstable? Table II gives guidelines based on these three criteria.

If the overall budget is sufficient and R is mostly static with a stable message type distribution, then the simple model with $\beta \geq 0.1$ is sufficient. However, note that a large value of β increases the influence of measurement noise on the estimation accuracy. If the overall budget is sufficient, but R is likely to change and/or the distribution of message type is unstable, use the normal model with SVM, but select $\beta \leq 0.1$.

If you require budget estimation per message type and R is either static or dynamic and you observe a stable message type distribution, then deploying the simple and extended model in parallel is the best choice. However, the stable message type distribution will not contain enough information to accurately learn the cost per message type.

If you require budget estimation per message type and R is either static or dynamic and the message type distribution is changing over time, then the extended model gives accurate estimation results per message type. However, note that in our experiments the extended models in general converged slower than the simple model. Furthermore, note that for best estimation results the trade-off parameters between linear and negative exponential behavior might require adjustments to the systems at hand. For example, during our experiments we noticed that in environments with power saving settings turned on, the relationship between λ and ρ tends to be less linear and more negative exponential.

In general, the estimation accuracy also depends on the range of samples seen by the learning process. The estimation becomes better if many samples close to 90 % utilization exist. If the samples are dominated mostly by low utilization samples, e.g. $\rho \leq 10\,\%$, the estimation will be worse. This can present a challenge where very large message budgets are allocated to tenants at the same time. However, we expect that in a realistic deployment the budget allocated to one tenant is far less than the overall budget and as our previous study shows [3], samples with an utilization of about $20\,\% - 25\,\%$ are already well suited for estimation using the negative exponential model.

VIII. CONCLUSION & OUTLOOK

In this paper we propose and evaluate an online machine learning pipeline for the capacity estimation of network hypervisor instances in dynamic cloud environments. The evaluation shows that the learned performance model provides accurate estimations of the message rate budget at run-time. Furthermore, a reduction or increase of available resources assigned to the network hypervisor is detected by the pipeline and the estimations are adapted accordingly. The proposed pipeline is an important step towards autonomous scaling and load-balancing of virtualized SDN control planes. Future work in this area should investigate the convergence time of the extended model and evaluate the trade-off between SVM sensitivity and convergence time in more detail.

References

[1] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. Mckeown, and G. Parulkar, "FlowVisor: A network virtualization layer," OpenFlow Consortium, Tech. Rep., 2009.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[3] C. Sieber, A. Basta, A. Blenk, and W. Kellerer, "Online resource mapping for sdn network hypervisors using machine learning," in *2nd IEEE Conference on Network Softwarization (NetSoft 2016)*, June 2016.

[4] C. Sieber, A. Blenk, A. Basta, and W. Kellerer, "hvbench: An open and scalable sdn network hypervisor benchmark," in *Workshop on Open-Source Software Networking (OSSN)*, June 2016.

[5] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, W. Snow, and G. Parulkar, "OpenVirteX: a network hypervisor," in *Proc. Open Networking Summit (ONS)*, Santa Clara, CA, Mar. 2014.

[6] A. Blenk, A. Basta, J. Zerwas, and W. Kellerer, "Pairing sdn with network virtualization: The network hypervisor placement problem," in *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*. IEEE, 2015.

[7] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani, "Data center network virtualization: A survey," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 2, pp. 909–928, 2013.

[8] R. Jain and S. Paul, "Network virtualization and software defined networking for cloud computing: a survey," *IEEE Communications Magazine*, vol. 51, no. 11, pp. 24–31, 2013.

[9] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. J. Jackson *et al.*, "Network virtualization in multi-tenant datacenters." in *NSDI*, 2014, pp. 203–216.

[10] A. Blenk, A. Basta, J. Zerwas, M. Reisslein, and W. Kellerer, "Control plane latency with sdn network hypervisors: The cost of virtualization," *IEEE Transactions on Network and Service Management*, 2016.

[11] G. Yang, K. Lee, W. Jeong, and C. Yoo, "Flo-v: Low overhead network monitoring framework in virtualized software defined networks," in *Proceedings of the 11th International Conference on Future Internet Technologies*. ACM, 2016.

[12] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson, "Predicting multiple metrics for queries: Better decisions enabled by machine learning," in *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 2009, pp. 592–603.

[13] K. Lee, A. C. König, V. Narasayya, B. Ding, S. Chaudhuri, B. Ellwein, A. Eksarevskiy, M. Kohli, J. Wyant, P. Prakash *et al.*, "Operator and query progress estimation in microsoft sql server live query statistics," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1753–1764.

[14] P. Xiong, Y. Chi, S. Zhu, H. J. Moon, C. Pu, and H. Hacgm, "Smartsla: Cost-sensitive management of virtualized resources for cpu-bound database services," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 5, May 2015.

[15] A. Matsunaga and J. A. Fortes, "On the use of machine learning to predict the time and resources consumed by applications," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE Computer Society, 2010.

[16] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, "Agile: Elastic distributed resource scaling for infrastructure-as-a-service," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, 2013, pp. 69–82.

[17] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 5.

[18] Z. Gong, X. Gu, and J. Wilkes, "Press: Predictive elastic resource scaling for cloud systems," in *2010 International Conference on Network and Service Management*. IEEE, 2010, pp. 9–16.

[19] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance anomaly detection and bottleneck identification," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, p. 4, 2015.

[20] L. M. Manevitz and M. Yousef, "One-class svms for document classification," *Journal of Machine Learning Research*, vol. 2, no. Dec, pp. 139–154, 2001.

[21] P. T. Boggs, R. H. Byrd, and R. B. Schnabel, "A stable and efficient algorithm for nonlinear orthogonal distance regression," *SIAM Journal on Scientific and Statistical Computing*, vol. 8, no. 6, pp. 1052–1078, 1987.

[22] "hvbench & hvmonitor," https://github.com/csieber/hvbench.

[23] "Apache Kafka," http://kafka.apache.org/.