NFVPerf: Online Performance Monitoring and Bottleneck Detection for NFV

Priyanka Naik, Dilip Kumar Shaw, Mythili Vutukuru
Department of Computer Science and Engineering, Indian Institute of Technology, Bombay
Email: {ppnaik, dilip13, mythili}@cse.iitb.ac.in

***Abstract*—Network Function Virtualization (NFV) is a new trend in networking, where network functions are moving from custom hardware appliances to software implementations running on virtual machines (VMs) hosted on commodity hardware. While the benefits of NFV such as cost reduction and increased agility are well understood, doubts still exist on whether a software implementation can match up to the high performance that hardware appliances deliver. In this context, network operators would benefit from frameworks that monitor performance and identify bottlenecks in Virtual Network Function (VNF) implementations obtained from vendors. While several techniques already exist to identify performance issues in cloud-based applications, most of them either use hardware resource utilizations to identify hot-spots (making them incapable of detecting non-hardware performance bottlenecks) or rely on application specific measurements (which may not be exposed by VNFs to vendors always). This paper describes NFVPerf, a performance monitoring and bottleneck detection tool for NFV. NFVPerf works as part of a cloud that hosts a NFV deployment, and takes a configuration file specifying the basic architecture of the VNF as input. It sniffs packets on all VM-to-VM communication paths, computes per-hop throughputs and delays, and uses these "black-box" measurements alone to identify performance bottlenecks (including software bottlenecks) in real time, without requiring any instrumentation of the VNF. Further, NFVPerf can be customized to any VNF implementations with just configuration changes. Our evaluation of NFVPerf shows that it can monitor performance and identify bottlenecks in an NFV deployment, with high accuracy and minimal overhead. We believe that a system like NFVPerf would form a great addition to cloud management systems in the era of NFV.**

I. Introduction

Network Function Virtualization (NFV) [1] is a new network architecture model, where network functions that are traditionally implemented as custom hardware appliances are now being implemented in software that runs in virtual machines (VMs), and is hosted on commodity servers or clouds. Examples of network functions being virtualized include firewalls [2], load balancers, and a variety of signaling and control plane elements in telecommunication service provider networks. The telecom industry is currently abuzz with several vendors providing prototype Virtual Network Functions (VNFs, e.g., Connect-em [3]), consortia developing suitable cloud-based platforms to host these VNFs (e.g., OpenStack [4], OPNFV [5]), and operators trying to find a way to migrate from physical network functions to VNFs.

978-1-5090-0933-6/16/$31.00

The recent interest in NFV has been spurred by the advent of faster CPUs and techniques for efficient packet processing in software (e.g., Intel DPDK [6]), NFV is expected to save costs for operators because software implementations are cheaper to build and maintain than hardware appliances, especially given the increasing complexity of network functions. Further cost savings come from the fact that VNFs can be easily scaled on demand to accommodate increased load, while hardware appliances are often over-provisioned to account for future increase in demand [7]. NFV also makes it easier to add new features, improving the flexibility and agility of network services. However, the excitement around NFV is tempered by doubts over performance: it is not clear if software appliances will have the high performance and resiliency of hardware. Ideally, NFV should provide the same (or similar) level of performance, availability, and SLA compliance that operators are accustomed to from physical network functions, to make NFV a clear winner for operators.

In a typical NFV deployment, a network operator sets up an NFV infrastructure (NFVI, typically a private or public cloud), obtains VNFs from vendors that build the software, and installs the VNFs on the NFVI. The components of the VNF are typically installed over several physical servers, and the network is configured to correctly forward the packets along the VNF forwarding chain (i.e., the sequence of VNF components through which a request flows). The operator must then orchestrate and manage the VNF components throughout their lifecycle, to ensure good performance and SLA compliance. For example, the operator must spawn new VMs and scale the VNFs to meet increased load. The operator must restart, repair, or replace failed VNFs. All these actions need a basic mechanism to begin with: a way to monitor performance of the VNF and identify performance bottlenecks.

Most cloud operators [8] identify performance bottlenecks by monitoring hardware resource utilizations, or other application-specific metrics obtained from instrumenting the application itself (§II). For example, some cloud services use thresholds on the utilizations of resources to identify hot-spots, and spawn a new VM if the utilization of the bottleneck resource of the service exceeds the threshold. Other research uses request service times or other measurements from the application to estimate system capacity and detect overload

(§II). However, both these techniques are not sufficient to identify all performance issues. For example, a performance bottleneck in a VNF could occur due to software issues (e.g., multiple threads contending for a lock, or a "noisy neighbor" VM), and may not always correspond to an increase in hardware resource utilization [9]. Further, given the variety of VNFs and vendors building them, coupled with a lack of standardized interfaces analogous to SNMP (Simple Network Management Protocol) in exporting application-specific metrics [10], the operator may not have easy access to application layer metrics from the VNF. This situation calls for a tool that can monitor application-layer performance and identify performance bottlenecks or SLA violations in an NFV deployment [11]. The tool should work seamlessly across different VNF implementations, without assuming any support from the VNF itself, and identify all performance issues including those caused due to non-hardware bottlenecks.

This paper presents the design and implementation of NFVPerf (§III), a tool to monitor performance and identify performance bottlenecks in an NFV system. NFVPerf runs as part of a cloud management system like OpenStack [4], and sniffs traffic between NFV components in a manner that is transparent to the VNF. NFVPerf takes as input a configuration file specifying the VNF forwarding graph as well as logic to parse and identify different types of packets between the VNF components. Using this information, NFVPerf computes the per-hop (and end-to-end) application-layer throughputs and delays along the VNF forwarding chain. NFVPerf analyzes these application layer metrics in real-time to estimate the capacity of each component of the VNF, and identify performance bottlenecks. Because NFVPerf uses application layer metrics (and not hardware resource utilizations alone) to identify performance issues, it can identify software related performance bottlenecks along with hardware resource bottlenecks. Further, NFVPerf is a generic tool that can be easily customized to most NFV applications with only configuration changes, and requires no additional metrics or logs from the VNF itself for its functioning. Our validation of NFVPerf (§IV) shows that the tool can analyze large volumes of data with reasonable monitoring overhead, and can accurately identify hardware as well as software performance bottlenecks with high accuracy.

Given the current excitement surrounding NFV, we believe that a tool like NFVPerf greatly eases operator's performance concerns, by enabling easy detection of performance issues across a wide range of VNFs. NFVPerf nicely complements existing monitoring tools that report hardware resource utilizations in cloud management systems, and can form a part of a larger framework to orchestrate and manage VNFs for optimal performance.

II. Related Work

While NFV is a new concept, its basic ideas are similar to those of multi-tier virtualized cloud-based services in data centers. While there is no fundamental difference between an NFV deployment and any other cloud-based application, the additional emphasis accorded to performance by telecom operators calls for a renewed look at cloud-based application performance in the context of NFV. The ETSI NFV ISG (European Telecommunications Standards Institute NFV Industry Specification Group) [12] has recently come up with a taxonomy of service quality metrics to monitor when deploying NFV in telecom networks that have high availability requirements. NFVPerf forms a part of the ecosystem that can be deployed to measure these metrics, by measuring metrics related to application performance.

The most common cause of performance issues in a cloud based application is incoming load exceeding capacity of a hardware bottleneck resource. Cloud service providers today [13], [14], [15] monitor the resource utilizations of the various hardware resources like CPU, memory and network, and identify bottleneck based on these utilizations. These simple techniques fail to detect performance issues when the application has a non-hardware bottleneck such as lock contention among multiple threads (e.g., in the LTE EPC VNF described in [9]), while NFVPerf can detect performance bottlenecks in such cases as well.

A large body of research on autonomous scaling of multi-tier applications uses insights from queuing theory to build a model of the system, and use this model to identify capacity of various components of the application, thereby identifying performance bottlenecks. These models [16], [17], [18], [19], [20] largely assume the network operator has access to application-specific parameters (e.g., request service times), or that these parameters can be easily inferred from hardware resource utilizations (implicitly assuming a hardware resource bottleneck). NFV-Vital [21], Sandpiper [22], CloudScale [23] also monitor the utilization of multiple hardware resources for scaling and resource allocation. Because of the assumption of a hardware bottleneck or knowledge of application-specific metrics, we believe that these ideas cannot directly apply to the NFV scenario, where there are a large number of applications and vendors, and no standardized interfaces to export metrics.

Other research [24] proposes using application-layer throughputs to identify performance bottlenecks. Some previous works also use packet analysis based monitoring for network diagnosis [25] [26] [27]. The NetAlytics [28] packet capture framework is closest to our method, but unlike NFVPerf, the authors do not use the captured packets to identify application performance issues. Further, these methods are dependent on support from the application or the end client to extract these metrics, whereas NFVPerf estimates application layer metrics without explicit support from the VNF.

III. Design

A. Goals and Assumptions

We envisage NFVPerf to be deployed in the following setup: a network operator has a NFVI (NFV Infrastructure), presumably a private (or even public) cloud. The operator obtains VNF implementations for one or more network functions from VNF vendors, installs them on to VMs running on the NFVI, and runs network traffic through them. NFVPerf runs as part of the cloud management service on every physical

machine on the cloud, and alerts the network operator about any performance issues, and localizes the issue to a specific hop in real time. Operators can use this information to take suitable mitigation steps to manage the problem.

We assume that the operator does not have a good idea about the internals of the various VNF components, which is a reasonable assumption if the VNF was not developed in-house. The operator, however, knows the various components that make up the VNF, and the VNF forwarding graph that specifies how traffic must be routed along the chain of VNF components. The operator needs this information to install the VNF components and setup traffic forwarding rules between them. Note that the VNF forwarding graph can be different for different types of requests.

We assume that the VNF has a discrete number of request types that it handles, which are known to the operator. While NFVPerf can be extended to systems that do not comply with this assumption, this description captures the most popular network functions being considered for virtualization in the telecom industry today, such as the LTE Evolved Packet Core (EPC) functionality or the IP Multimedia Subsystem (IMS). We also assume that the operators know the packet formats of requests and responses that flow through the VNF: such information can be easily obtained from standards specifications that the VNF implementation is based upon. NFVPerf uses this information as input to parse network data in real-time to identify application-layer throughputs and delays. Note that VNFs today do not have any standard interfaces to export or query application-layer metrics like throughputs and delays. While the network traffic to the VNF comprises of multiple types of requests (i.e., heterogeneous traffic), we assume that the mix of the traffic (i.e., the ratio of the different types of requests) is fixed or changes infrequently. We believe this is a reasonable assumption for some real life traffic cases, and we will relax this assumption in future work.

The goal of NFVPerf is to measure various application layer metrics like application throughput and delays by sniffing traffic passively, and identify performance issues or bottlenecks. We define a performance bottleneck as one where the incoming load exceeds the capacity of the VNF. When incoming load exceeds capacity, the component can no longer keep up with the load, causing the application throughput to drop below incoming load. This overload also results in a steep increase in processing delay at the component. As a result, NFVPerf uses a drop in application-layer throughputs and a steep rise in application-layer delays to identify performance bottlenecks. Note that a performance bottleneck may or may not correspond to high hardware resource utilizations (e.g., if the application has a software bottleneck, all hardware resource utilizations will be low), but will always correspond to worse application-layer metrics like throughputs and delays (by definition of a "performance issue"). We assume that the load (as measured in requests/sec or bytes/sec) to the VNF is varying with time. NFVPerf's measurement and analysis requires a few seconds to flag an overload scenario; we therefore assume that the incoming load changes not more than once every few seconds, to enable NFVPerf's calculations to converge.

B. Overview

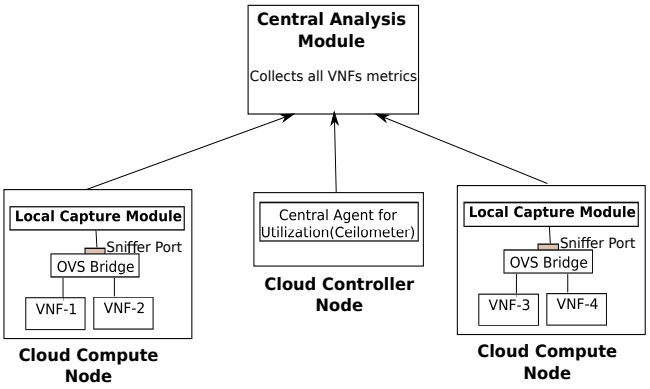

Fig. 1: NFVPerf Architecture.

The overall architecture of NFVPerf is as shown in Figure 1. NFVPerf comprises a local capture module that runs on every physical machine in the NFVI (say, the operator's private cloud), and a central analysis module (that can run anywhere in the operator's network). The local capture module computes application-layer performance metrics and passes them on to the central analysis module. Note that all the VMs in a cloud typically plug into a software switch on every physical machine. For obtaining performance metrics without interfering with the flow path of the packets through the VNF, all ingress and egress packets of all the VNFs from all ports on a physical host are mirrored (copied) on to a single port of the software bridge using port mirroring (§III-C). The local capture module then uses knowledge of packet formats of the VNF to parse the traffic and compute application-layer throughputs and delays on each hop of the VNF forwarding graph (§III-D). In addition to the metrics, the local capture module also infers other properties of the VNF, e.g., if the communication on a given hop is synchronous, that help the central module in identifying bottlenecks. (We define synchronous communication as when the sender waits for an acknowledgement for the previous request before sending the next request.) All of these metrics are periodically conveyed to the central analysis module via `memcached` [29], where the bottleneck detection algorithm (§III-E) is executed.

Once a performance issue is flagged by NFVPerf, an operator can take suitable steps to mitigate the problem. For example, if a hardware resource bottleneck is identified by NFVPerf, the operator can add more resources (e.g., allocate more CPU cores) and scale-out the bottlenecked component on demand. If the issue is due to a software bottleneck in the code, the operator can request for a redesigned VNF. If the cause of the poor application performance is found to be a non-overload related hardware or software fault in the VNF, suitable steps can be taken to patch the bug. NFVPerf currently collects only application-layer metrics to identify application-layer performance issues. As part of future work, we plan

to extend the framework to capture more metrics (e.g., errors from I/O devices) that will enable a fine-grained diagnosis and root cause analysis of the performance issues.

C. Port Mirroring

NFVPerf uses port mirroring on every physical machine to capture all incoming and outgoing packets (including packets between VMs on the same host) in the NFV deployment. The mirrored packets are then analyzed by a script that performs deep packet inspection to compute various metrics of interest. While the setup of port mirrors will vary with the exact deployment, below we describe the setup of port mirrors in our testbed.

Our NFVI consists of a private cloud running the OpenStack [4] cloud management system. Our cloud consists of several physical machines (PMs) that can host several VMs each. OpenStack uses Open vSwitch as the software switch to manage all the VMs on a single PM. The packets going in and out of every VM are captured by mirroring the OVS ports to which the VMs are connected. To understand how to setup port mirrors, we first describe the path taken by packets through the NFVI. The inter-VM communication between two VMs on the same PM and on different PM in OpenStack happens as shown in Figure 2 . The packets flow through the `br-int` bridge of OVS if both the VMs are on the same PM. If the VMs are on different PMs (VM1 and VM3), the communication is through the `br-int` and `br-tun` bridges of OVS and through the VXLAN/GRE tunnel. From the figure, we can see that the

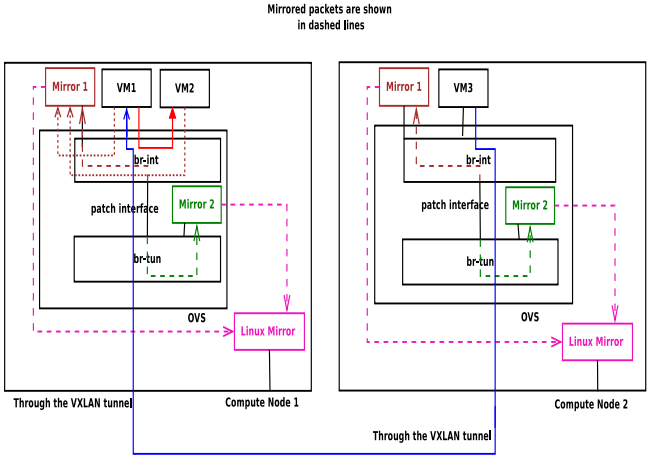

Fig. 2: Inter-VM Communication captured on port mirrors in OpenStack cloud.

ingress and egress packets from a VM can be captured by mirroring:

- the corresponding VM interface on `br-int` (Mirror 1 in Figure 2) and
- the patch interface through which the packets flow to `br-tun` and in turn, through the tunnel to a VM on different PM (Mirror 2 in Figure 2).

Now, notice that packets are mirrored on to ports on `br-int` or `br-tun` based on whether they are flowing between VMs on the same PM or different PM. To collect all packets in to a single port, both the `br-int` and `br-tun` mirror ports are once again mirrored onto a new mirror port on a Linux bridge on the PM (Linux Mirror in Figure 2).

NFVPerf obtains the VNF forwarding graph as input, and the mapping of VMs to PMs and OVS ports from the cloud management system. Using this information, NFVPerf sets up suitable port mirrors to capture packets on all hops of a VNF forwarding chain. Next, NFVPerf analyzes these packets in real time using the `python-pcapy` packet capture and analysis library [30]. We tried several other packets capture libraries (e.g., scapy [31]), but zeroed in on `python-pcapy` due to its low overhead and fast packet capture. In addition to per-hop application layer metrics, NFVPerf also collects hardware resource utilizations (e.g., CPU utilization of each VM) obtained from the Ceilometer service of OpenStack. Our capture script is currently single threaded, because we found that a single threaded application sufficed for the purpose of our experiments. We are currently working on a (relatively straightforward) multi-threaded extension of our capture script.

D. Computing performance metrics

After collecting a copy of all packets between VNF components using port mirrors, NFVPerf proceeds to analyze the packets to compute application-layer metrics. NFVPerf takes as input (say, from a configuration file) the packet formats of various types of requests and responses through the VNF, and the MAC addresses of the various VNF components in the cloud. NFVPerf matches the bytes in every collected packet against the packet formats provided to calculate the count of the various types of requests through the VNFs. These counts over one-second intervals translate to per-second per-hop application layer throughputs for each request type.

In addition to application-layer throughputs, NFVPerf also calculates the time-stamp for when a request entered or exited a VNF component. NFVPerf uses a "key" field in a packet to uniquely identify a packet at both the ingress and egress of a VNF component. This key can be a user identification number (e.g., IMSI) in telecom VNFs, or a TCP 4-tuple and sequence number field for VNFs handling TCP/IP flows. The packet header field that can serve as a key is also provided as input to NFVPerf, along with the packet formats. The timestamps at the ingress and exit are subtracted to obtain the per-hop delay contributed by each component in the VNF chain. The delay samples are binned into discrete bins. The number of bins is set to 10. The per-second average delay of each hop is calculated as the average value of the samples in the bin which has the maximum samples, to avoid bias due to outliers. Note that there is an inherent inaccuracy in timestamps computed in software, unlike timestamps provided by hardware, due to noise added by the various software layers like the device driver, kernel, the software switch, port mirroring, and our own processing scripts. Our binning mechanism is used to reduce the impact of this noise. All of these metrics are transferred to the central node from each local capture node via `memcached` every second. The data structures used to compute the metrics

are periodically flushed at the local capture nodes to keep memory consumption low.

E. Central Analysis Module

The central module receives metrics corresponding to all VNF components in the VNF chain every second, as described above. The central node averages these metrics over epochs of 5 second duration, and runs a performance bottleneck detection algorithm every epoch. The algorithms runs through all the components of the VNF chain and examines the average throughput, delay, and incoming load of that hop. The averages may be computed over one or more epochs (we compute averages over the past 10 seconds, i.e., the past two epochs), even though the algorithm runs every epoch. The averages are also weighted across all request types. For every component, the algorithm checks if the component has seen an *application performance degradation*, defined as the ratio of the average throughput to the average load falling below a certain threshold (set to 0.97 in our implementation), and the ratio of the average delay to minimum delay exceeding a threshold (set to 8), indicating that the component is unable to keep up with the incoming load. If such a degradation is noticed, the algorithm first identifies if there is a synchronous communication with a downstream node; if it exists, the downstream itself could be the bottleneck. Once the bottleneck is identified (as the current node or its downstream), the algorithm notes the incoming load level at which the degradation has occurred. If this incoming load is lower than the previously learned capacity, the algorithm updates capacity to the current load level. That is, as the algorithm runs over several epochs, it builds up a profile of throughputs and delays over several incoming load levels, and eventually converges to the capacity, defined as the highest load level that does not cause a performance degradation. For every epoch where the incoming load at a component exceeds its current known capacity, a performance bottleneck alarm is raised to the operator.

The intuition behind the algorithm is fairly straightforward: any component that has a hardware or software bottleneck will eventually fail to keep up with the incoming load, causing its throughput to fall below the incoming load. Further, the queuing delay at the component increases, causing the average delay to increase steeply over the minimum delay (that corresponds to a low load scenario). As a result, the lowering throughput and increasing delay will eventually flag the component as a bottleneck, once the system has run for a few epochs and learnt the delay and throughput profile at a few load levels. Note that this algorithm takes a few epochs to calibrate itself, e.g., to learn the minimum delay. Further, the load levels during this calibration period should vary over a good range of values, to enable the algorithm to learn good delay and throughput profile over several values of load. However, once the algorithm has run for a few epochs, it should be able to detect performance issues (i.e., falling throughputs and rising delays) without any additional metrics from the VNF itself. The delay and throughput profiles are periodically flushed to learn changes in capacity.

Note that the average delays and throughputs across various load levels are comparable as long as the traffic mix (i.e., the ratio of different types of requests) stays roughly the same. If the mix of the requests varies rapidly, one could end up with widely different average delays for the same load level, depending on the relative proportion of harder and easier requests. Extending our algorithm to work with a varying mix of requests is part of future work.

IV. Setup and Evaluation

We begin by describing our testbed setup, and the validation performed to quantify the performance overheads of our tool. We then evaluate the accuracy of NFVPerf in identifying performance bottlenecks.

A. VNF setup

We used the OpenIMSCore NFV prototype [32] as a candidate VNF in our experiments. OpenIMSCore is an open-source NFV-based implementation of IMS (IP Multimedia Subsystem [33]). IMS is a control-plane framework for setting up multimedia communication (e.g., voice and video calls) over IP networks. While it is not widely deployed in telecom networks today, NFV-based implementations of IMS are receiving significant interest in the telecom operator community, and are one of the most popular VNFs being tested out. The IMS Core network function has several components: Proxy Call Session Control Function (P-CSCF), Interrogating CSCF (I-CSCF), Serving CSCF (S-CSCF), and a Home Subscriber Server(HSS). Each of these components is implemented as a separate software appliance in the OpenIMSCore prototype. The IMS Core subsystem mainly handles two types of requests: registration requests from new users that wish to use the system, and call setup requests from registered IMS users to other IMS or non-IMS users. Each request is serviced by various components of the IMS core subsystem before a response is returned to the user.

We used a custom load generator based on the open-source software UCTIMSClient [34] for generating load to the NFV prototype. Our load generator also prints out client side throughputs and delays to serve as ground truth for validating our tool. We also optimized our load generator and verified that it did not become the bottleneck in our experiments.

B. Testbed Setup

We used our private cloud running the OpenStack cloud management system [4] for all our experiments. OpenStack comprises of a controller and several compute nodes. The compute nodes run a hypervisor (kvm) and a software switch (OpenvSwitch) to manage the VMs. Our cloud used in the experiments below consisted of 4 physical servers, each with 8 CPU cores (Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz) and 32GB RAM, all connected by a 1Gbps LAN. For our tests, the 4 IMS Core VNF components (P-CSCF, I-CSCF, S-CSCF and HSS) were each installed on a VM and hosted on the cloud. The P-CSCF, I-CSCF, S-CSCF and HSS VMs were provisioned with the following number of cores and

RAM respectively: (1 core, 4GB), (2 cores, 4GB), (2 cores, 4GB), and (4 cores, 10GB). The load generator was run on a VM with 4 cores and 10 GB RAM. The local capture module was running on all physical machines, and the central analysis module was running on one of the compute nodes. The configuration file given as input to NFVPerf consisted of the MAC addresses of the VNF components, and VNF forwarding chain and packet formats for both request types.

C. Tool Validation

We now verify that NFVPerf can capture packets and parse them in real time without significant overhead on the system. For this purpose, we use iperf [35] to send and receive UDP packets between VMs at the maximum possible rate, and use NFVPerf to capture and parse the packets. The capture script running on the mirrored ports was a single threaded python script. The capture script ran in two modes: with deep packet inspection (DPI) to emulate parsing packets with a real VNF, and without DPI to quantify the base case overhead. The iperf client and server both ran on a single core VM, on the same PM as well as on different PMs. Our experiments showed that a single core capture module of NFVPerf is able to keep up with real-time capture and parsing for up to 800 Mbps of incoming traffic before it hits a CPU bottleneck. We believe than an optimized multi-threaded version of NFVPerf can provide even better performance; this is part of our future work. Next, we compare the delays computed by NFVPerf with those obtained from analyzing the pcap files offline using tcpdump. We found that the delays captured by NFVPerf differed from those computed by offline analysis by 7% on average, due to the overheads of real-time calculations in our script. Therefore, we conclude that online measurement of NFVPerf does not add much error as compared to offline packet capture. Note, however, that the delays in the offline analysis of pcaps could themselves be off from the true value of delays, due to the overhead of software-based timestamps. To validate the delays from NFVPerf against "ground truth", one would need access to accurate hardware timestamps.

D. Accuracy of NFVPerf

We now evaluate the accuracy of NFVPerf in detecting performance bottlenecks. We setup the the OpenIMSCore VNF components on our cloud testbed, and load the components with traffic from our load generator. All the requests in OpenIMSCore are CPU intensive. First, we performed an offline benchmarking of the VNF to identify its capacity, i.e., the incoming request rate it can process without dropping requests. *Note that this exercise was only to know the ground truth, and NFVPerf did not have access to the offline experimental results.* Next, we ran experiments where the incoming load (req/s) to the VNF was varied every 20 seconds over the course of the experiment, and we observed if NFVPerf was able to flag a performance bottleneck when the load exceeded the ground truth capacity. We also used hardware resource utilizations to detect bottlenecks, in order to compare with NFVPerf: when the CPU utilization of any component crosses a threshold of 80% (given that we know our workload is CPU-bound), we flag it as an overload, as such heuristics are commonly used in current commercial cloud-based systems [8].

We present results from an experiment where varying rates of registration requests are sent to the OpenIMSCore VNF components; results with other traffic scenarios were similar. Figure 3 shows the incoming load as a function of time, the ground truth capacity obtained from offline benchmarking, and the times when NFVPerf detects a bottleneck. We see from the figure that NFVPerf accurately identifies all epochs where the load exceeds the ground truth capacity. NFVPerf's algorithm also correctly localized the fault to the correct VNF component (HSS in this case). We also show the epochs where the utilization threshold based technique detects bottlenecks, and we find that its performance is comparable to NFVPerf.

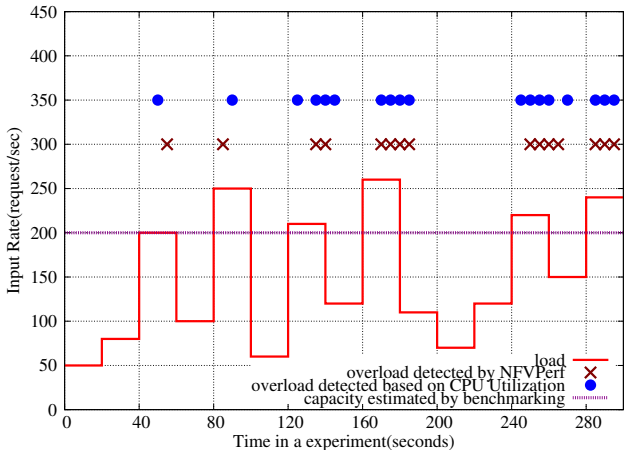

Fig. 3: Performance bottleneck detection in NFVPerf.

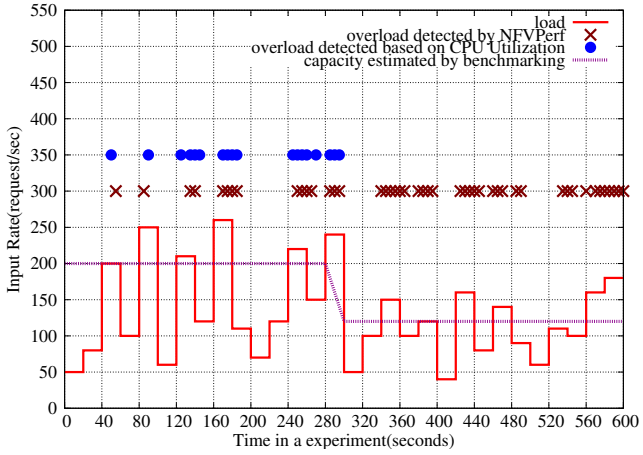

Fig. 4: Detection of software performance bottlenecks.

Next, we introduce a software bottleneck in the code of HSS midway in our experiment, causing the HSS performance to drop even before it hits high CPU utilization. This models real-life scenarios where VNF implementations have non-hardware bottlenecks. We then repeat our experiments and compare the

accuracy of NFVPerf and the method based on CPU utilization threshold, as shown in Figure 4. We find that, while NFVPerf can still accurately detect performance degradation due to a non-hardware bottleneck, the CPU utilization based method fails to identify performance bottlenecks in all overloaded epochs. This experiment highlights the benefits of a sophisticated tool like NFVPerf over simpler tools based on measuring hardware resource utilizations.

V. Conclusion

As the concept of NFV is gaining attention from industry, providing performance guarantees has become a critical need for NFV infrastructure providers. The current techniques to identify performance bottlenecks do not suit all VNF implementations, e.g., if the VNF has a non-hardware bottleneck, or if the VNF implementation does not expose performance metrics. This paper describes NFVPerf, a powerful tool to detect performance bottlenecks in real-time in a NFV system. NFVPerf works by passively monitoring traffic through the VNF forwarding graph, calculating application layer throughputs and delays, and identifying performance bottlenecks based on a degradation of these metrics. The design of NFVPerf is generic enough to work across a variety of VNFs. Our evaluation of NFVPerf on a real VNF prototype running on our private cloud shows that NFVPerf can detect performance bottlenecks with high accuracy. Our experience showed us that NFVPerf is limited by the accuracy of software timestamps available in kernel code, and would greatly benefit from accurate hardware supported timestamps, which will lead to more accurate estimates of delays over shorter epochs. We will explore such avenues as part of future work.

Acknowledgments

This work was supported by a grant from Tech Mahindra Ltd., and a seed grant fellowship from IIT Bombay.

References

[1] ETSI, "Network Functions Virtualisation," https://portal.etsi.org/nfv/nfv_white_paper.pdf, 2012, [Online; accessed 23-October-2014].

[2] C. DeCusatis and P. Mueller, "Virtual firewall performance as a waypoint on a software defined overlay network," in *Proc. of the IEEE Intl Conf on High Performance Computing and Communications*, 2014.

[3] "Connectem," https://www.proceranetworks.com/collateral/casestudy-connectem , 2015, [Online; accessed 15-December-2015].

[4] "OpenStack," https://www.openstack.org/, 2014.

[5] "OPNFV," https://www.opnfv.org/, 2014.

[6] "Intel Data Plane Development Kit for Linux*: Guide," http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/int el-dpdk-getting-started-guide.html , 2014.

[7] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proc. of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[8] "Industry Cloud Scaling," http://www.affirmednetworks.com/products-solutions/virtualization/ , 2015, [Online; accessed 11-August-2015].

[9] A. Rajan, S. Gobriel, C. Maciocco, K. Ramia, S. Kapury, A. Singhy, J. Ermanz, V. Gopalakrishnanz, and R. Janaz, "Understanding the bottlenecks in virtualizing cellular core network functions," in *Proc. of the IEEE International Workshop on Local and Metropolitan Area Networks (LANMAN)*, 2015.

[10] "Managing and monitoring performance in sdn/nfv," https://www.virtualizationpractice.com/managing-monitoring-performance-sdn-nfv-3 2088/ , [Online; accessed 15-December-2015].

[11] "Importance-performance-monitoring-sdn-nfv," https://www.sevone.com/network-project/importance-performance-monitoring-sdn-nfv , [Online; accessed 15-December-2015].

[12] ETSI, "Network Functions Virtualisation (NFV); Service Quality Metrics ," http://www.etsi.org/deliver/etsi_gs/NFV-INF/001_099/010/01.01.01_60/gs_nfv-i n f 0 1 0v010101p.pdf, 2014, [Online; accessed 16-September-2015].

[13] "Amazon auto scaling service." http://aws.amazon.com/autoscaling/, 2015, [Online; accessed 23-October-2015].

[14] D. Bellenger, J. Bertram, A. Budina, A. Koschel, B. Pfänder, C. Serowy, I. Astrova, S. G. Grivas, and M. Schaaf, "Scaling in cloud environments," in *Proc. of the 15th WSEAS International Conference on Computers*, 2011.

[15] R. Han, L. Guo, M. Ghanem, and Y. Guo, "Lightweight resource scaling for cloud applications," in *Proc. of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2012.

[16] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal, "Dynamic provisioning of multi-tier internet applications," in *Proc. of the Autonomic Computing, ICAC 2005*, 2005.

[17] C. Stewart, T. Kelly, and A. Zhang, "Exploiting nonstationarity for performance prediction," in *Proc. of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, EuroSys '07*, 2007.

[18] Q. Zhang, L. Cherkasova, and E. Smirni, "A regression-based analytic model for dynamic resource provisioning of multi-tier applications," in *Proc. of the Autonomic Computing, ICAC '07*, 2007.

[19] A. Ali-Eldin, M. Kihl, J. Tordsson, and E. Elmroth, "Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control," in *Proc. of the 3rd Workshop on Scientific Cloud Computing Date, ScienceCloud '12*, 2012.

[20] R. Singh, U. Sharma, E. Cecchet, and P. Shenoy, "Autonomic mix-aware provisioning for non-stationary data center workloads," in *Proc. of the 7th International Conference on Autonomic Computing, ICAC '10*, 2010.

[21] L. Cao, P. Sharma, S. Fahmy, and V. Saxena, "Nfv-vital: A framework for characterizing the performance of virtual network functions," in *Proc. of the IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, 2015.

[22] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Sandpiper: Black-box and gray-box resource management for virtual machines," *Comput. Netw.*, vol. 53, no. 17, 2009.

[23] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: Elastic resource scaling for multi-tenant cloud systems," in *Proc. of the 2nd ACM Symposium on Cloud Computing, SOCC '11*, 2011.

[24] A. Gember, R. Grandl, A. Anand, T. Benson, and A. Akella, "Stratos: Virtual middleboxes as first-class entities," *UW-Madison TR1771*, 2012.

[25] V. Mann, A. Vishnoi, and S. Bidkar, "Living on the edge: Monitoring network flows at the edge in cloud data centers," in *Proc. of the 5th International Conference on Communication Systems and Networks (COMSNETS)*, 2013.

[26] Y. Zhuang, E. Gessiou, S. Portzer, F. Fund, M. Muhammad, I. Beschastnikh, and J. Cappos, "Netcheck: Network diagnoses from blackbox traces," in *Proc. of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[27] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *Proc. of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03*, 2003.

[28] G. Liu and T. Wood, "Cloud-scale application performance monitoring with sdn and nfv," in *Proc. of the IEEE International Conference on Cloud Engineering (IC2E)*, 2015.

[29] J. Petrovic, "Using memcached for data distribution in industrial environment," in *Proc. of the 3rd International Conference on Systems, ICONS 08*, 2008.

[30] "Pcapy," http://www.coresecurity.com/corelabs-research/open-source-tools/pcapy , 2015.

[31] "Scapy," http://www.secdev.org/projects/scapy/ .

[32] "Openimscore," http://www.openimscore.org/.

[33] "Ip multimedia subsystem," http://en.wikipedia.org/wiki/IP_Multimedia_Subsystem, 2015.

[34] D. Waiting, R. Good, R. Spiers, and N. Ventur, "The UCT IMS Client," in *Proc. of the 5th IEEE International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities and Workshops, TridentCom '09*, 2009.

[35] "Iperf," https://iperf.fr/, 2014.