NFV-VITAL: A Framework for Characterizing the Performance of Virtual Network Functions

Lianjie Cao*, Puneet Sharma*, Sonia Fahmy†, Vinay Saxena‡
*Hewlett Packard Labs †Purdue University ‡Hewlett Packard Enterprise
e-mail: {lianjie.cao, puneet.sharma, vinay.saxena}@hpe.com, fahmy@purdue.edu

***Abstract*—Network Function Virtualization (NFV) brings a cloud service automation paradigm to demand-driven elastic flexing of infrastructure resources. Thus, it is essential to characterize the impact of hardware and virtualization options on the virtual network function (VNF) performance, and on the load on underlying infrastructure.**

In this paper, we present VNF characterization case studies with three sample open-source VNF platforms, the *Clearwater* IMS VNF and two intrusion detection system VNFs (*Snort* and *Suricata*). We demonstrate that VNF characterization is vital for optimizing VNF performance, as well as efficient utilization of infrastructure resources. We use the lessons learned from our case studies to design and implement a VNF characterization framework, NFV-VITAL, to characterize VNFs based on user preferences and available resources. We demonstrate how NFV-VITAL can automatically determine optimal configurations under different workloads with the three sample VNFs.

I. Introduction

Driven by the requirements for faster provisioning of network services, Communication Service Providers (CSPs) have embarked on a major transformation of their network infrastructure by adopting Network Function Virtualization (NFV) [1]. NFV entails implementing network functions – currently available on proprietary middleboxes and network equipment hardware – in software. Such Virtual Network Functions (VNFs) can be deployed on industry standard commodity servers, storage and switches. NFV allows CSPs to leverage virtualization and cloud automation technologies. As with the "Cloudification of IT services," the NFV transformation not only enables agile network service deployment, but also improves demand-driven elastic flexing for scale-out, and breaks the hardware vendor lock-in as VNFs are portable across different hardware platforms.

Several challenges need to be tackled for successful NFV deployment. First, the network equipment providers should keep the performance degradation from software implementation of network functions to a minimum. Second, the NFV orchestration tools need to determine the virtualization setups and configuration options to optimize VNF performance and automatically scale the VNF resource allocation with workload. Third, a unified interface for decoupling virtualized instances from underlying hardware is needed. Although understanding the impact of virtualization on network functions is of paramount importance for NFV to succeed, current studies are driven by vendor efforts to demonstrate either the overhead associated with their respective technologies, or the performance of their respective VNF implementations. Performance studies of NFV fall into three broad categories: (i) Performance benchmarking of a single option, such as CPU pinning, Intel DPDK, PF_RING, and SR-IOV, *e.g.*, [2], [3], (ii) Performance testing of a single VNF running on a hypervisor in an isolated environment, *e.g.*, [4], and (iii) Network performance measurement, such as UDP/TCP throughput and delay, of a VM or a public cloud deployment, *e.g.*, [5].

One difficulty in creating an NFV characterization framework is the large number of configuration knobs and hardware settings (*e.g.*, CPU pinning, c-states, and memory interleaving) available in NFV deployments. Additionally, the compute and network requirements of various VNFs vary significantly. While VNFs such as virtual routers and firewalls are primarily bounded by network throughput, others such as load balancers are bounded by network and compute (or memory) for session state management. Some VNFs may comprise multiple simpler VNFs/components with a communication and dependency relationship among them. Each of these components can exhibit different virtualization impacts and scalability requirements.

This paper makes two key contributions. First, we conduct a VNF case study on the Clearwater [6] cloud-based IP Multimedia Subsystem (IMS) (section II). Motivated by this case study, we propose *NFV-VITAL* (Virtualization Impact on Throughput And Load) – a framework for performance characterization of different types of VNFs in a real private cloud deployment (OpenStack) with different options (*e.g.*, CPU pinning) (section III). We demonstrate the benefits of the NFV-VITAL framework for analyzing optimal sizing and configuration for the Clearwater, Snort, and Suricata VNFs. NFV-VITAL can be used for automatically: (1) Estimating VNF capacity for a given resource configuration, (2) Computing virtualization and system overhead associated with resource flexing such as scale-out and scale-up, (3) Determining the optimal resource configuration for a given workload, (4) Evaluating different virtualization and hardware options, and (5) Fine-tuning VNF implementation and performance.

II. Case Study

Before designing NFV-VITAL, we conduct a case study to understand how different orchestration/scaling methods affect the performance of a clustered VNF with different components. We seek answers to questions such as (1) how VNF performance varies when allocating the same resources in different setups; (2) causes for the differences: load balancing, inter-component synchronization or intra-component communication; (3) how to alleviate performance degradation with different orchestration methods; (4) how to detect when a VNF is approaching a performance bottleneck; and (5) how best to scale the system to achieve the highest performance gain. With

these answers, we can derive the characterization framework in section III.

We construct a testbed with 3 HP DL360p blade servers and 2 HP Z420 workstations, connected by an HP ProCurve 3500yl Gigabit switch. Table I gives the basic configuration of the testbed machines. We use the OpenStack [7] Icehouse release as a cloud orchestrator to manage the compute and network resources. The 3 DL360p blade servers are used as compute nodes (CNs) and the two Z420 workstations are used as controller and network nodes (NN), respectively. The OpenStack networking component is configured with Modular Layer 2 (ML2) GRE tunnels. All tests in this paper are conducted on this testbed.

TABLE I: Blade server, workstation and VM configurations

PM/VM	CPU	Cores	RAM
DL360p	2x Intel Xeon E5-2680 v2	20	212 GB
Z420	1x Intel Xeon E5-1620	4	16 GB
cw1.small	1x vCPU	1	2 GB
cw1.medium	2x vCPU	2	4 GB
cw1.large	4x vCPU	4	8 GB

A. Clearwater: An IP Multimedia Subsystem VNF

Clearwater [6] is a real world, telco-grade IP Multimedia Subsystem (IMS). [1] It is a typical clustered VNF that comprises a number of components, each of which plays a unique role in the system and exhibits unique resource utilization patterns: ***Bono*** is the SIP edge proxy component, providing both a SIP IMS Gm compliant interface and a WebRTC interface to clients. ***Sprout*** serves as combined SIP registrar and authoritative routing proxy, and handles client authentication and interfaces to other application servers. ***Homestead*** is the home subscriber server with web interfaces provided to Sprout for retrieving authentication credentials and user profile information. ***Homer*** is a standard XML document management server that stores multimedia telephony service settings for each user. ***Ralf*** is used for offline billing. ***Ellis*** is a sample web-based user provisioning portal for self sign-up, password management, line management and control of service settings.

Our lab deployment of Clearwater excludes Ralf and Ellis, since these two components are for billing and account management which are not key for our characterization case study. Each of the deployed components consists of one or more VMs of different sizes orchestrated by OpenStack. A separate VM is used as a DNS server for internal communication and load balancing in clustered deployments of Clearwater.

Sprout and Homestead use *memcached* [8] and *cassandra* [9] to store registration state and user information, respectively. Therefore, clustering these two components involves synchronization of the datastore and timer service. As the SIP edge proxy, each Bono node works independently without any synchronization with other Bono nodes. All Clearwater VMs are assigned to the same compute nodes of OpenStack and each vCPU is pinned to a physical core to eliminate potential effects of network and CPU dynamics.

B. Testing Methodology

We vary the workload and the deployment size of Clearwater (scaling up/down, in/out) to understand the impact of orchestration strategies. We collect the CPU, memory and network usage of all instantiations for deeper analysis.

1) Workload Generation: We use the open source tool SIPp [10] as the workload generator, and choose user registration and deregistration as a test scenario. Each SIPp *reg-dereg* call contains three REGISTER requests – the first two are for registration and authentication and the third is for deregistration. If any of the three requests yields an unexpected response (*e.g.*, 408 Request Timeout and 503 Service Unavailable), the call is considered as "failed." If a request does not receive any response in 10 seconds, this call also fails due to timeout.

SIPp runs on a dedicated physical server. We change the call rate with a granularity of 50 calls/sec from 200 calls/sec to 1100 calls/sec depending on the deployment size. Each test lasts 300 seconds and we repeat it 10 times to average the results.

2) System Capacity Measurement: The definition of VNF capacity may vary depending on the functionality of the VNF and type of workload. In our study of the Clearwater IMS, the most important metric is the maximum offered workload at which it can provide the desired stable service. We measure this by monitoring how closely the successful call rate (*SCR*) reported by SIPp follows the input call rate (*ICR*) we specify in SIPp. Both quantities are in units of calls per second.

Clearwater uses a token bucket mechanism [11] to control the load. It accepts a request only if the token bucket is non-empty. The token bucket is replenished based on a token rate that is adjusted as follows. If current message queuing delay has not reached a predefined threshold value, the token rate increases additively; otherwise it decreases multiplicatively:

$$ICR \uparrow \Rightarrow message\ queuing\ time \uparrow \Rightarrow token\ rate \downarrow$$
$$\Rightarrow reject\ requests\ \Rightarrow SCR \downarrow$$

Therefore, we define Clearwater system capacity with *reg-dereg* traffic to be the maximum ICR when $\frac{SCR}{ICR} \geq$ a specified threshold, *e.g.*, 0.8 or 0.9. In other words, when *reg-dereg* call rate exceeds the system capacity, a certain percentage, say 10 or 20%, of the calls will fail.

C. Results

Based on our observations, Bono, Sprout and Homestead are the three most heavily loaded Clearwater components and they are all CPU-bound in single VM per component instantiation. In our tests, we change the total number of vCPUs assigned to the Clearwater cluster and evenly distribute them among the three components. Homer uses one small instance in all tests. Therefore, in the rest of the analysis, we focus on CPU usage analysis of Clearwater instances. We did not cluster Bono nodes due to the observation that Bono nodes work independently without any synchronization overhead and it is not the bottleneck in any of our tests.

Table II shows the deployment sizes we tested and the maximum ICR when $\frac{SCR}{ICR}$ is 0.8 and 0.9. The column $\frac{SCR}{ICR} = 0.9$ gives the system capacities. In the remainder of this section, we discuss the results using different scaling methods.

[1] Clearwater is still under active development (released twice a month). Different versions may exhibit different performance.

TABLE II: Clearwater system capacity using different scaling methods. `bn1s-sp1s-hs1s` stands for one small instance for Bono, 1 small instance for Sprout and 1 small instance for Homestead

Clearwater Sizing	vCPUs	ICR (0.9)	ICR (0.8)
`bn1s-sp1s-hs1s`	3	350	350
`bn1m-sp1m-hs1m`	6	600	650
`bn1m-sp2s-hs2s`	6	300	350
`bn1l-sp1l-hs1l`	12	1000	1100
`bn1l-sp1l-hs2m`	12	550	650
`bn1l-sp2m-hs1l`	12	650	650
`bn1l-sp2m-hs2m`	12	450	500
`bn1l-sp1l-hs4s`	12	400	450
`bn1l-sp4s-hs1l`	12	500	500
`bn1l-sp4s-hs4s`	12	350	450

1) Scaling Up: To test Clearwater scaling up, we use one instance per component and increase the size of instances from small to large (*e.g.*, `bn1s-sp1s-hs1s`, `bn1m-sp1m-hs1m` and `bn1l-sp1l-hs1l` in table II). Fig. 1 shows the SCR changes as ICR increases.

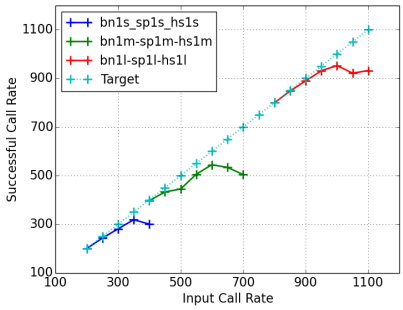
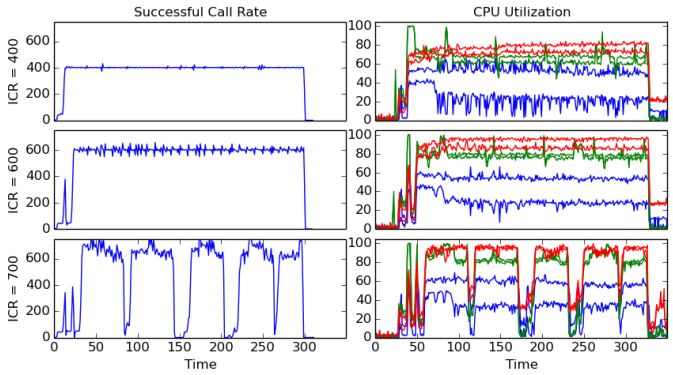


Fig. 1: Clearwater: scaling up

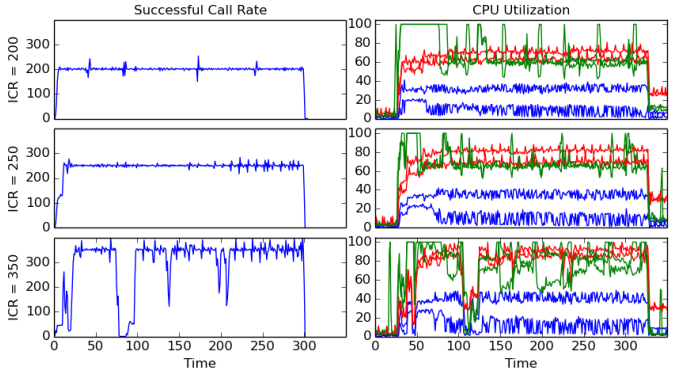
For the same deployment size, SCR changes significantly when ICR exceeds the system capacity point of $SCR = 0.9$, which means that Clearwater can no longer provide stable service beyond that point. Scaling up Clearwater by doubling the CPU resource almost doubles the system capacity. For instance, when Clearwater scales from `bn1s-sp1s-hs1s` to `bn1m-sp1m-hs1m`, the system capacity increases from 350 calls/sec to 600 calls/sec, and scaling further to `bn1l-sp1l-hs1l` boosts system capacity to 1000 calls/sec.

Fig. 2a shows SCR and CPU usage in test case `bn1m-sp1m-hs1m` collected from the SIPp report and utilization traces at three different workload levels: underload, system capacity and overload. Additional plots are available in [12]. We found that both SCR and CPU utilization oscillate significantly when the workload exceeds the system capacity point. This is due to the Clearwater load control mechanism on the Sprout instance, which rejects requests when it detects that the request queuing time exceeds a given latency (*e.g.*, 100 ms). CPU resources are freed during this load adjusting period and SIP requests are not handled. This situation can occur on all instances of Sprout and Homestead when the CPU usage approaches 100%. In other words, this instance may become the bottleneck of the entire Clearwater system.

2) Scaling Out: We investigate scaling out by increasing the number of instances for different components. We compared Clearwater deployment sizes from 1 small instance per



(a) `bn1m-sp1m-hs1m`



(b) `bn1m-sp2s-hs2s`

Fig. 2: SIPp statistics and CPU utilization for `bn1m-sp1m-hs1m` and `bn1m-sp2s-hs2s`. Blue, red and green curves stand for CPU usage of Bono instance, Sprout instances and Homestead instances respectively.

component to 4 small instances per component or 2 medium instances per component (*e.g.*, `bn1s-sp1s-hs1s`, `bn1m-sp2s-hs2s`, `bn1l-sp2m-hs2m` and `bn1l-sp4s-hs4s`) shown in fig. 3. Scaling out from `bn1s-sp1s-hs1s` to `bn1m-sp2s-hs2s` does not improve the performance of Clearwater even when the total CPU resource assigned to Clearwater is doubled. Scaling out further to `bn1l-sp2m-hs2m` and `bn1l-sp4s-hs4s` improves the performance, but it is still significantly less than the base case `bn1l-sp1l-hs1l` with equivalent vCPU resources.

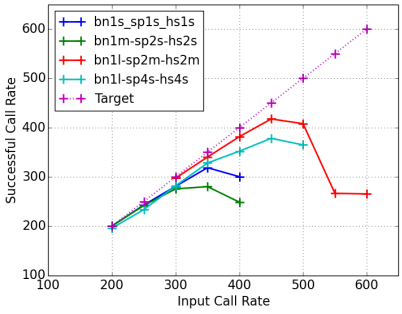


Fig. 3: Clearwater: scaling out

Compared to scaling up, scaling out does not increase Clearwater system capacity due to two key reasons: (1) Data synchronization among nodes of the same component: In

Clearwater, this includes synchronization of *chronos* and *memcached* on Sprout nodes and cassandra on Homestead nodes. *chronos* is a synchronized timer service to track registration expiry information on different Sprout nodes; and (2) Imperfect load balancing: Bono creates 50 connections to all Sprout instances, and these connections get refreshed every minute following a Poisson distribution. If too many rejections (503 response) occur on one specific Sprout node, this Sprout node is blacklisted by Bono for several periods. Therefore, this Sprout node becomes idle. Hence we conclude that, Clearwater does not always benefit from more CPU resources, which results in the imbalance of CPU usage we observed on different Sprout nodes in Fig. 2b. This does not occur when allocating all CPU resources to one instance in the scaling up tests. The blacklisting combined with per instance load control make load balancing ineffective.

3) Hybrid Scaling: The goal of hybrid scaling is to isolate clustering impact from different components using the same amount of CPU resources. For instance, Sprout clusters *memcached*, while Homestead clusters *cassandra*. We compared `bn1l-sp1l-hs2m` to `bn1l-sp2m-hs1l`, and `bn1l-sp4s-hs1l` to `bn1l-sp4s-hs4s`, all of which use 12 vCPUs. As shown in fig. 4, Clearwater exhibits worse performance when clustering more and smaller instances, compared to fewer and larger instances. Clustering different components changes the performance patterns as workload increases. For instance, SCR shows flatter reduction when clustering Homestead, while it drops suddenly after reaching the system capacity point when clustering Sprout.

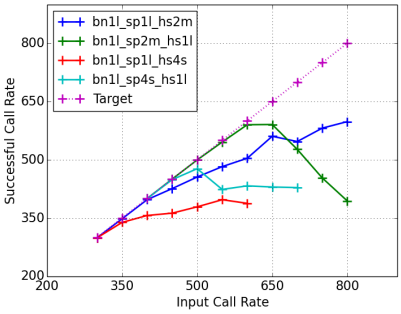


Fig. 4: Clearwater: hybrid scaling

III. Framework Design

Motivated by our case study in section II, we believe that an NFV performance characterization framework should: (1) accommodate different types of VNFs, (2) adapt the deployment size of a VNF, with awareness of VNF components, (3) generate different VNF workloads, (4) collect resource utilization traces of VNF instances, and (5) generate VNF performance evaluation reports.

The aim of NFV-VITAL is to quickly determine the configuration yielding maximum performance of a VNF. This poses the following challenges: (1) how to handle different types of VNFs, and (2) how to make practical and thorough performance testing plans for a given VNF. To generalize to different VNF types, NFV-VITAL allows users to plug in their own scripts to deploy and configure a target VNF, run workload generators, and specify high-level testing "hints" for different testing modes. Metrics vary based on the VNF. Researchers may evaluate intrusion detection system (IDS) software in terms of accuracy, throughput, or additional latency. NFV-VITAL computes the relationship between the offered load and the system throughput when varying virtualization/platform options and orchestration strategies. In other words, NFV-VITAL computes the maximum workload a VNF can handle before service quality degrades, using different deployment sizes and virtualization options.

We leverage the proposed NFV architectural framework from ETSI [1] and implement NFV-VITAL as shown in Fig. 5.

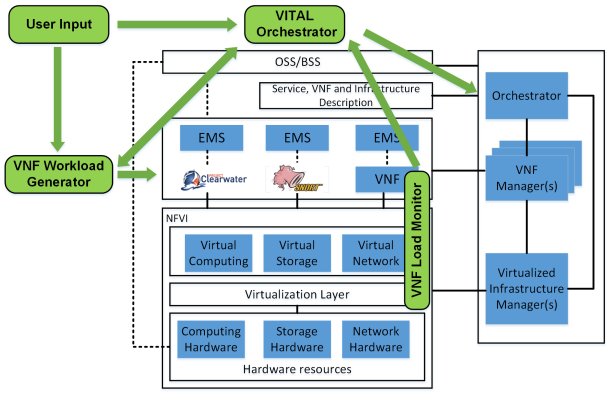


Fig. 5: NFV-VITAL framework architecture

The framework consists of four components: VITAL orchestrator, VNF workload generator, VNF load monitor, and user input.

User Input: As discussed above, VNFs are designed for different purposes, making it difficult to unify their usage. For testing purposes, we classify their differences into three types: (1) installation/deployment, (2) workload generation, and (3) evaluation metric. To bridge these differences, NFV-VITAL allows users to provide their own deployment specification and workload specification files, both of which are in `json` format. Deployment specifications include VNF information such as `vnf_name`, `components`, `instances`, `flavors`, `servers`, `install`, and `mode`. `flavors` are the VM sizes to test and `server` includes user preferences when choosing hosts for VNF instances, such as the preferred number of servers and virtualization features. `mode` gives the testing mode for the target VNF, which we will discuss in section III-A. The workload specification instructs the VNF workload generator on different `rates`, `stop`, `generator` and `repeat` values. Examples and detailed explanations of user input can be found in [12].

VITAL Orchestrator: The VITAL orchestrator generates Heat templates [13] that represent the deployment sizes that the framework will test. The VITAL orchestrator uses each Heat template to start all instances specified and then runs the given installation script to bootstrap the VNF. After deployment is complete, it invokes the VNF workload generator to initiate the testing process.

The orchestrator also executes a daemon process to receive and manage resource utilization traces from the VNF load monitor, and VNF performance logs from the VNF workload generator. After all tests complete, system performance and resource utilization plots are generated for each test. Comparison

plots can be produced for all deployment sizes tested. Based on the testing mode, the VITAL orchestrator may scan resource utilization traces immediately after testing each deployment size.

VNF Workload Generator: The workload generator is invoked by the VITAL orchestrator once a test deployment is complete. The generator reads a specification file, which includes the name of an input script to run in order to generate the workload(s) on the target VNF. The users specify the location of this script in the `generator` field. NFV-VITAL relies on the users to specify the range in which the workload generator operates and the type of traffic to use, *e.g.*, SIPp registration requests. The generator starts with the minimum workload rate given in a `range` field, and linearly increases the rate based on an `increase` field. The same test may be repeated several times depending on the value of a `repeat` field. The VNF workload generator also expects the user script to return a performance indicator (*e.g.*, SCR in Clearwater tests) to describe system performance.

Since we use the same `range` to generate workloads for all deployment sizes, it is possible that some target VNF deployments with small sizes reach the system capacity point ahead of reaching the maximum value in `range`. In this case, increasing workload further is not useful. Thus, we provide an optional feature for the user to define a stopping condition in the `stop` field. The testing process for a given deployment size terminates when the maximum rate is reached or the stopping condition is satisfied.

VNF Load Monitor: Since users may not have access to physical hosts on some platforms, the VNF load monitor runs on all VNF instances to collect (i) CPU, (ii) memory, and (iii) network utilization, and record them in a csv file during a test. When a test is complete, the three utilization traces are uploaded to the VITAL orchestrator. The users can decide to use any of three utilization values for scaling.

A. Testing Modes

To make practical and thorough testing plans, NFV-VITAL allows users to choose from three testing modes: custom sizing, exhaustive search, and component-aware directed search.

Custom Sizing: In this mode, users can specify different deployment sizes in the `sizes` field of the deployment specification, *e.g.*., number and flavor of each VNF component. The VITAL orchestrator directly translates these deployment sizes into Heat templates. A user can use this mode to test the maximum workload and the bottleneck components of specific deployment sizes.

Exhaustive Search: Exhaustive search is ideal when a user wants to test all possible deployment sizes for given resources. With exhaustive search, the VITAL orchestrator first computes all possible combinations based on the given flavors of each VNF component that satisfy the resource requirements, then translates all possible sizes into Heat templates. As with custom sizing, deployment sizes to test are determined beforehand in an offline fashion.

Component-aware Directed Search: An important lesson we learned from section II is that the performance of a VNF deployment can be limited by a specific component. Thus, we design a component-aware directed search system to determine the optimal deployment size for a given workload, or the maximum performance that can be reached using given resources. Unlike custom sizing and exhaustive search, the deployment sizes in directed search are computed *online* during the testing process.

The VITAL orchestrator starts with the minimal deployment size (*e.g.*, one instance with minimum flavor per component). After tests on this initial deployment size are complete, the VITAL orchestrator analyzes the resource utilization traces when the system capacity point is reached, and determines the VNF component with highest resource usage. Directed search then scales up or out. We are still investigating the integration of hybrid scaling in directed search to increase the search space. The testing process terminates either when the given workload is reached or when the given resource is exhausted.

With exhaustive search and component-aware directed search, the solution space may grow exponentially as the resources and the number of components increase. However, the available VM sizes are limited, which reduces the number of possible solutions. For instance, we only use three different VM sizes `cw1.small`, `cw1.medium` and `cw1.large` in our demonstrations. For exhaustive search, since all possible deployment sizes are computed offline, this problem does not affect the performance at run time. For component-aware directed search, we limit the users to use either scaling up or scaling out in one set of tests. If the users choose scaling out, only one VM size can be defined. Therefore, the users can determine the optimal VM size of the VNF by scaling up and the optimal number of VMs for each component by scaling out. These strategies yield a reasonable solution space size in our evaluation.

B. Framework Demonstrations

Using NFV-VITAL, we conduct three demonstrations on Clearwater and two types of IDS software: Snort and Suricata.

1) Clearwater Demonstration: Unlike the Clearwater tests in section II, we use component-aware directed search in this demonstration. The user input file has a 6-vCPU resource restriction, CPU core pinning, and scaling up. This input file can be found on our webpage (denoted in the footnotes).

NFV-VITAL first tests `bn1s-sp1s-hs1s` and finds that the bottleneck is CPU usage on the Sprout component. It then scales up to `bn1s-sp1m-hs1s` adding one more vCPU to Sprout. Now Homestead CPU usage becomes the system bottleneck. NFV-VITAL then scales to `bn1s-sp1m-hs1m`. Sprout becomes the bottleneck again. However, we cannot scale up the Sprout instance to `cw1.large` because of the 6-vCPU threshold we specified. NFV-VITAL then repeats the same process using core pinning. Fig. 6 shows how SCR changes when applying the same *reg-dereg* traffic. Based on this result, we conclude that the optimal deployment size with a 6-vCPU restriction is `bn1s-sp1m-hs1m` and the maximum system capacity is 600 calls/second of reg-dereg traffic.

As shown in section II, different components of Clearwater exhibit different CPU usage patterns: Sprout is the bottleneck with twice the CPU usage of Bono when we assign them the same resources. However, the optimal Clearwater deployment

size that NFV-VITAL finds in this demonstration yields similar CPU usage for Sprout, Homestead, and Bono (75% to 85%).

We also tested with *reg-invite* traffic, with 21 requests and responses in each call. The optimal deployment size that NFV-VITAL finds is also `bn1s-sp1m-hs1m`. The maximum system capacity is 150 calls/second, however. This confirms that with different types of traffic, the same VNF can exhibit significantly different system capacity.

In summary, NFV-VITAL can locate the performance bottleneck of the entire VNF under different conditions. Given the resources or workload, NFV-VITAL can determine the most efficient deployment size in an automated fashion.

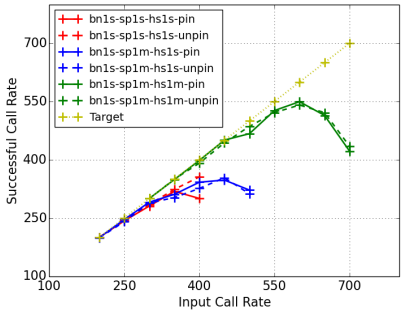


Fig. 6: NFV-VITAL demonstration on Clearwater

2) IDS Demonstration: In this demonstration, we compute the packet processing ability of Snort (version 2.9.7.3) and Suricata (version 2.0.8) using custom sizing. Since both IDSs are single VNF, we test them with three deployment sizes: one small instance, one medium instance, and one large instance.

Similar to the baseline test in [14], we keep the default configuration of both Snort and Suricata using the same VRT rule set. The only change is that we enabled the "set-cpu-affinity" option on Suricata. The test environment involves four VMs in the same virtual subnet running hping3 [15] to generate UDP traffic at the same rate with packet size 64 bytes. All traffic is mirrored to a separate VM running IDS software. We add an additional rule to match UDP packets with the keyword "malicious" in the payload. A VM that generates traffic matching this rule is identified as a malicious node. We vary the number of malicious nodes from 0 to 4 to generate five types of traffic: 0% malicious traffic, 25% malicious traffic, 50% malicious traffic, 75% of malicious traffic, and 100% malicious traffic. The workload is defined by the packet generation speed from all four senders, and the system performance is represented by the packet processing speed of the IDS – both quantities are in kilo-packets per second (kpps).

Fig. 7 shows how the packet processing speed on Snort changes with increasing traffic. We observe that for the same type of traffic, scaling the Snort instance from small to large yields little improvement. Investigating the resource utilization traces that our framework collected, we find that at a given time, Snort only uses a single vCPU even with medium and large instance sizes. The rest of the vCPUs have low utilization. This confirms the fact that Snort scales poorly on multi-core systems. A potential solution to this problem is to run multiple instances of Snort and configure them to handle partial traffic in the same VM. However, this may lead to false negatives for some stateful attacks. When we increase the malicious traffic proportion, we find that the processing speed of Snort significantly decreases due to the rule matching overhead.

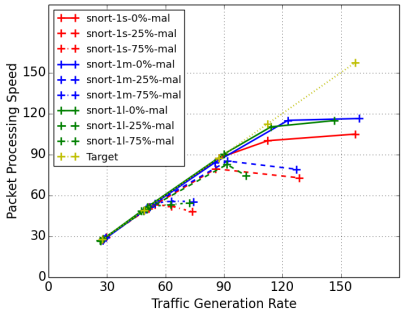


Fig. 7: NFV-VITAL demonstration on Snort

Fig. 8 shows that Suricata exhibits very different behavior from Snort in the same testing environment: increasing the proportion of malicious traffic does not impact the packet processing speed as much as on Snort. Suricata benefits from native multi-threading support and decoupling of packet acquisition, decoding, detection, and output into different modules. For instance, when increasing malicious traffic, detection threads consume more CPU cycles. Since they use different vCPUs, this does not affect the performance of packet acquisition, leading to better performance than Snort. With no malicious traffic, multi-threading only boosts packet processing speed of Suricata by 30% over Snort. However, scaling Suricata from a medium to a large instance does not improve its performance. This is because Suricata only uses 250 out of 400 (4×100) of the CPU resources in a large instance. Further tuning of the Suricata "threading" options may increase performance in the case of large instances.

To conclude, with NFV-VITAL, users can compare the performance of VNFs in a controlled environment with multiple types of workload. NFV-VITAL aids in understanding performance differences and optimizing VNFs with different configurations.

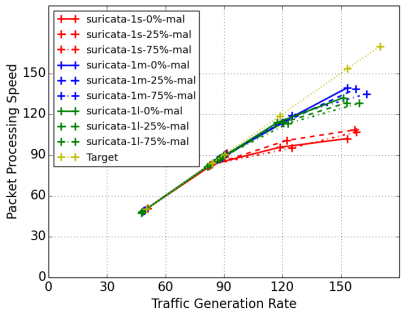


Fig. 8: NFV-VITAL demonstration on Suricata

IV. Related Work

The performance optimization of middleboxes long predates the recent interest in Network Functions Virtualization. In addition to the work mentioned in the introduction, a number of studies have applied clever techniques to enhance the performance of middleboxes. Dobrescu *et al* [16] proposed a general-purpose packet-processing system that combines ease of programmability with predictable performance, while supporting a diverse set of applications and serving multiple

clients with different needs. Anwer *et al* [17] describe the design of Slick, a prototypical control plane for network middleboxes. Martins *et al* [18] proposed a new middlebox platform with a smaller size, shorter booting time, and strong network processing ability.

Sekar *et al* [19] proposed CoMb to systematically explore opportunities for consolidation, both at the level of building individual middleboxes and managing a network of middleboxes. Using a prototype implementation in Click, they show that CoMb reduces the network provisioning cost and the load imbalance in the network. Gember *et al* [20] investigated application deployment in the cloud from various perspectives, including elasticity, network flow distribution, and virtual machine placement. They also advocate mechanisms that help exercise unified control over the key factors influencing middlebox operations [21]. They realize a software-defined middlebox networking framework to simplify management of complex, diverse functionalities. More recently, they have proposed a control plane called OpenNF to address race conditions, bound overhead, and accommodate a variety of network functions [22].

V. Discussion and Conclusions

This work has demonstrated the importance of VNF performance characterization. Based on our extensive case study of the *Clearwater* IMS VNF, and two IDS VNFs (Snort and Suricata), we proposed *NFV-VITAL*, a general framework for VNF characterization. We observe that *scaling up* VM resources (vCPU and memory) for different *Clearwater* components results in almost linearly proportional increase in system capacity. This is because a single VM per component instantiation is compute-bound for registration-deregistration workload. Since it is impossible to infinitely increase vCPU and memory allocation to a single VM, NFV orchestration controllers have to resort to *scale out* by instantiating multiple VM clusters for each component. However, comparing different equivalent instantiations with the same vCPU and memory resources, scaling out performs worse than scaling up. This is due to clustering overhead associated with underlying subsystems like *memcached* (for Sprout) and *cassandra* (for Homestead). The clustering overhead is amortized as the size (resource allocation) of individual VM instances increases. We also characterized system performance with hybrid scaling instantiations. Since different components are under different loads, such characterization can be leveraged to design control algorithms for selecting optimal VM sizing and clustering for different components on the available infrastructure. Analysis of performance degradation due to clustering enabled us to detect deficiencies in the load balancing approach of *Clearwater*. Similarly, we leveraged the NFV-VITAL framework to validate the limitations of Snort compared to Suricata. Scaling up Snort by allocating more vCPU resources is not an effective option, whereas it is effective for Suricata. The NFV-VITAL characterization framework can demystify unexpected performance degradation.

We are currently completing the prototype implementation of NFV-VITAL, and characterizing additional VNFs (*e.g.*, [23], [24]). Data collected will serve as input to our future work on automated hybrid scaling and on the importance of different configurations, such as SR-IOV and CPU pinning. NFV orchestration engines can optimize the scaling strategy for better and more balanced overall system performance. In case of complex VNFs like *Clearwater*, NFV-VITAL can be used for preemptive resource flexing of different components before individual components are overloaded. VNF characterization can also form the analytic basis for capacity planning for given workloads and infrastructure resources. Another application of NFV-VITAL is to fine tune the VNF implementation and performance with "devops" VNF development models enabled by NFV adoption.

References

[1] "ETSI Network Functions Virtualisation (NFV) Architectural Framework," http://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.02.01_60/gs_NFV002v010201p.pdf.

[2] J. Liu, "Evaluating standard-based self-virtualizing devices: A performance study on 10 gbe nics with sr-iov support," in *IPDPS '10*. IEEE, 2010.

[3] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan, "High performance network virtualization with sr-iov," in *HPCA '10*. IEEE, 2010.

[4] "Supporting Evolved Packet Core for One Million Mobile Subscribers with Four Intel Xeon Processor-Based Servers," https://networkbuilders.intel.com/docs/MESH_Group_Intel_EPC_TB_FINAL.pdf.

[5] G. Wang and T. S. E. Ng, "The impact of virtualization on network performance of amazon ec2 data center," in *INFOCOM '10*. IEEE, 2010.

[6] "Clearwater," http://www.projectclearwater.org/.

[7] "OpenStack," http://www.openstack.org/.

[8] "Memcached," http://memcached.org/.

[9] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, Apr 2010.

[10] "SIPp," http://sipp.sourceforge.net/.

[11] "Clearwater Performance and Load Monitor," http://www.projectclearwater.org/clearwater-performance-and-our-load-monitor/.

[12] L. Cao, P. Sharma, and S. Fahmy, "NFV-VITAL Results and Sample User Input," https://www.cs.purdue.edu/homes/fahmy/nfv-vital/.

[13] "Heat developer documentation," http://docs.openstack.org/developer/heat/.

[14] J. S. White, T. Fitzsimmons, and J. N. Matthews, "Quantitative analysis of intrusion detection systems: Snort and suricata," in *SPIE Defense, Security, and Sensing*. International Society for Optics and Photonics, 2013.

[15] "hping3," http://www.hping.org/.

[16] M. Dobrescu, K. Argyraki, and S. Ratnasamy, "Toward predictable performance in software packet-processing platforms," in *NSDI '12*. USENIX, 2012.

[17] B. Anwer, T. Benson, N. Feamster, D. Levin, and J. Rexford, "A slick control plane for network middleboxes," in *HotSDN '13*. ACM, 2013.

[18] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *NSDI '14*. USENIX, 2014.

[19] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *NSDI '12*. USENIX, 2012.

[20] A. Gember, R. Grandl, A. Anand, T. Benson, and A. Akella, "Stratos: Virtual middleboxes as first-class entities," University of Wisconsin-Madison, Tech. Rep., 2012, technical report TR1771.

[21] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella, "Toward software-defined middlebox networking," in *HotNets-XI*. ACM, 2012.

[22] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," in *SIGCOMM '14*. ACM, 2014.

[23] "CloudRouter," https://cloudrouter.org/.

[24] "HP VSR1000 Virtual Services Router Series," http://www8.hp.com/us/en/products/networking-routers/product-detail.html?oid=5443163.