Paper published in: Elsevier Future Generation Computer Systems (Volume 99, October 2019, Pages 164-176), https://doi.org/10.1016/j.future.2019.04.007

Overload Control for Virtual Network Functions under CPU Contention

Domenico Cotroneo, Roberto Natella, Stefano Rosiello*

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione (DIETI)
Università degli Studi di Napoli Federico II, Via Claudio 21, 80125, Naples, Italy

Abstract

In this paper, we analyze the problem of overloads caused by physical CPU contention in cloud infrastructures, from the perspective of time-critical applications (such as Virtual Network Functions) running at guest level. We show that guest-level overload control solutions to counteract traffic spikes (e.g., traffic throttling) are counterproductive against overloads caused by CPU contention. We then propose a general guest-level solution to protect applications from overloads also in the case of CPU contention. We reproduced the phenomena on a IP Multimedia Subsystem (IMS) testbed based on OpenStack on top of KVM. The results show that the approach can dynamically adapt the service throughput to the actual system capacity in both cases of traffic spikes and CPU contention, by guaranteeing at the same time the IMS latency requirements.

Keywords: Overload control, Resource over-commitment, CPU contention, Traffic throttling, Network function virtualization, IP Multimedia Subsystem

1. Introduction

Overload conditions are a major cause of cloud service failures [1]. These conditions occur when the incoming traffic exceeds the available capacity (e.g., by tens, or even hundreds of times). Overloads are not only due to exceptional *traffic spikes* (e.g., due to mass events): an important, and often underestimated, cause of overload conditions is the *resource contention* inside the cloud infrastructure, whose effect is to decrease the available capacity for serving

*Corresponding author.
Email addresses: cotroneo@unina.it (D. Cotroneo), roberto.natella@unina.it (R. Natella), stefano.rosiello@unina.it (S. Rosiello)

the incoming traffic. The risk of resource contention arises due to *over-commitment* of cloud computing infrastructures, e.g., the provider incorrectly allocates resources to too many users due to bad capacity planning or operator mistakes [2, 3]. Moreover, resource contention can also be caused by *faults* in the infrastructure, e.g., a background service may consume too much resources because of a software bug, a failed update, or configuration problem [4, 5].

To mitigate overload conditions, time-critical applications include overload control mechanisms at the guest-level (i.e., inside VMs), such as real-time rate adaptation [6], graceful performance degradation through brown-out [7] and traffic shaping [8, 9] to reject the application-level traffic in excess. Unfortunately, *physical CPU contention* at the infrastructure-level has severe side effects on these existing overload control algorithms, which are designed with only traffic spikes in mind. These algorithms collect resource utilization metrics at run-time, in order to tune the amount of traffic that can be accepted by the service [9, 10, 11]. To these algorithms, traffic spikes and physical CPU contention both appear as a saturation of the virtual CPU, but these conditions need to be managed differently. As an example, when CPU contention occurs, the virtual CPU of the VM becomes saturated since the hypervisor schedules less virtual CPU time, in order to share the physical CPU among several competing VMs and host processes. Such condition can be misinterpreted by the application as a traffic spike. If the application discards part of the incoming traffic to compensate for CPU saturation, the hypervisor can preempt more virtual CPU time form the VM (i.e., the CPU bandwidth that has just been freed by discarding traffic) in favor of the competing VMs, leading again to a saturated virtual CPU. This sequence of events triggers a *vicious cycle* that degrades the quality of service (QoS).

In this paper, we analyze the problem of overloads that are caused by physical CPU contention, and address the limitations of traditional traffic throttling at handling these conditions. We study this problem in the context of *virtual network functions* (VNFs), i.e., network appliances implemented in software, and deployed on industry-standard COTS hardware and cloud computing technology, which are becoming very popular due to the increasing industry interest in the *Network Function Virtualization* (NFV) paradigm. Virtual network functions are a class of cloud applications with very strict performance and high-availability requirements, where the overload problem assumes a critical importance. The contributions of this paper include:

- We present a critical discussion of the problem of overload control from the perspective of time-critical applications deployed on virtualization technology. In particular, we analyze the issue of detecting physical CPU contention using guest-level CPU utilization metrics, and how this issue impacts on overload control algorithms.

- We propose a solution to enhance overload control algorithms, by making them able to handle overload conditions caused both by traffic spikes, and by physical CPU contention.

- We present a experimental study of overload control algorithms with respect to both traffic spikes and physical CPU contention, in the context of an NFV-oriented IMS case study. The results demonstrate that existing overload control algorithms are vulnerable to physical CPU contention, which can cause severe latency degradation. Moreover, we show that the proposed approach is robust against physical CPU contention, as it can assure that the IMS application can guarantee low latency and high throughput at the same time.

The key point of the proposed solution is that it is designed to be deployed *with the application at the guest-level* (i.e., inside a VM). This aspect is especially relevant in the case of *NFV Infrastructures-as-a-Service* (NFVIaaS), where a time-critical VNF has little visibility and control on the underlying physical resources (e.g., on scheduling priorities at the physical CPU level). To the best of our knowledge, no previous work has addressed the problem of physical CPU contention from the guest-level perspective. Moreover, the proposed solution is complementary to recovery mechanisms at the infrastructure-level, by mitigating the overload during the period while the infrastructure recovers the available capacity (e.g., through elasticity and migration), which can take several minutes [12, 13].

In the following, we introduce the problem of physical CPU contention in Section 2. We present the proposed overload control strategy in Section 3, and the experimental evaluation in Section 4. Section 5 discusses related work. Section 6 concludes the paper.

2. Overview of CPU overloads and CPU utilization metrics

In this section, we expose the problem of overload conditions, how to interpret CPU utilization metrics, and the pitfalls for overload control strategies when using these metrics.

Ideally, the input traffic for a service should not exceed its **engineered capacity**, that is, the maximum amount of input traffic that can be served while achieving SLAs. SLAs typically require a low probability of failures (such as, traffic loss or processing errors) and low latency (such as, the time to process or respond to an individual traffic unit). These requirements are especially demanding in the case of the telecom domain [14, 15], where the engineered capacity is carefully planned at design time, by allocating computing resources according both to cost considerations, and to the expected **reference workload**: for example, according to the expected rate of *busy-hour call attempts* (BHCA) in the case of a VoIP service.

In the context of IaaS, the designers of VNFs need to plan in advance the flavor and the expected amount of VMs; for example, a common rule-of-thumb is to plan for VMs such that each VM consumes at most 90%, or some other threshold (the **engineered level**), of the available virtual CPUs under the reference workload, leaving a small amount of residual capacity as a factor of safety [1, 16]. Overload conditions saturate the capacity of virtual CPUs; in these cases, the VNFs should **throttle** the input traffic (i.e., rate-limit by dropping or rejecting requests) in order to assure that the traffic processed by the VNFs is within the engineered capacity and can meet the SLAs. This strategy is further discussed in Section 3.

Physical resource contention is a special case of overload condition, in which the available capacity of the VNFs is reduced due to competition. However, the behavior of the system is different than the case of traffic spikes. To illustrate the problem, we consider thorough the paper a generic example (in Figure 1) of a VNF with a 1-GHz virtual CPU, deployed on a 2-GHz physical CPU. Therefore, the CPU quota of the VM is 50% of the physical CPU. In this example, we assume that the engineered capacity of the VNF uses 75% of the virtual CPU under the reference workload. From inside the VM (Figure 1, CASE 1), the OS measures the virtual CPU utilization by counting the virtual CPU cycles that have been spent *busy* at executing applications or the OS kernel, and *idle* at waiting for I/O or without any workload (i.e., **vCPU utilization** = $^{\text{busy}}/_{\text{busy+idle}}$).

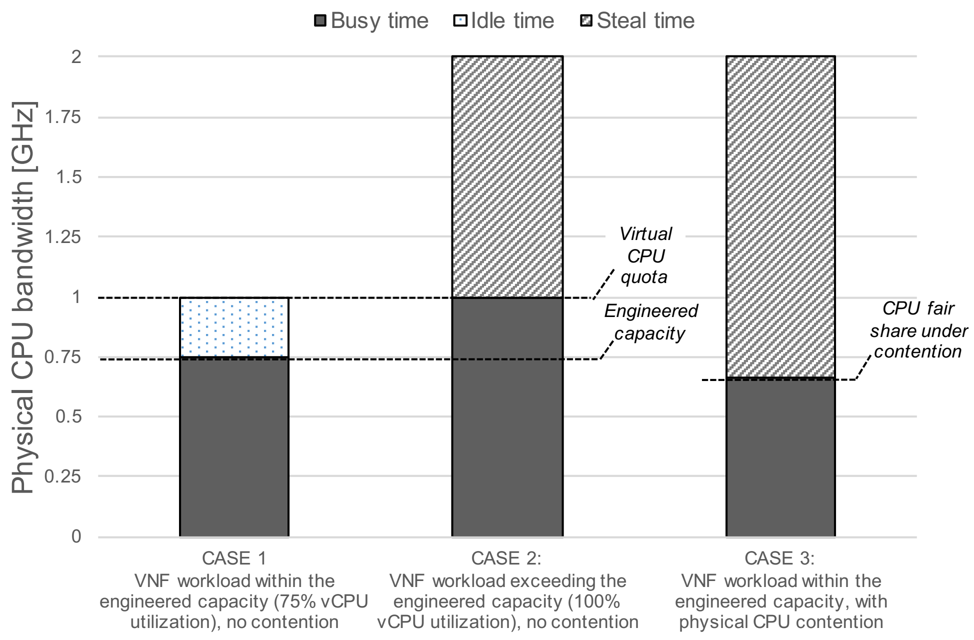


Figure 1: CPU utilization metrics under three scenarios.

When the input traffic overloads the VNF (Figure 1, CASE 2), the virtual CPU utilization raises to 100% to serve all of the traffic, and hits the CPU quota at 1 GHz enforced by the hypervisor.

In addition to these metrics, we also consider the **CPU steal time** metric, which is also influenced by overload conditions, but can be mistakenly considered as an indicator of physical CPU contention. We use the name “CPU steal” in reference to the metric available in Linux and in the KVM and IBM z/VM hypervisors [17, 18, 19]; an equivalent metric is also available in other hypervisors such as VMware ESXi, Xen and Microsoft Hyper-V, respectively under the name “CPU Stolen Time” [20, 21] and “CPU Wait Time Per Dispatch” [22]. This metric is provided by hypervisors to VMs, e.g., through hypervisor calls. In all these systems, this metric is technically defined as *the time that a virtual CPU is ready to execute, but it is waiting to execute on the physical CPU*. In other terms, the metric represents the time spent by the virtual CPU on the hypervisor’s scheduling queue. The term “steal” refers to CPU cycles that a VM spends waiting because either the hypervisor or other VMs are using the physical CPU (for example, the hypervisor is using CPU cycles to emulate an I/O device). However, in most situations no CPU cycle is actually “stolen” from the VM, as the hypervisor still assures the CPU quota for the VM, and that the VM is eventually scheduled; it would be better understood as an “involuntary wait” time. For example, in the CASE 2 of Figure 1, the VM is put on hold after that it consumes its virtual CPU quota; thus, the rest of the physical CPU time is accounted as “steal time” from the perspective of the VM, since it is waiting on the scheduling queue. Even in the case of low workload, it is still possible that a moderate share of CPU time is accounted as stolen, e.g., when two VMs are sporadically ready to execute at the same time. Thus, steal time is not a sufficient condition for an overload condition.

The third scenario involves physical CPU contention (Figure 1, CASE 3). In this case, we are assuming that 3 VMs with equal priority are scheduled on the same physical CPU (e.g., because of overcommitment, bug or misconfiguration of the infrastructure). The 3 VMs all have a CPU quota set to 1 GHz, and an engineered capacity that uses 75% of the virtual CPU (as in the previous two scenarios). Since the total CPU demand ($0.75 \cdot 3$ GHz) exceeds the capacity of the physical CPU (2 GHz), the hypervisor equally divides the CPU bandwidth among the VMs, where each virtual CPU actually gets a slice (**fair share**) of 0.66 GHz (i.e., 33% of the physical CPU time). Since the VNF is ready to execute even after consuming this slice (as the workload exceeds the virtual CPU capacity), the rest of the physical CPU time (66%) is accounted as steal time for the VM.

Both in CASE 2 and CASE 3 of Figure 1, the VNF is in an overload condition. However, if the VNF is deployed on IaaS, it cannot easily distinguish between the two cases, since the VNF cannot inspect or control the underlying infrastructure. From the perspective of the VNF, only looking for high virtual CPU consumption or for high CPU steal time does not suffice to discriminate between a traffic spike or physical resource contention. The only difference between the two cases is that the actual CPU share of the VM (0.66 GHz) is lower than the original CPU quota (1 GHz). Therefore, to address both these cases, the proposed overload control approach throttles the workload by adapting to the CPU share (either the quota or the fair share) that is actually available to the VNF.

3. Overload control strategy

In this section we discuss the problem of overload control in the context of NFV services. In detail, in Section 3.1 we present a well known feedback-loop based overload control in order to discuss how this kind of approaches can cause the problem of the vicious cycle (Section 3.2) during physical cpu contention. Finally, in Section 3.3 we propose an enhancement to the feedback-loop based overload control along with a technique to avoid the vicious cycle.

To recover from overload conditions, the long term solution would be to meet the high demand by scaling up the computing resources, or to relieve physical resource contention by shutting down other services that have a lower priority or that are hogging the resources.

However, these recovery actions can take several minutes, even in an optimistic case. During this transient period, VNFs are still exposed to the risk of outages. This problem is exemplified in Figure 2. Typically, the capacity of a VNF is designed to support up to a *reference load* (point C1), in terms of requests per seconds completed with an acceptable quality of service. When a mass event or a cascade failure occurs, the network load exceeds the reference load. If the network does not have enough resources to process all of the incoming flows, then the individual requests will not get enough computing resources to meet SLA requirements. For example, even if many requests are processed at the same time (the interval between the points C1 and C2 in the Figure 2), the latency of the responses can become exceedingly long. Beyond a given point (point C2 in the Figure 2), if the overload condition is not managed, the rate of successfully-processed traffic can significantly degrade because of too much resource competition [23, 11]. Handling too much traffic also increases the likelihood of *software failures* such as failed resource allocations, timeouts, and race conditions [24, 25]. Therefore, long-term recovery actions should

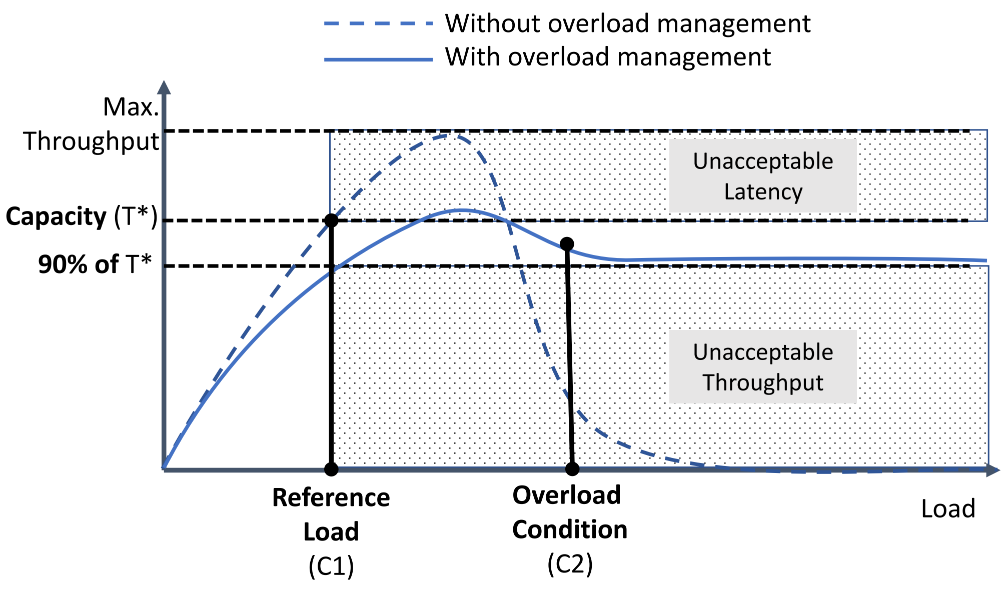


Figure 2: Network performance under overload conditions.

be combined with short-term solutions for throttling the traffic, in order to allow only the traffic that can be processed with the currently available capacity entering the system [8, 26, 9, 10]. Ideally, using overload control, the network should maintain a steady throughput (for example, no lower than 90% of the engineered capacity (T*), the continuous curve in the Figure 2) even under an overload condition.

3.1. Basic feedback control-based overload control

In the case of traffic spikes in VNFs (the CASE 2 in Figure 1), the throttling algorithm should reject part of the traffic, in order to reduce the virtual CPU utilization to the engineered capacity (i.e., to return to the CASE 1 in Figure 1). For example, *increase/decrease algorithms* are a popular solution to tune the amount of traffic to be accepted (e.g., the window size for packet flow control) [27, 28, 29] by decreasing the traffic when the network is overloaded (e.g., by a constant or multiplicative factor), or by increasing the traffic otherwise. This approach has been recently applied in the context of NFV [30], using a heuristic criterion to tune the traffic that a VNF can serve (*capacity*):

$$\text{capacity} = \frac{\text{processed\_traffic}}{\text{current\_vcpu\_usage}} \cdot \text{reference\_vcpu\_usage} \tag{1}$$

where the first factor estimates the cost per traffic unit (in terms of virtual CPU cycles), which is multiplied by the reference virtual CPU budget (i.e., the engineered level) to get the total amount of traffic that can be correctly served. When the virtual CPU utilization exceeds the engineered level, the heuristic drops a percentage of the incoming traffic (*drop rate*):

$$\text{drop\_rate} = 100 \cdot \left(1 - \frac{\text{capacity}}{\text{incoming\_traffic}}\right) \tag{2}$$

in which the higher the gap between capacity and the incoming traffic, the higher the drop rate. The drop rate is periodically updated every few seconds, and is capped between 0% and 100%. In the case of a traffic spike, the virtual CPU utilization increases, thus the heuristic lowers the *capacity* and increases the *drop rate*; as result, the virtual CPU utilization settles again around the engineered level.

3.2. The control loop vicious cycle

The throttling heuristic presented in the previous section, may not work correctly in the case of physical CPU contention. We consider again the example of Section 2, where the physical CPU contention leads to the following chain of events (see also the Figure 3):

1. Due to the contention, the hypervisor allocates less physical CPU time to the VM (0.66 GHz, as in the CASE 3 in Figure 1). As a result, the current workload saturates the VNF, and the virtual CPU utilization becomes 100% (i.e., the ratio $^{\text{busy}}/_{\text{busy+idle}}$), which is higher than the reference CPU utilization (e.g., 75% in the example).
2. The heuristic increases the drop rate to reduce the load. The virtual CPU utilization then settles around 75%. It is important to note that the 75% of the virtual CPU is equal to $0.66 \cdot 75\% = 0.5$ GHz of physical CPU. The residual 25% of the virtual CPU (i.e., $0.66 \cdot 25\% = 0.166$ GHz of physical CPU) becomes idle.
3. Due to the physical CPU contention, the hypervisor opportunistically schedules these idle CPU cycles for the demand of other VMs or processes on the host machine. Thus, the virtual CPU is not anymore idle, and virtual CPU utilization becomes again 100%.
4. The heuristic further increases the drop rate, to reduce again the virtual CPU utilization down to 75% (as in the previous step 2). The virtual CPU now consumes $0.66 \cdot 75\% \cdot 75\% = 0.375$ GHz of physical CPU.
5. The hypervisor preempts again the idle CPU time. The heuristic enters a vicious cycle where the virtual CPU utilization is reduced more and more.

The vicious cycle is caused by the *work-conserving* behavior of hypervisor schedulers (i.e., they ensure that the CPU is never idle if there is at least one VM ready for execution) [31, 32, 33]. The VNF yields to the hypervisor part of its virtual CPU time, by dropping part of the incoming traffic. In the case of physical CPU contention, in which several VMs or processes on the host machine are demanding more CPU time than the available physical CPU, the hypervisor scheduler uses the freed CPU cycles to meet these demands. Then, the virtual CPU shrinks again and causes the vicious cycle. In general, the feedback control loop approach (not limited to the heuristics of eqs. (1) and (2), but any other control rule based on virtual CPU utilization) can be vulnerable to physical CPU contention, due to the distortion of virtual CPU utilization metrics.

3.3. Enhanced feedback control-based overload control

To address the problem of overload control under physical CPU contention, we extend the feedback control loop approach with an additional mechanism to break the vicious cycle. The design goal of the approach is to assure that the VNF gets no less than its fair share of the physical CPU even under contention (e.g., 0.66 GHz in the previous example); and, at the same time,

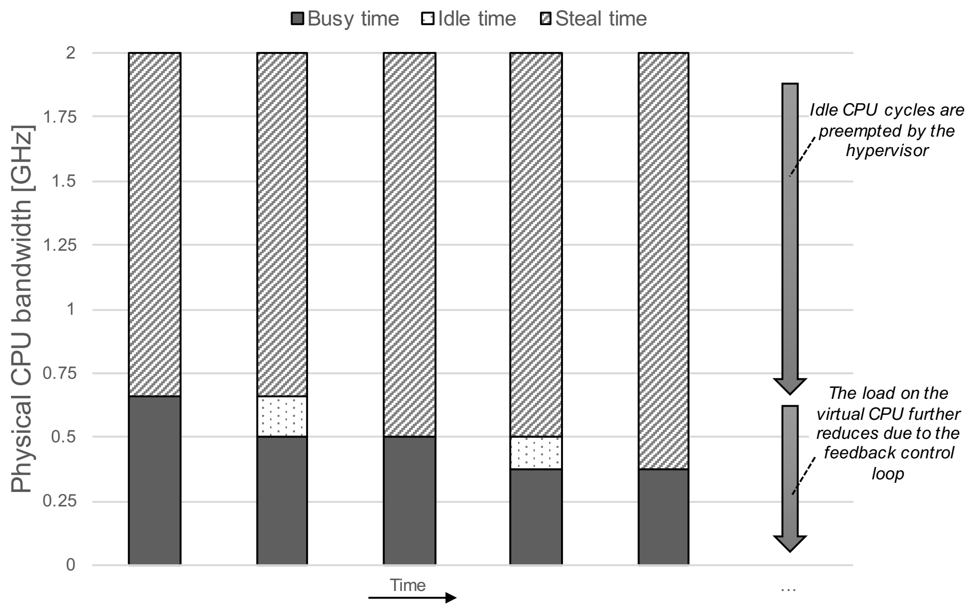


Figure 3: Chain of events caused by physical CPU contention.

that the virtual CPU utilization inside the VNF settles at the engineered level (e.g., 75% of the virtual CPU in the example). This condition is showed in Figure 4: the available virtual CPU under physical contention reduces to 0.66 GHz; since this virtual CPU is not sufficient to reach the original engineered level (0.75 GHz), we still apply the feedback control loop to reduce the virtual CPU utilization down to 75% of the available virtual CPU (i.e., 0.5 GHz of physical CPU). This is the same condition of the step 3 of the vicious cycle; we break the cycle at this point, using the following approach.

We introduce a mechanism into the VNF to avoid the preemption of idle virtual CPU cycles under physical CPU contention. This effect can be obtained in different ways depending on the guest OS used in the VNF. The most generic approach is to add a *placeholder* process (one process per virtual CPU of the VM) that actively consumes virtual CPU cycles to avoid preemption by the hypervisor; it executes a CPU-bound task for the sake of consuming virtual CPU cycles. The placeholder process should execute at minimal priority on the guest OS of the VNF; moreover, it should be configured as a *batch* task in order not to take away any virtual CPU cycle from the VNF software (i.e., the placeholder only uses the virtual CPU when the VNF is not executing). For example, this effect can be obtained on Linux by setting the `SCHED_BATCH` or `SCHED_IDLE` scheduler class for the task [34], and on Windows by setting an idle trigger [35]. Yet another approach is to configure or to modify the idle loop of the guest OS [36]. As a result, the placeholder takes the place of the idle time of the virtual CPU, as in Figure 4: at any given time, the virtual CPU is either executing the VNF or the placeholder process, and the virtual CPU consumes the residual physical CPU cycles granted by the hypervisor scheduler. This behavior breaks the vicious cycle, since the hypervisor cannot preempt the virtual CPU cycles that are freed by the feedback control loop. Moreover, settling the virtual CPU utilization at the

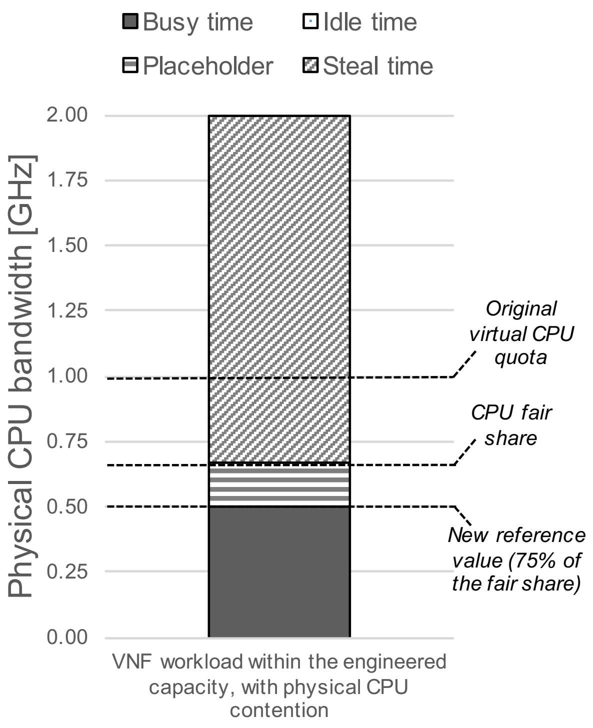


Figure 4: CPU utilization metrics under physical contention, with virtual CPU placeholder.

engineered level provides a “margin of safety" (e.g., to compensate for small random workload fluctuations) as in the case of the original engineered level, since the VNF software can preempt the placeholder process at anytime.

We enable the placeholder process on the condition that the CPU steal time spans all the physical CPU not used by the VM. This condition occurs when the VNF consumes its available virtual CPU, either because of a traffic spike that saturates the virtual CPU quota (CASE 2 in Figure 1), or because of physical CPU contention that reduces the available virtual CPU (CASE 3 in Figure 1). We apply the same solution regardless of which one of these two cases is causing the saturation of the virtual CPU. The solution still applies the feedback control loop, but excluding the CPU consumption of the placeholder process from the virtual CPU utilization metric, that is:

$$\text{vCPU utilization} = \frac{\text{busy}_{\text{all}} - \text{busy}_{\text{placeholder}}}{\text{busy}_{\text{all}} + \text{idle}} \quad . \tag{3}$$

For example, in Figure 4, the virtual CPU utilization is 75% if the utilization of the placeholder is not included. The virtual CPU utilization metric (i.e., the dependent variable controlled by the feedback loop) is thus not influenced by the presence of the placeholder process (which only opportunistically consumes the idle virtual CPU cycles). Therefore, in the case of traffic spikes, the proposed feedback control loop still works as in previous work [30]. In the case of physical CPU contention, the placeholder avoids the interaction between the feedback control loop (that frees the virtual CPU) and the hypervisor (that preempts the freed virtual CPU), thus allowing the feedback control loop to work correctly in this additional case.

Since CPU contention is a relatively rare event, we designed the placeholder process not to execute when there cannot be physical CPU contention. Since CPU steal time is a necessary condition (even if not sufficient, as discussed in Section 2) for physical CPU contention, the placeholder process remains idle if there is no accounted CPU steal time (CASE 1 in Figure 1). The placeholder process becomes active (i.e., it consumes virtual CPU cycles) once it detects that the CPU steal time has peaked (which denotes that the virtual CPU is trying to exceed a limit), and runs for a fixed amount of time T_{active}. Once T_{active} has elapsed, the placeholder process returns in the idle state. Then, the placeholder process inspects again the CPU steal time to check whether the VNF is not anymore saturating its virtual CPU. If there is still either a traffic spike or CPU contention, the placeholder process continues to be active, repeating the check later. The T_{active} should be chosen according to the expected duration of the recovery actions, such as for scaling out, hot-fixing a bug, or migrating the services to another host machine. Eventually, the virtual CPU executes again on a non-overloaded physical CPU.

4. Experimental analysis

We performed experiments on an NFV IMS system to reproduce the problem of physical CPU contention, and to evaluate the effectiveness of overload control solutions, including both the basic and the enhanced feedback control-based approaches.

We executed experiments on a testbed based on the *Clearwater* open-source IMS system [37]. Clearwater is a complete, commercial-grade implementation of the IMS core network, including components (P-CSCF, S-CSCF, etc.) for SIP signaling, user authentication and authorization, charging, and other IMS functions. The architecture of the IMS is showed in Figure 5. The IMS components are intended to run in separate VMs and deployed on cloud computing infrastructures, and to support load balancing and horizontal scalability.

Our experimental testbed runs these components on three Dell PowerEdge servers, equipped with two 8-core 2.6 GHz Intel Xeon CPUs, connected by two Gigabit Ethernet networks, and attached to a Fiber Channel storage area network. The physical machines are managed using OpenStack (version Juno) and the KVM hypervisor (based on the Linux kernel version 3.10). Each Clearwater service is replicated in two VMs, configured with 1 virtual CPU and 4GB of RAM; each VM runs one VNF instance, and Ubuntu Linux 14.04 as guest OS. We use the *SIPp* workload generator [38] to exercise the IMS with register and call-setup requests. The IMS workload reproduces the typical message flows between subscribers, according to the SIP protocol. These flows are also adopted to test the Clearwater IMS, and the complete scenario used in our tests is available online [39].

Therefore, our workload reproduces a stressful traffic profile of 5 BHCA (i.e., Busy Hour Call Attempt) per user and 60 BHRA (i.e., Busy Hour Registration Attempt) per user. We regulate the workload intensity by varying the number of subscribers in order to reach the engineered level of the system. The engineered capacity of the experimental testbed is 40,000 subscribers, which can perform on average 660 registration requests and 55 call requests per second without SLA violations. The engineered level for the virtual CPU utilization is 75% under this reference workload.

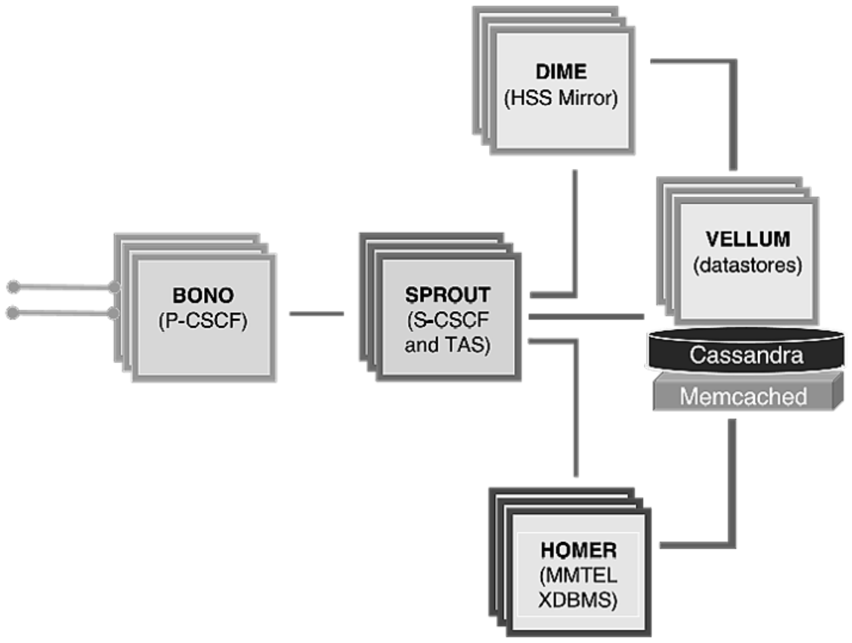


Figure 5: Architecture of the Clearwater NFV IMS.

We reproduce physical CPU contention by pinning an additional VM running a CPU-bound workload on the same physical CPU core running VMs of the IMS. CPU pinning and CPU scheduling affinities are often adopted as best-practices to optimize latency-sensitive applications [40], since they can optimize memory access in NUMA architectures and reduce the hypervisor scheduling latency. However, due to poor load balancing caused by CPU over-subscription, these practices also represent a potential cause of physical CPU contention. Moreover, manually setting CPU affinities can increase the risk of contention problems due to misconfiguration by the system administrators of the infrastructure [41]. Thus, this CPU contention scenario can be considered representative of typical issues occurring in time-critical applications running in virtualized environments.

In the following, we present and discuss three groups of experiments:

1. In the first group (Section 4.1), we consider a basic overload control solution, using the feedback control loop and heuristic that was introduced in Section 3. On this configuration, we reproduce overload conditions both due to traffic spikes and to physical CPU contention, in order to show that the feedback control loop can degenerate because of the vicious cycle.
2. In the second group of experiments (Section 4.2), we enhance the feedback control loop with the mechanism for breaking the vicious cycle, and reproduce the same overload conditions to evaluate the proposed solution.
3. In the third group (Section 4.3) we evaluate how the performance of overload control varies across different conditions, by considering both the basic and the enhanced feedback control loop solutions under different contention patterns.

During the following failure scenarios, we analyze the registration attempts and the registration throughput which include both new users and retries of failed attempts. After a failure, a user starts a back-off period (uniform between 0 and 2 min) before making a new registration attempt.

4.1. Basic feedback control-based overload control

We deployed the basic feedback overload control in the two Clearwater VMs running the IMS P-CSCF network function, since this component is a capacity bottleneck for our deployment configuration. In a first experiment we reproduce a workload surge which is 2.5 times higher than the engineered capacity level. The experiment lasts 15 minutes and it consists of two phases: in the first phase, we gradually introduce 40,000 subscribers and wait until the workload reaches the steady state at engineered level; in the second phase, starting at second 450s, we introduce in the system 100,000 additional subscribers, causing a workload surge and the overload of P-CSCF components. Figure 6 shows the registration request rate and throughput of the IMS during the experiment. Before the overload phase, the average registration throughput at steady state is 624 registrations per second; during the overload phase the average throughput is 634 registrations per second, with an average CPU utilization of 73.32%. The basic overload control solution described in Section 3 (eqs. (1) and (2)) has been able to successfully protect the IMS system: it avoids service failures for already-established sessions, by correctly estimating the capacity of the system and rejecting the requests in excess with respect to the capacity, which would saturate resources and cause failures both for the initial and the new subscribers. As a result, the throughput is constant despite the traffic spike.

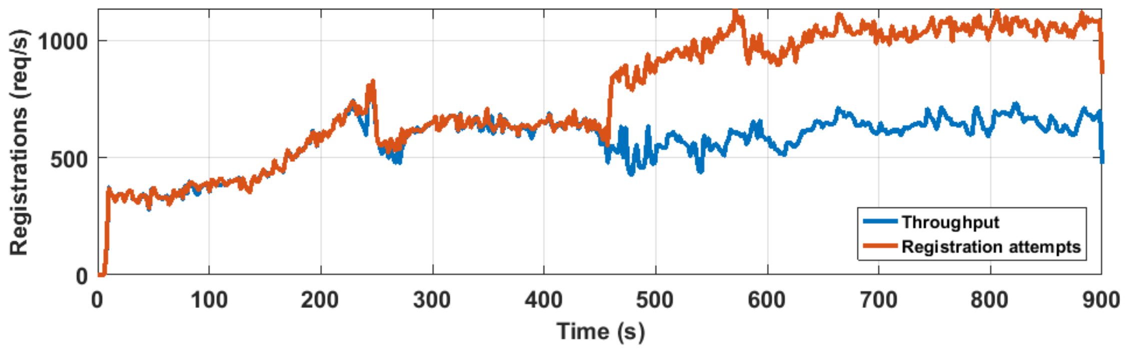


Figure 6: Performance of the IMS registrations during a 2.5x traffic spike (450-900s), using the basic feedback control loop.

In a second experiment, we consider again the basic control loop approach, and we reproduce an overload condition due to physical CPU contention. To this purpose, we pin the virtual CPU of the VMs running the IMS P-CSCF functions to a separate, reserved physical CPU. Then, we introduce a new VM running a CPU-bound workload (generated using the *cpuburn* tool [1]) and we pin its virtual CPU to the same physical CPU core of the IMS P-CSCF, in order to cause the contention. The experiment lasts 15 minutes (900s) and it is organized in three phases:

[1]The tool can be downloaded at https://patrickmn.com/projects/cpuburn/

during the first 5 minutes we generate a workload up to the engineered level; then, we activate the CPU-bound workload in the second VM to cause physical CPU contention for additional 5 minutes; finally, in the last 5 minutes of the experiment, we simulate the resolution of the CPU contention (e.g., as an effect of scaling out or migration of VMs to relieve the contention), by unpinning the virtual CPU of the CPU-bound VM. Figure 7 shows the registration request rate and the throughput of the system during the experiment with CPU contention.

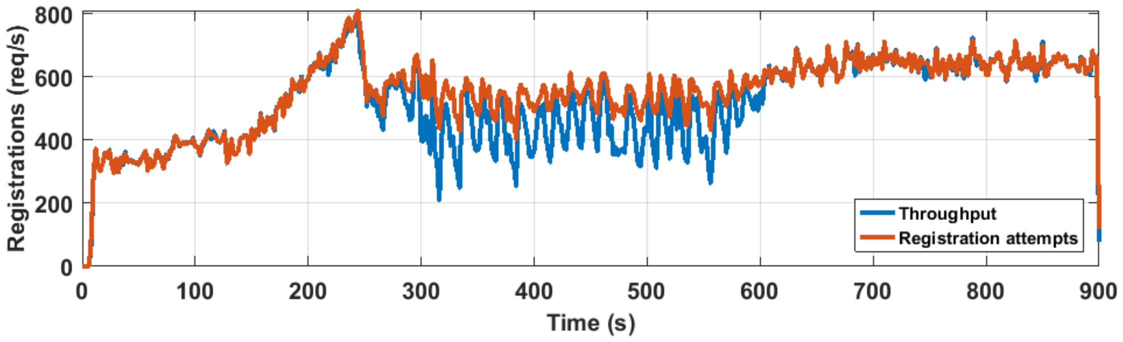


Figure 7: Performance of the IMS registrations during CPU contention (300-600s), using the basic feedback control loop.

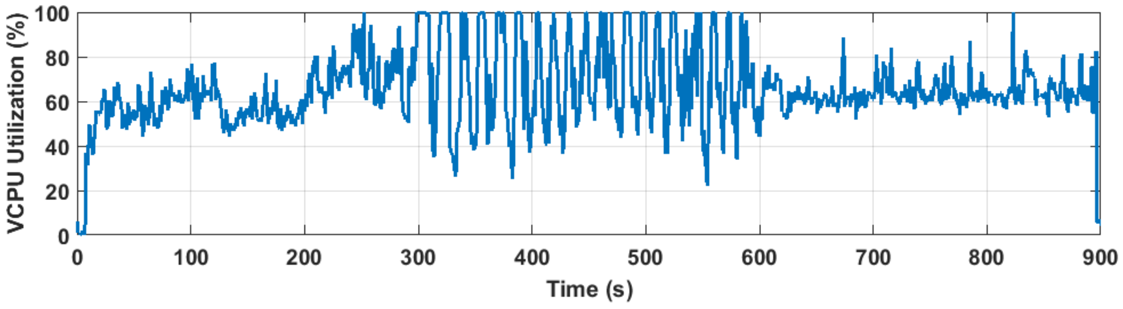


Figure 8: Virtual CPU utilization during CPU contention (300-600s), using the basic feedback control loop.

During the contention in the middle of the experiment, the throughput is affected by a high variability, which is a symptom that the basic control loop is unable to stabilize the load at the actual capacity of the VM. By looking at the virtual CPU usage during the experiment, showed in Figure 8, we noticed that as soon as we inject the CPU contention at min 5, the virtual CPU utilization raises to 100% since the hypervisor scheduler preempts physical CPU time from the virtual CPU, causing involuntary waits of the VM. As a consequence, the basic feedback control loop starts dropping part of incoming requests to reduce the virtual CPU utilization to the reference value of 75%. The Clearwater VM reduces its load and enters the vicious loop, since the CPU-bound VM takes advantage of the idle CPU time freed by the overload control mechanism. As a result, the virtual CPU utilization gradually drops down to about 20%. We also observed that the virtual CPU utilization saturates again to 100% after a period of approximately 10 seconds. This pattern is repeated periodically until the physical CPU contention is removed at minute 10, causing the high variability of CPU utilization. We found that this behavior is a side effect of the overload control mechanism, which sporadically resets the drop rate to 0 when the virtual CPU utilization becomes much lower than the reference value, thus admitting a high amount of input traffic and saturating again the virtual CPU. This high variability has a strong impact on the service latency, as further discussed in the next subsection.

4.2. Enhanced feedback control-based overload control

We deployed the enhanced feedback overload control strategy, and validated it by reproducing the same scenarios described in the previous subsection.

In the first experiment, after 450s, we caused a workload surge 2.5 times higher than the engineered capacity, and we evaluate the throughput of the IMS. As shown in Figure 9, in absence of physical CPU contention, the enhanced approach exhibits the same performance of the basic approach. Before the overload phase, the average registration throughput at steady state is 620 registrations per second while; during the overload phase the average throughput is 645 registrations per second with an average virtual CPU utilization of 74.55%. Therefore, our extension to the feedback loop does not cause any negative effect in the case of traffic spikes.

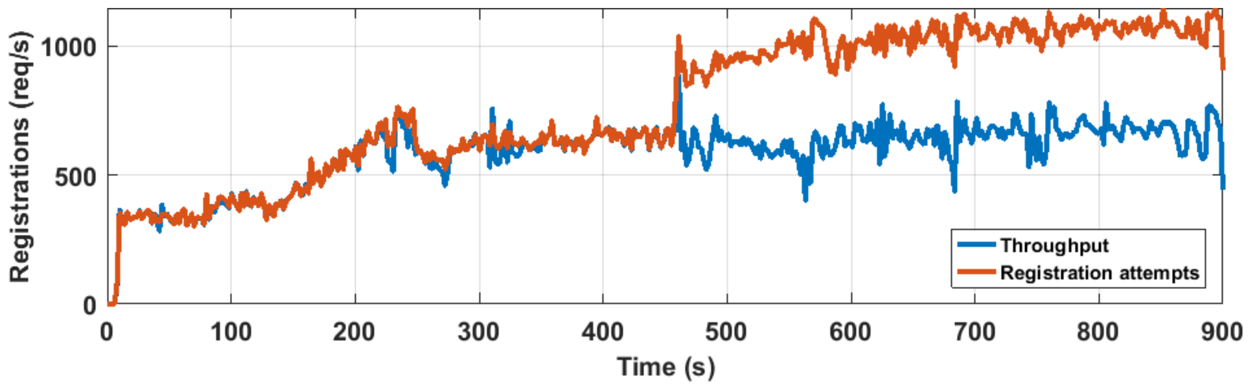


Figure 9: Performance of the IMS registrations during a 2.5x workload spike (450-900s), using the enhanced feedback control enabled.

In the second experiment, we reproduced the scenario with physical CPU contention, under the same conditions of Section 4.1. The time series in Figure 10 shows the throughput of the IMS. At 5 min, we enable the CPU-bound VM. The enhanced heuristic described in Section 3 timely detected a change in the system capacity. The during the contention the average throughput is reduced by about 32% and the system is able to complete 380 registration per second, with an average CPU consumption of 68%. Moreover, the throughput during the contention is more stable than the case with the basic feedback approach: since the placeholder process avoids the preemption of CPU time from the hypervisor, since the reference value of CPU utilization is not anymore a “moving target”, thus avoiding the variations of the heuristic for capacity estimation.

If CPU contention is not properly managed, the system accepts more requests that it can actually handle with the available CPU. However, many of the accepted requests are served with a poor quality of service, and many others fail in the middle of a session (therefore, the “goodput” of the system is actually lower than the throughput). A key goal of service providers is to ensure an appropriate QoS for users that are admitted into the system, and to gracefully handle users that cannot be admitted (e.g., to notify an overload status without starting a session that cannot be assured).

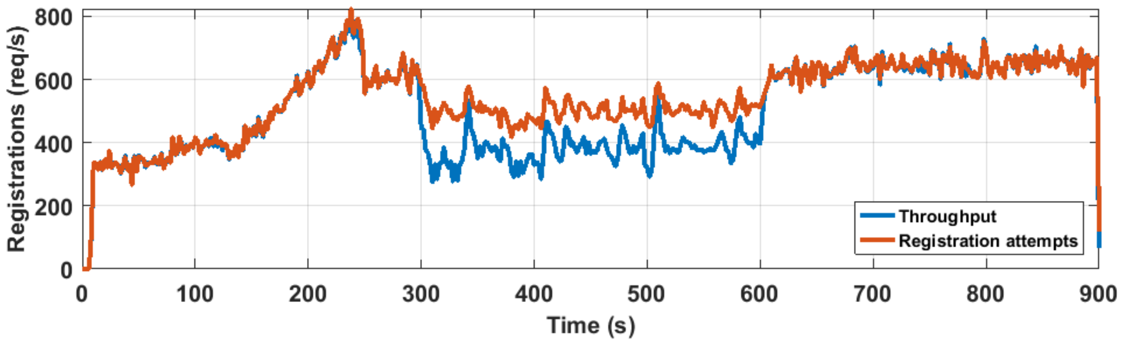


Figure 10: Performance of the IMS registrations during CPU contention (300-600s), using the enhanced feedback control loop.

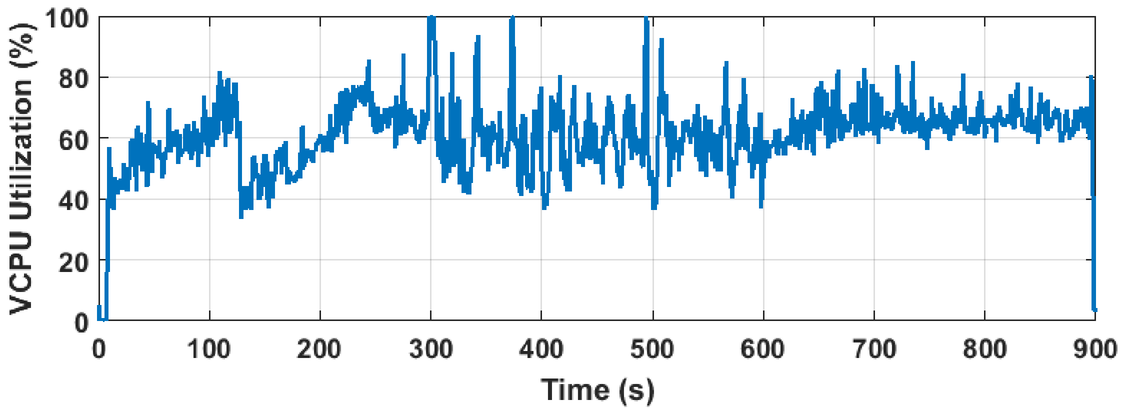
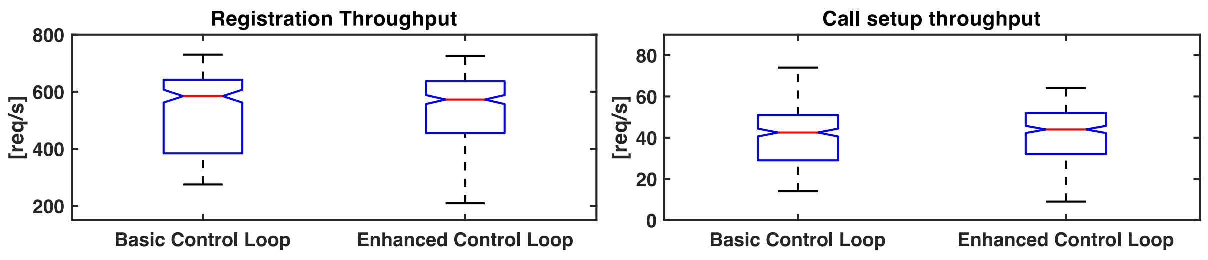


Figure 11: Virtual CPU utilization during CPU contention (300-600s), using the enhanced feedback control enabled.

It is worth noting that the throughput only appears to be higher without our enhanced control. Figure 12 compares the IMS throughput under CPU contention, with the basic and the enhanced feedback control loop. By looking at the throughput of the two approaches, the differences of the throughput are not significant. However, the variance of the enhanced control approach is slightly lower, for both the registration workload (Figure 12a) and call-setup workload (Figure 12b).

A more accurate capacity estimation has also a strong positive impact on the quality of service perceived by the IMS users in terms of *service latency*, which is a key performance indicator considered by SLAs for telecommunication systems. In particular, SLAs typically mandate latency requirements for the average (e.g., the median latency) and the worst cases (e.g., the 90th percentile of latency) [14]. In Figure 13, we compare the CDFs of the latency of the successful registrations, respectively under the basic and the enhanced overload control strategies, during the contention phase of the experiments. The median latency (i.e., the average case, represented by the 50th-percentile of the CDF) is up to 118.6ms for the basic approach. In the worst case,



(a) Registration Throughput

(b) Call-setup Throughput

Figure 12: IMS Registration (12a) and IMS Call-setup (12b) throughput during CPU contention, with the basic and the enhanced feedback control.

represented by the 90th-percentile, the IMS with the basic approach exhibits latencies up to 369.9ms. These latency values are close, and even exceed the SLA objectives typically adopted for IMS systems (e.g., 150ms and 250ms respectively for the 50th and 90th percentiles) [42, 43]. Instead, the proposed approach significantly improves the quality of service, by achieving a service latency up to 28.5ms and 106.2ms respectively for the 50th and 90th percentiles.

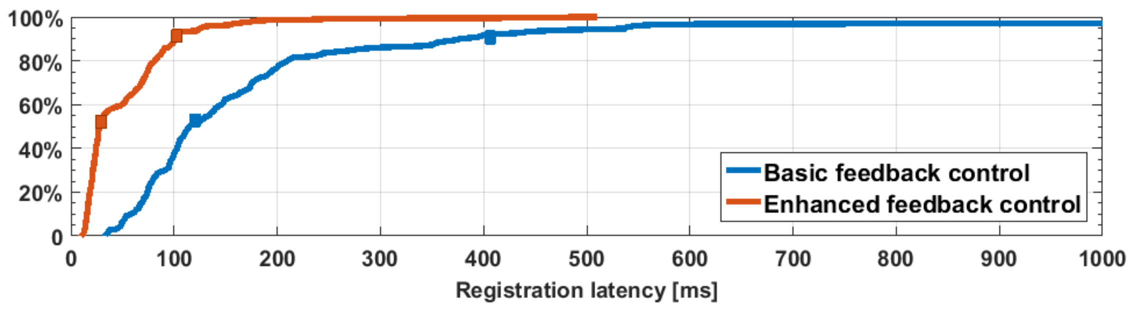


Figure 13: Cumulative distribution of registration latency, with the basic (red line) and the enhanced (blue line) feedback control.

4.3. Performance evaluation under different contention patterns

In the following, we present another group of experiments, to assess the performance of the basic and the enhanced feedback loops in response to different CPU contention patterns. The purpose of this analysis is to identify which scenarios will benefit the most from the proposed solution. We vary the *intensity* of the CPU contention, and the *duration* of CPU contention periods.

- **Intensity**. The intensity of contention is determined by the amount of competing virtual machines that are deployed on the same physical machine. The intensity of contention can impact on the variability of CPU utilization by the virtual machine (e.g., the amplitude of swings in CPU utilization metrics), with side effects on the overload control loops. Therefore, we performed additional experiments where we vary the intensity of contention between 1x (i.e., 50% available CPU time due to the CPU contention with one additional

VM) and 3x (i.e., 25% available CPU time due to the CPU contention with three other VMs).

- **Duration**. The duration of contention is determined by the overlap over time of CPU-bound activities on several virtual machines. If contention periods are long (e.g., due to a configuration error with persistent effects), then the overload control algorithm can eventually converge to a stable condition; instead, if contention periods are short and intermittent (e.g., due to transient high CPU usage by background tasks in the VMs), the overload control algorithm may exhibit unstable behavior and poor performance. Therefore, in addition to the previous experiments (where the CPU contention is constant for a relatively long period), we perform more experiments with short, periodic contention periods, where the periods last respectively for *5s* and *10s*.

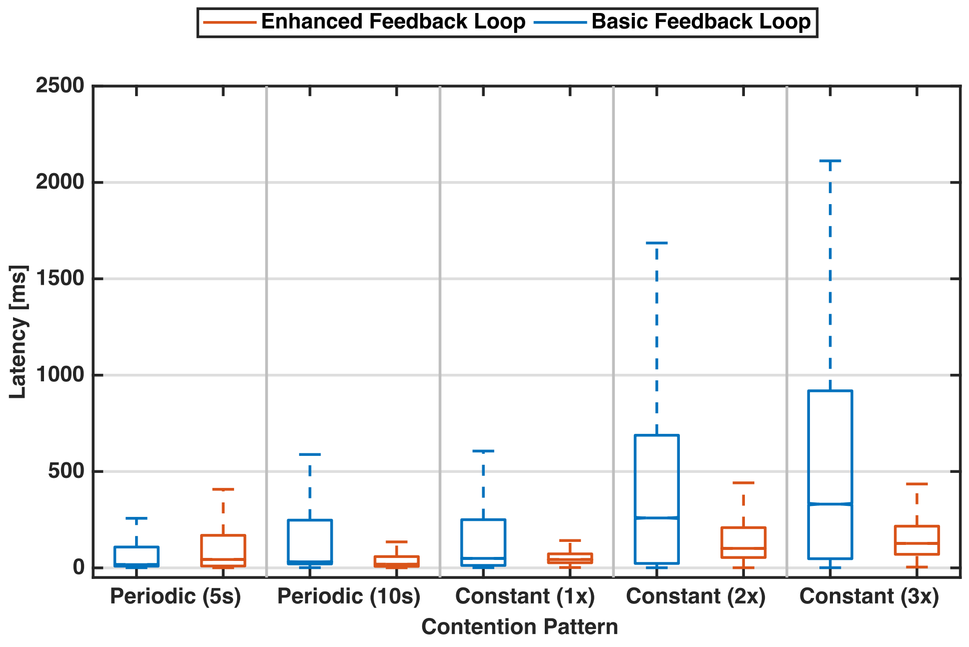
We applied these conditions both on the basic and on the enhanced feedback loop solutions. Each experiment lasts 15 minutes (900s) and it is organized in three phases: during the first 5 minutes we generate a workload up to the engineered level; then, for 5 more minutes, we force physical CPU contention (either periodically or constantly, depending on the *duration* as discussed above), by activating the CPU-bound workload in the additional VMs (between one and three VMs, depending on the *intensity* as discussed above); finally, in the last 5 minutes of the experiment, we simulate the resolution of the CPU contention, by unpinning the virtual CPU of the CPU-bound VMs, as an effect of scaling out or migration of VMs to relieve the contention.

Figure 14 summarizes the performance of the IMS (latency and throughput) during these additional scenarios, with both the basic (blue boxes) and the enhanced (red boxes) feedback loop strategies. When the contention period is very short (5s) there are no significant differences between the two solutions. This scenario represents the most unfavourable condition for our enhanced solution, since the control feedback is based on a sampling window of 5 seconds, and thus the proposed solution is unable to provide any improvement. The percentage of requests violating the latency goal of 250ms is 9.6% for the basic approach and 9.8% with the other. In both cases, the SLA goal of 90%-percentile is not violated.

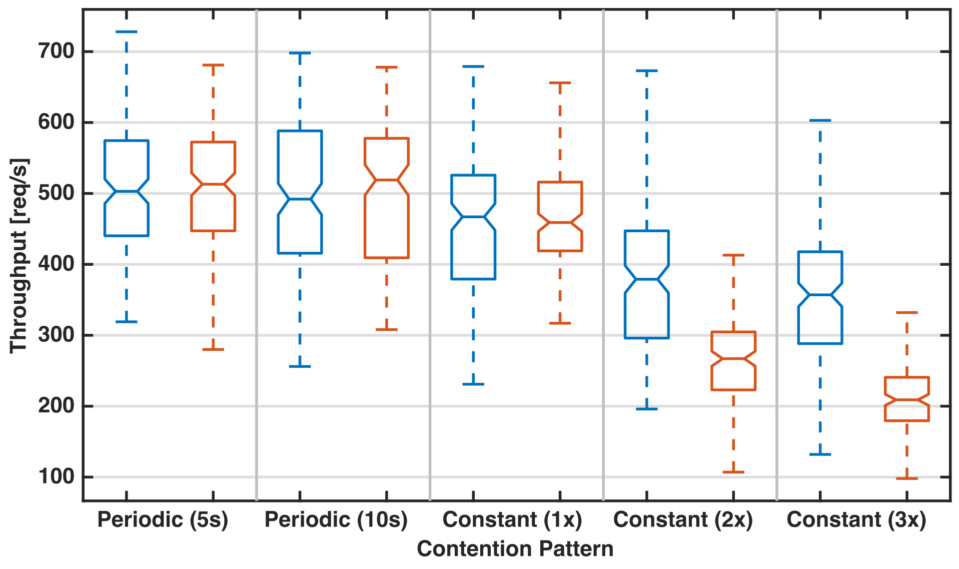
Starting with a period of 10s up to constant patterns, the enhanced feedback loop shows a significant improvement of latency compared to the basic feedback loop. In the case of a periodic contention of *10s*, only the 0.1% of the requests experiences latency higher than 250ms, in contrast to the basic approach in which the 20.1% of the requests were served with a latency higher than the requirement, thus violating the SLA goal.

With a constant contention pattern at 1x intensity, the average available CPU capacity is reduced to 50%, since the CPU is contended between 2 VMs. As discussed in the previous section, there are no significant differences in the IMS throughput between the two approaches, but the proposed approach shows significant reduction in latency: by using the enhanced approach the percentage of requests violating the 90%-percentile requirement decreases from 17.9% to 0.2%.

This behavior is exacerbated by higher contention intensities (2x, 3x). In these cases, the basic overload control solution is unable to accurately estimate the available CPU capacity (eq. (1)), due to the wider swings in CPU utilization metrics; therefore, it degenerates by accepting more requests than the actual capacity of the IMS system. The result is a significant increase



(a) IMS Latency



(b) IMS Throughput

Figure 14: IMS Throughput (14b) and IMS Latency (14a) under different CPU contention patterns, with the basic and the enhanced feedback control.

of IMS latency in the basic feedback loop. With a constant CPU contention at 2x intensity, the available CPU capacity of the VNF is reduced to 33% on average, since the CPU time is contended with 2 additional VMs. In this case, more than the 42% of the requests violates the 90%-percentile latency requirement, in contrast to the 8.1%, when using the enhanced approach. With CPU contention at 3x intensity, with only 25% of the CPU time is available to the VNF. By using the enhanced solution, the number of requests violating the 90%-percentile latency requirement drops from 48% to 9%, thus achieving the SLA requirement.

It is interesting to note that, under the higher intensities of CPU contention, the average throughput with the basic approach is higher than the enhanced approach (e.g., 380 req/s versus 270 req/s in the case of 2x intensity). As discussed in Section 3, the (apparently) better throughput comes at the cost of a poor quality of service (Figure 2), since the IMS is processing a volume of requests which is higher than its capacity. The result is that the IMS takes a long time to serve many of these requests, thus violating the latency requirement. Instead, the enhanced solution only lets in the IMS the subset of requests than can be processed with adequate quality of service: this is a desirable effect of throttling, which is intended to drop the traffic in excess to the system. This result points out that the proposed feedback solution is best suited for those applications (such as the IMS, and NFV in general) where latency and throughput are both important SLA goals. If the proposed solution is not deployed, the throttling mechanism degenerates and lets in the IMS too much traffic, thus favoring throughput at the expense of latency.

4.4. Threats to validity

Internal validity threats concern the relation between theory and observations. As this is an experimental study, there are *measurement errors* (such as the randomness of the workload generator, or perturbations on the network and the hardware). We repeated each experimentation five times, in order to be confident that the effect obtained on the throughput and the latency of the service is not due to other factors out of our control. Moreover, since we collect measurements from repeated experiments on the same system, we need to assure the *independency between consecutive experiments*. Therefore, before every experiment, we restored the same initial condition of the VMs, by reverting them to a fixed snapshot. After restoring the state of VMs, in every experiment we performed a warm-up phase in common with all the experiments. We excluded the performance measures of this phase from the experimental results.

External validity threats concern the possibility to generalize our findings. To perform experiments on a *representative NFV system*, we consider an IMS system, which is a key use case of NFV [44], and adopted an NFV IMS implementation (*Clearwater*) that is backed up by a commercial vendors and is gaining popularity across the open-source community. Moreover, we reproduced typical interactions between end-users and the IMS, by using the well-known SIPp workload generator and traffic scenarios from the company that supports the Clearwater project. In order to get *representative results with respect to the potential CPU contention patterns* that can affect cloud infrastructures, we performed experiments under several different durations and intensities of contention. Finally, in order to get *representative results with respect to the underlying virtualization technology*, we checked that the work-conserving behavior of the hypervisor scheduler is comparable across the main open-source and commercial hypervisors, including Xen, VMware ESXi, and Microsoft Hyper-V [31, 32, 33], and the metrics involved in the overload control solution (such as the CPU utilization and the steal time) are available for all of them. Thus, we can expect that the findings are applicable to these virtualization technologies.

Construct validity threats concern the connection between theory and the experimental evaluation. Therefore, we designed experiments to *assert that the theoretical behavior discussed in the paper matches the actual behavior of the system.* Both the basic and the enhanced approaches are evaluated with the two kinds of overload (i.e., traffic spikes and CPU contention). We demonstrated that the degenerative condition of basic feedback control loop happens only in the case of CPU contention. Moreover, we demonstrated that this degenerative condition can be mitigated by the enhanced approach described in Section 3.

Reliability validity threats concern the possibility of replicating this study. To *ease replications*, we adopted open-source software for the workload generator and the software stack. Moreover, we provided detailed information on the configuration of the hardware, software, and workload, and we suggest several technical suggestions for re-implementing the placeholder process of the proposed solution.

5. Related work

5.1. CPU contention in virtualization infrastructures

CPU contention is a typical problem of virtualization infrastructures and suffered by guest VMs. Nikounia et al. [45] characterized the performance degradation due to resource overcommittment in virtualized environments. Their study identified the CPU resource as the one that impacts the most on service performance during contention with noisy neighbors VMs, and found a major case of execution time slowdown in the hypervisor CPU scheduler. Since this problem is widespread, there have been many studies on ensuring performance isolation at infrastructure level, in order to avoid side-effects from CPU contention. In general, these solutions *prevent or mitigate contention* by enhancing the placement and scheduling of VMs on the physical infrastructure. *Q-Clouds* [46] is a representative solution of this kind, which is a QoS-aware framework aiming to enforce performance isolation by opportunistically provisioning additional resources to alleviate contention. Caglar et al. [47] proposed *HALT*, a performance-interference aware placement strategy based on on-line monitoring and machine learning. To avoid the side effects of the contention for time-sensitive services, HALT adopts a VM migration plan to a different host, based on the learned workload behavior. More recently, in the context of NFV, Kulkarni et al. [48] presented *NFVnice*, a framework to dynamically adjust the scheduling behavior according to the relative priority of the running services and the estimated load. This approach uses *cgroups* to optimize the scheduling behavior and traffic throttling at host level to prevent overloads in the guest.

A drawback of these solutions is that they require *full control of the underlying infrastructure*, since they are meant for system administrators and infrastructure management products. However, in the case of *NFV Infrastructures as a Service* (NFVIaaS), the VNFs do not have control on the underlying infrastructure, where the infrastructure provider may adopt an over-commitment policy that increases the risk of physical CPU contention, at the expense of the VNFs. Another drawback is that, despite the best efforts of system administrators, performance isolation strategies cannot completely prevent contention issues at infrastructure level, due to unexpected maintenance tasks, accidental misconfigurations, or other faults. Thus, services with very high-availability requirements need to include protection mechanisms that mitigate unexpected overloads until its root cause has been solved.

In practice, system administrators adopt CPU consumption metrics to troubleshoot CPU contention issues. At guest level, the *steal time* metric is a well-known indicator of physical CPU contention. This indicator is typically exposed by hypervisors to the guest OSes, as discussed in Section 2. Ayodele et al. [49] demonstrated the impact of the steal time on cloud applications performance under physical CPU contention. Moreover, other studies focus on quantifying the effect of the steal time on CPU time metrics at process- and thread-level [50, 51], provided by the guest OS. This metric is often adopted through heuristics sets by system administrators (e.g., using threshold), for example, by triggering VM migration when the steal time is very high for a prolonged period [52].

However, such heuristics based on the steal time are *unsound*, since a high steal time is not a sufficient condition for a physical CPU contention. For example, a VM voluntary suspended for I/O activity can be subject to high wait time, due to the contention with other VMs that run a CPU-bound workload). In this case, the guest OS metrics will report a lower CPU utilization and low steal time. The best practices from VMware also suggest not to trust CPU consumption metrics provided by the guest OS, since they can be inaccurate in case of physical CPU contention [53] due to time accounting issues. Additionally, even in case of CPU-bound workloads, the steal time can also be inaccurate in case of hyper-threading enabled at host level [54]. Thus, the percentage of steal time is dependent by the workload running in the guest VM. Moreover, a steal time quota can be the consequence of CPU quotas and CPU credits imposed by the infrastructure providers [55].

In this work, we revise the use of CPU consumption metrics in overload control mechanisms inside VNFs. In particular, we discussed why using the steal time metric would not be adequate for self-adaptive overload control solutions, which require to estimate the available CPU capacity in a feedback loop. We proposed a solution that is robust to the inaccuracy and high variability of the steal time, by triggering a low-priority CPU-bound task when the steal time becomes high. If there is no actual physical CPU contention, the mechanism does not cause any problematic effect on the quality of service perceived by users, since it has low priority and it is preempted by the VNF software. Instead, in the case of an actual condition of physical CPU contention, this mechanism can break the degenerative condition caused by the behavior of the hypervisor scheduler, as discussed in Section 2.

5.2. Capacity management and overload control in NFV

Previous research on NFV reliability and performance covered several areas. The majority of the recent research efforts are on the problem of allocating computing resources to VNFs, e.g., by formulating several flavors of *optimization problems* to place VNFs across an infrastructure [56, 57, 58, 59] and to route traffic [60, 61], according to different objective functions and problem constraints (e.g., to take into account resource utilization, performance, reliability, etc.). These approaches forecast the user workloads to plan the allocation of resources, but are not intended for handling sudden overload conditions that deviate from the expected workload. Therefore, these studies are complementary to our work on overload control.

Another branch of research studies has been focused on detecting and diagnosing performance issues in NFV. NFV-VITAL [62] is a solution that can be applied *pre-deployment*, in order to identify and prevent performance bottlenecks and bugs: it is a framework to characterize the performance of VNFs at different scales, by allowing engineers to design performance evaluation experiments (i.e., by varying the flavor of virtual machines, the workload rate, etc.) and to conduct and to report on the experiments. However, NFV-VITAL is not meant to address performance issues after the system has been deployed in operation. Recent frameworks for operational monitoring purposes include anomaly detection systems from Sauvanaud et al. [63] and Cotroneo et al. [64], which analyze metrics at guest- and hypervisor- level through machine learning and statistical techniques to detect performance issues at run-time, and NFVPerf [65], which inspect the VM-to-VM traffic to infer potential bottlenecks affecting the performance of a network function. Once performance issues are detected, this information can be conveyed to NFV orchestration solutions, such as the UNIFY framework [66, 67] to perform load balancing and elastic scaling of VNFs. These solutions are useful to pinpoint the reliability and performance problems, and to reconfigure the VNFs to address them. However, these solutions can take a significant amount of time (up to tens of minutes) to recover from overloads. Therefore, they need to be complemented by mechanisms to protect *in the short-term* the VNFs from the traffic in excess, such as the overload control solutions investigated in this paper.

Traffic throttling is a typical overload control approach used in many IT and telecom systems. A complete survey of these throttling solutions is beyond the scope of this paper; other survey papers, such as the one by Hong et al. [11], present a detailed overview of such overload control schemes. A representative example is represented by the adaptive overload control approach by Welsh et al. [9], which uses a *token bucket* and a closed control loop to dynamically tune the traffic according to the service latency. This approach has also been implemented in the Clearwater NFV IMS that we analyzed in this paper. Other throttling algorithms adopted in the context of carrier-grade telecom switches have been analyzed by Kasera et al. [8]: the *Random Early Discard* (RED [68]) algorithm throttles traffic according to the request queue size, while the *Occupancy* algorithm ensures a target CPU utilization by throttling the traffic according to the CPU utilization and the rate of accepted calls. Traffic throttling mechanisms and algorithms have also been recently ported in the context of NFV systems. In particular, NFV-Throttle [30] provides deployment strategies for both NFVIaaS and VNFaaS, and adopts a combination of closed-loop algorithms at different granularity levels (guest-, host-, and network-level traffic throttling).

A limitation of these existing approaches is that they do not consider the subtleties of overload conditions caused by physical CPU contention, which have different implications than overloads due to workload peaks. Therefore, in this paper, we investigated the interplay between physical CPU contention and overload control algorithms, and we proposed a technique for making overload control algorithms applicable for the case of physical CPU contention.

6. Conclusion

In this paper, we analyzed the problem of overload conditions caused by physical CPU contention. We pointed out that this form of overload conditions have a different behavior (e.g., in terms of CPU utilization metrics) than the case of traffic spikes; and that the overload control solutions for traffic spikes can be ineffective, or even counterproductive, in the case of physical CPU contention.

Therefore, we extended the existing feedback control-based approach to also address physical CPU contention. A key requirement of this solution has been to support VNFs deployed on IaaS, where the VNF has little visibility or control of the underlying infrastructure. Our solution introduces a mechanism inside the VNF to occupy the CPU cycles freed by traffic throttling, in order to protect the feedback control loop from the opportunistic behavior of the hypervisor that may reclaim the CPU cycles. Moreover, we discussed CPU utilization metrics available in IaaS infrastructures, and how these metrics should be interpreted in order to deal with physical CPU contention. Our experiments on an NFV IMS system confirmed that the proposed solution can avoid interferences between resource hogs and the overload control mechanisms.

Acknowledgments

This work has been supported by Huawei Technologies Co., Ltd.

References

[1] E. Bauer, R. Adams, Reliability and Availability of Cloud Computing, 1st Edition, Wiley-IEEE Press, 2012.

[2] L. Wang, R. A. Hosn, C. Tang, Remediating Overload in Over-Subscribed Computing Environments, in: 5th Intl. Conf. Cloud Computing (CLOUD), 2012.

[3] S. A. Baset, L. Wang, C. Tang, Towards an Understanding of Oversubscription in Cloud, in: USENIX Wksp. on Hot Topics in Mgmt. of Internet, Cloud, and Enterprise Netw. and Serv. (Hot-ICE), 2012.

[4] Amazon.com, Inc., Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region (2011).
URL http://aws.amazon.com/message/65648/

[5] A. Warren, What Happened to Google Docs on Wednesday (2011).
URL https://googleenterprise.blogspot.com/2011/09/what-happened-wednesday.html

[6] X. Wang, X. Fu, X. Liu, Z. Gu, Power-aware cpu utilization control for distributed real-time systems (2009) 233–242doi:10.1109/RTAS.2009.12.

[7] L. Tomás, C. Klein, J. Tordsson, F. Hernández-Rodríguez, The straw that broke the camel's back: Safe cloud overbooking with application brownout (2014) 151–160doi:10.1109/ICCAC.2014.10.

[8] S. Kasera, J. Pinheiro, C. Loader, M. Karaul, A. Hari, T. LaPorta, Fast and robust signaling overload control, in: IEEE Intl. Conf. Network Protocols, 2001.

[9] M. Welsh, D. E. Culler, Adaptive Overload Control for Busy Internet Servers, in: USENIX Symp. on Internet Technologies and Systems, 2003.

[10] J. L. Hellerstein, Y. Diao, S. Parekh, D. M. Tilbury, Feedback control of computing systems, John Wiley & Sons, 2004.

[11] Y. Hong, C. Huang, J. Yan, A comparative study of SIP overload control algorithms, Network and traffic engineering in emerging distributed computing applications.

[12] G. Galante, L. de Bona, A Survey on Cloud Computing Elasticity, in: IEEE 5th Intl. Conf. Utility and Cloud Computing (UCC), 2012.

[13] P. C. Brebner, Is your Cloud Elastic Enough?: Performance Modelling the Elasticity of Infrastructure as a Service (IaaS) Cloud Applications, in: 3rd ACM/SPEC Intl. Conf. Performance Engineering, 2012.

[14] Quality Excellence for Suppliers of Telecommunications Forum (QuEST Forum), TL 9000 Quality Management System Measurements Handbook 4.5, Tech. rep. (2010).

[15] R. L. Freeman, Telecommunication system engineering, Vol. 82, John Wiley & Sons, 2015.

[16] E. Stahl, A. Corona, F. De Gilio, et al., Performance and Capacity Themes for Cloud Computing, IBM Redbooks, 2013.

[17] Red Hat Inc., Virtualization deployment and administration guide (2014).
URL `https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_deployment_and_administration_guide/index`

[18] C. Ehrhardt, CPU time accounting (2010).
URL `http://public.dhe.ibm.com/software/dw/linux390/perf/CPU_time_accounting.pdf`

[19] R. van der Heij, Understanding Linux on z/VM steal time (2014).
URL `http://www.velocitysoftware.com/VMSTEAL.PDF`

[20] VMware Inc., Guest and HA Application Monitoring Developer's Guide (2016).
URL `https://pubs.vmware.com/vsphere-6-0/topic/com.vmware.ICbase/PDF/vs600_guest_HAappmon_sdk.pdf`

[21] A. Lê-Quôc, M. Fiedler, C. Cabanilla, The top 5 AWS EC2 performance problems (2013).
URL `https://www.datadoghq.com/blog/top-5-ways-to-improve-your-aws-ec2-performance/`

[22] N. Shestakov, Hyper-V performance analysis with Veeam ONE (2016).
URL `https://hyperv.veeam.com/blog/how-to-analyse-visualise-hyper-v-performance/`

[23] V. Hilt, I. Widjaja, Controlling Overload in Networks of SIP Servers, in: IEEE Intl. Conf. Network Protocols, 2008.

[24] S. Chandra, P. M. Chen, Whither generic recovery from application faults? a fault study using open-source software, in: IEEE Intl. Conf. Dependable Systems and Networks, 2000.

[25] M. Grottke, D. S. Kim, R. Mansharamani, M. Nambiar, R. Natella, K. S. Trivedi, Recovery From Software Failures Caused by Mandelbugs, IEEE Transactions on Reliability 65 (1) (2016) 70–87.

[26] C. Shen, H. Schulzrinne, E. M. Nahum, Session Initiation Protocol (SIP) server overload control: Design and evaluation, IPTComm 8 (2008) 149–173.

[27] J. Mo, R. J. La, V. Anantharam, J. Walrand, Analysis and comparison of TCP Reno and Vegas, in: Annual Joint Conference IEEE Computer and Communications Societies (INFOCOMM), 1999.

[28] K. Ramakrishnan, R. Jain, A binary feedback scheme for congestion avoidance in computer networks with a connectionless network layer, in: ACM SIGCOMM Computer Communication Review, Vol. 18, 1988, pp. 303–313.

[29] Y. Xia, L. Subramanian, I. Stoica, S. Kalyanaraman, One more bit is enough, ACM SIGCOMM Computer Communication Review 35 (4) (2005) 37–48.

[30] D. Cotroneo, R. Natella, S. Rosiello, NFV-Throttle: An overload control framework for Network Function Virtualization, IEEE Transactions on Network and Service Management.

[31] L. Eggert, J. D. Touch, Idletime scheduling with preemption intervals, in: ACM SIGOPS Operating Systems Review, Vol. 39, 2005, pp. 249–262.

[32] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, A. Fedorova, The Linux scheduler: A decade of wasted cores, in: 11th ACM European Conference on Computer Systems, 2016.

[33] Xen Project, Credit scheduler (2017).
URL `https://wiki.xen.org/wiki/Credit_Scheduler`

[34] J. Corbet, A safe SCHED_IDLE implementation (2002).
URL `https://lwn.net/Articles/4073/`

[35] Windows Dev Center, Task Idle Conditions (2017).
URL `https://msdn.microsoft.com/en-us/library/windows/desktop/aa383561(v=vs.85).aspx`

[36] L. Torvalds et al., Linux kernel parameters (2007).
URL `https://www.kernel.org/doc/Documentation/admin-guide/kernel-parameters.txt`

[37] Project Clearwater.
URL `http://www.projectclearwater.org/`

[38] R. Gayraud, O. Jaques, et al., SIPp: SIP load generator (2010).
URL `http://sipp.sourceforge.net/`

[39] Metaswitch Networks Ltd., Clearwater SIP Stress scenario (2016).
URL `https://github.com/Metaswitch/sprout/blob/dev/clearwater-sip-stress.root/usr/share/clearwater/sip-stress/sip-stress.xml`

[40] J. Heo, L. Singaravelu, Deploying extremely latencysensitive applications in vsphere 5.5: Performance study, VMware, Inc., Tech. Rep.

[41] D. M. Davis, Demystifying cpu ready (%rdy) as a performance metric: Don't trust available cpu, Quest Software, White paper.

[42] D. Cotroneo, L. D. Simone, A. K. Iannillo, A. Lanzaro, R. Natella, Dependability evaluation and benchmarking of Network Function Virtualization Infrastructures, in: IEEE Intl. Conf. Network Softwarization (NetSoft), 2015.

[43] D. Cotroneo, L. De Simone, R. Natella, NFV-Bench: A dependability benchmark for Network Function Virtualization systems, IEEE Transactions on Network and Service Management.

[44] ETSI Industry Specification Group, NFV Use Cases.

[45] S. H. Nikounia, et al., Hypervisor and neighbors' noise: Performance degradation in virtualized environments, IEEE Transactions on Services Computing.

[46] R. Nathuji, A. Kansal, A. Ghaffarkhah, Q-clouds: managing performance interference effects for qos-aware clouds, in: Proceedings of the 5th European conference on Computer systems, ACM, 2010, pp. 237–250.

[47] F. Caglar, S. Shekhar, A. Gokhale, A performance interference-aware virtual machine placement strategy for supporting soft realtime applications in the cloud, Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, USA, Tech. Rep. ISIS-13-105.

[48] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, T. Wood, M. Arumaithurai, X. Fu, Nfvnice: Dynamic backpressure and scheduling for nfv service chains, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, ACM, 2017, pp. 71–84.

[49] A. O. Ayodele, J. Rao, T. E. Boult, Performance measurement and interference profiling in multi-tenant clouds, in: Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on, IEEE, 2015, pp. 941–949.

[50] P. Hofer, F. Hörschläger, H. Mössenböck, Sampling-based steal time accounting under hardware virtualization, in: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, ACM, 2015, pp. 87–90.

[51] M. Yamamoto, K. Kohta Nakashima, Execution time compensation for cloud applications by subtracting steal time based on host-level sampling, in: Companion Publication for ACM/SPEC on International Conference on Performance Engineering, ACM, 2016, pp. 69–73.

[52] S. Team, Understanding CPU Steal Time - when should you be worried? (2013).
URL `http://blog.scoutapp.com/articles/2013/07/25/understanding-cpu-steal-time-when\-should-you-be-worried`

[53] VMware, Inc., Timekeeping in VMware virtual machines (2008).
URL `http://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/Timekeeping-In-VirtualMachines.pdf`

[54] G. Casale, C. Ragusa, P. Parpas, A feasibility study of host-level contention detection by guest virtual machines, in: Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on, Vol. 2, IEEE, 2013, pp. 152–157.

[55] P. Leitner, J. Scheuner, Bursting with possibilities–an empirical study of credit-based bursting cloud instance types, in: Utility and Cloud Computing (UCC), 2015 IEEE/ACM 8th International Conference on, IEEE, 2015, pp. 227–236.

[56] H. Moens, F. De Turck, VNF-P: A model for efficient placement of virtualized network functions, in: IEEE Intl. Conf. Network and Service Management (CNSM), 2014.

[57] S. Clayman, E. Maini, A. Galis, A. Manzalini, N. Mazzocca, The dynamic placement of virtual network functions, in: IEEE Network Operations and Management Symp. (NOMS), 2014.

[58] B. Németh, J. Czentye, G. Vaszkun, L. Csikor, B. Sonkoly, Customizable real-time service graph mapping algorithm in carrier grade networks, in: IEEE Intl. Conf. Network Function Virtualization and Software Defined Networks (NFV-SDN), 2015.

[59] F. Carpio, S. Dhahri, A. Jukan, VNF placement with replication for load balancing in NFV networks, arXiv preprint arXiv:1610.08266.

[60] B. Addis, D. Belabed, M. Bouet, S. Secci, Virtual network functions placement and routing optimization, in: IEEE Intl. Conf. Cloud Networking (CloudNet), 2015.

[61] J. Elias, F. Martignon, S. Paris, J. Wang, Efficient orchestration mechanisms for congestion mitigation in NFV:

Models and algorithms, IEEE Transactions on Services Computing.

[62] L. Cao, P. Sharma, S. Fahmy, V. Saxena, NFV-VITAL: A framework for characterizing the performance of virtual network functions, in: IEEE Intl. Conf. Network Function Virtualization and Software Defined Networks (NFV-SDN), 2015.

[63] C. Sauvanaud, K. Lazri, M. Kaaniche, K. Kanoun, Anomaly Detection and Root Cause Localization in Virtual Network Functions, in: IEEE Intl. Symp. Software Reliability Engineering, 2016.

[64] D. Cotroneo, R. Natella, S. Rosiello, A fault correlation approach to detect performance anomalies in virtual network function chains, in: IEEE Intl. Symp. Software Reliability Engineering, 2017.

[65] P. Naik, D. K. Shaw, NFVPerf: Online Performance Monitoring and Bottleneck Detection for NFV, in: IEEE Intl. Conf. Network Softwarization (NetSoft), 2016.

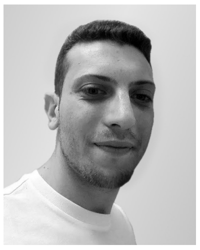
[66] R. Szabo, M. Kind, F.-J. Westphal, H. Woesner, D. Jocha, A. Csaszar, Elastic network functions: opportunities and challenges, IEEE Network 29 (3) (2015) 15–21.

[67] S. Van Rossem, W. Tavernier, B. Sonkoly, D. Colle, J. Czentye, M. Pickavet, P. Demeester, Deploying elastic routing capability in an SDN/NFV-enabled environment, in: IEEE Intl. Conf. Network Function Virtualization and Software Defined Networks (NFV-SDN), 2015.

[68] S. Floyd, V. Jacobson, Random early detection gateways for congestion avoidance, IEEE/ACM Transactions on Networking 1 (4) (1993) 397–413.

Domenico Cotroneo (Ph.D.) is associate professor at the Federico II University of Naples. His main interests include software fault injection, dependability assessment, and field measurement techniques. He has been member of the steering committee and general chair of the IEEE Intl. Symp. on Software Reliability Engineering (ISSRE), PC co-chair of the 46th IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN), and PC member for several scientific conferences on dependable computing including SRDS, EDCC, PRDC, LADC, SafeComp.

Roberto Natella (Ph.D.) is a postdoctoral researcher at the Federico II University of Naples, Italy, and co-founder of the Critiware s.r.l. spin-off company. His research interests include dependability benchmarking, software fault injection, and software aging and rejuvenation, and their application in operating systems and virtualization technologies. He has been involved in projects with Leonardo-Finmeccanica, CRITICAL Software, and Huawei Technologies. He contributed, as author and reviewer, to several leading journals and conferences on dependable computing and software engineering, and he has been organizing the workshop on software certification (WoSoCer) within the IEEE ISSRE conference.



Stefano Rosiello (Ph.D.) is a postdoctoral researcher at University of Naples Federico II, Italy within the Dependable Systems and Software Engineering Research Team (DESSERT) group. His main research activity focuses on overload control in carrier-grade network function virtualization and cloud infrastructures. His research interests also include experimental reliability evaluation, dependability benchmarking and fault injection testing.