Sidecar based resource estimation method for virtualized environments

Csaba Simon[1], Markosz Maliosz[2], Miklós Máté[3], Dávid Balla[4], and Kristóf Torma[5]

***Abstract*—The widespread use of virtualization technologies in telecommunication system resulted in series of benefits, as flexibility, agility and increased resource usage efficiency. Nevertheless, the use of Virtualized Network Functions (VNF) in virtualized modules (e.g., containers, virtual machines) also means that some legacy mechanisms that are crucial for a telco grade operation are no longer efficient. Specifically, the monitoring of the resource sets (e.g., CPU power, memory capacity) allocated to VNFs cannot rely anymore on the methods developed for earlier deployment scenarios. Even the recent monitoring solutions designed for cloud environments is rendered useless if the VNF vendor and the telco solution supplier has to deploy its product into a virtualized environment, since it does not have access to the host level monitoring tools. In this paper we propose a sidecar-based solution to evaluate the resources available for a virtualized process. We evaluated the accuracy of our proposal in a proof of concept deployment, using KVM, Docker and Kubernetes virtualization technologies, respectively. We show that our proposal can provide real monitoring data and discuss its applicability.**

***Index Terms*—Computer network management, Network function virtualization.**

I. INTRODUCTION

MODERN, high performance telecommunication software is implemented as a collection of stateless microservices for maximum scalability and fault-tolerance. These microservices have so far been running in controlled environments with known performance characteristics. In the near future, however, these systems must be able to work in any environment, even in heterogeneous ones, and ones with volatile resource availability [1]. Moreover, in a virtualized environment the available resources reported by the system may not accurately reflect the amount of resources that are physically available. Therefore, if the telecommunication systems want to perform load balancing, autoscaling or overload prediction, these applications need to measure their own performance, report it to the framework to provide sufficient information to deduce the available resources.

Porting such measurement tasks onto stateless microservice applications is challenging, since new resource monitoring approach should be applied in order to circumvent the resource estimation ambiguity. In this paper we examined the feasibility of using a separate measurement application for the estimation of the available resources. This measurement application runs in a container or a virtual machine separate from the main telecommunication application. This configuration is called "sidecar" to reflect on the similarities with attaching a sidecar to a motorbike and is a well-known usage pattern in virtualized computing systems [2].

The main goal of this paper is to validate the feasibility of performance measurements from a sidecar. In this paper we focus on telecommunication (telco) applications that, compared to generic webservices, must fulfill much stricter Service Level Agreements, and they are much vulnerable to insufficient (or less than agreed) resource sets. Therefore a correct evaluation of the resources available for a given telco app is crucial to operate within the agreed parameters. In principle, increasing resource usage by the telco application results in degraded computing performance in the sidecar, but the sensitivity and the accuracy of this method were previously unknown. In order to eliminate the dependency on (potentially) bogus CPU usage resource reporting available from inside a virtualized space, we monitored the completion time of a reference task as the main indicator of the computing performance of the underlying infrastructure.

In the next Section we present the technologies used in the investigated virtualized environments, present a problem statement and a literature survey. In Section III we introduce our proposal and present a proof of concept deployment of our proposal, based on which we present a detailed measurement-based evaluation of it. In Section IV we discuss the possible limitations and the applicability of our proposal and finally we conclude our work.

II. RELATED WORK

In this section we present the virtualization aspects of the infrastructure that are relevant to our work first. To the best of our knowledge, our approach described in this paper was not published before. Still, the wider topic of performance monitoring aspects of virtualized applications has been intensively investigated in the last decade and has a vast literature. In the related work part of this section we present the typical approaches to mitigate the performance monitoring problem of telecommunication systems deploying Virtualized Network Functions (VNFs). We also present a set of works that inspired us to use service completion times to characterize the resource set available to an application.

[1,2,3,4,5]The authors are with HSNLab, Dept. of Telecommunications and Media Informatics, Budapest University of Technology and Economics, Budapest, Hungary. E-mail: {simon, maliosz, mate, balla, torma}@tmit.bme.hu.
[4,5] BME Balatonfüred Student Research Group, Hungary

DOI: 10.36244/ICJ.2020.2.1

A. Virtualization technologies

Virtualization is a technology that introduces a layer of abstraction between computing, storage and networking hardware, and the applications running on it. Thus, the underlying physical resources (CPU, memory, disk and network) are shared, and there can be multiple systems (or virtual machines - VMs) running simultaneously and concurrently on the same host. There are several approaches to implement virtualization, but in modern cloud systems there are two alternatives that are used: the host-based and the operating system level virtualizations.

The Kernel-based Virtual Machine (KVM) is a hypervisor module of the Linux kernel [3]. It allows running guest operating systems in a virtualized environment. The KVM kernel module is only a hypervisor, the virtual devices, networking etc. must be supplied to the VM by the virtualization program, and the most widely known one is QEMU [4]. QEMU implements CPU emulation in software, but its qemu-kvm extension uses KVM instead of its soft-cpu implementation. Finally, we may use *libvirt* library [5] manage VMs, including *cgroup* policy groups for resource policy control [6]. Since *cgroups* is a powerful and important mechanism used by us also for both VM and container resource control, we describe it in detail in the following section.

The operating system level virtualization, also known as containerization, does not virtualize the host hardware as other types do. Instead, it virtualizes the kernel of the host. Opposed to the host-based virtualization, the containers do not need a hypervisor, instead they run directly within the host machine's kernel. The isolation and resource control tasks are assured by the namespaces [7] and control group (*cgroup*) [6] mechanisms of the kernel, respectively. The most well-known container technology is Docker [8]. An important technology within the container ecosystem is Kubernetes [9], a container management framework. Kubernetes extends the process-oriented approach of Docker and focuses on services instead. In Kubernetes, the service is implemented by a set of connected containers, called pods. In Kubernetes, the pods are the basic unit of scaling, and per-pod resource usage pattern can be specified.

The resource usage of a Linux system by default is governed by *cgroups*. The CPU scheduler of Linux shares the CPU time among the process groups according to their *cpu.shares* value; the default value is 1024. E.g., if there is one CPU, and two groups want to use it fully, by default they both get 50% share of the CPU. If we change the shares of one group to 512, that group will receive 33%, and the other will receive 66%. This division happens hierarchically: the sub-groups receive the CPU percentage of their parent group. When Docker is active on the host, it inserts its own slice, named *docker*. Similarly, QEMU based VMs get their own top-level slice, called machine-slice. As a consequence, Docker containers and KVM/QEMU VMs are handled in isolated resource buckets (cgroup slices) by the host-level cgroups scheduler. Kubernetes has its own mechanism that configures the resource reservation quotas of the containers started in its pods [10]. In our paper we use the so called burstable mechanism, where each container specifies its resource usage intent (*request*) but lets the Kubernetes framework to scale the resources according to the total available set. Then Kubernetes makes sure that the allocated resources to different containers keep the ratio of the declared *requests*.

B. Related work

The authors publishing in this field mostly focused their efforts on providing a working solution to address the monitoring needs of the cloud native telecom systems that emerged since the beginning of the 2010s. As part of these efforts, several solutions were proposed to provide accurate resource usage in cloud native telecom systems. Paper [1] introduces a complete monitoring framework for cloud native 5G systems. Still, it considers that the access to the physical node metrics is granted.

A more academic approach is followed in [11], where realtime prediction and long term forecasting is used to support the autoscaling process for container-based telecom microservices. The authors exploit the specific nature of typical telecom services due to the repetitive nature of human behavior. Still, this approach relies on generic Kubernetes monitoring technologies and the author's custom monitoring container, if they have access to the real performance data from the underlying host system.

Several works analyze the statistical characteristics of the observed resource usage parameters for the VNFs and infer the availability and sufficiency of the resources in the system based on these. A good example of these works is [12], where the skewness of the probability distribution of per VNF CPU usage is used as an indicator of system-wide resource availability. The authors show that their proposal can be used to provide automatic notifications in case of system overload. Nevertheless, this approach also requires the access of host level information or Docker API at the host.

The above cited articles [1][11][12] are representative for the prior work in this field. Due to lack of space we do not offer further insight into other proposals, but the interested reader is referred to the related work sections of these papers in order to get a wider knowledge of the state of art in this area. Our solution will differ from these, since our novel approach avoids any use of any information that may be obtained from the host.

As already described above, the resource monitoring approaches observed several parameters when tried to model the available resource sets, not only the CPU usage. This gave us the idea to verify if exists such a parameter that can be measured from inside the virtualized space and is a good indicator of the available resources (e.g., CPU power). Based on our literature survey we have seen that the service completion times, the resources consumption (i.e., the allocated resources, if the service uses all available resources) and the user demands are strongly dependent on each other. We found that relevant works were published since the mid-2000s and mostly relate to the field of BigData. A good

introduction of this approach is found in [13], where the authors measured both the response times of classical industrial IT applications and the CPU utilization, and used it to estimate the volumes of user demands. The approach of measuring the service completion time later was used in paper [14] to offer an accurate scheduling mechanism, where based on demand (i.e., job size) and resource availability (number of parallel worker instances) a certain completion time can be guaranteed.

In our scenarios user demand can be easily obtained, either by the framework itself or by the application by monitoring the incoming request rate. The service completion time can be measured from inside the virtualized space. Thus, based on [13][14] we supposed that observing these two parameters, we can provide a good estimation of the compute resources, and we proposed a method, which is introduced in the next section.

III. SIDECAR BASED RESOURCE ESTIMATION METHOD AND PROOF OF CONCEPT

A. Sidecar based resource estimation

As described in the previous section, we propose to evaluate the resource usage of a virtualized function (or application) by observing the duration of an application. In practice there is a large variety of VNFs in a telecom system, and each of these VNFs have their own resource usage characteristics, which also depend on the current load. Therefore, the measurement of the VNF is not useful for this role. Before using the measured response time of a VNF to evaluate the resources it used during the observation period, a detailed profiling of the VNF would be needed. Even if this is doable, as VNF vendors may be required to do this profiling before shipping their product, the management of release schedule and continuous update of this data in a large telecommunication system is not practical.

As an alternative we propose to use the *same* application for *every* VNF and use *this* application as a benchmark. This application should be selected such as it correlates with the resource set allocated to it and it has a stable performance.

We propose to deploy this monitoring application as a sidecar together with all the VNFs that require resource estimation. This sidecar should run in the same virtualized environment, as the "target" VNF. In the case of VMs or Docker containers both the monitoring sidecar application and the target VNF should run on the same machine, with further conditions detailed in Section IV. In the case of Kubernetes based deployment, the monitoring sidecar application and the target VNF should be deployed within the same pod.

B. Load emulation

In our work we used the *stress-ng* utility [15] to generate load on the CPU. It is a flexible utility capable of running several different stressor routines in any number of parallel processes. Therefore, we considered to be versatile enough to model a generic VNF during our evaluations. It was not designed to be a benchmark, but we judged that its metrics (called bogo operations/sec, referred to as *bogo ops*) are sufficiently accurate for our purposes. Thus, we used the same tool for both generating load (*gen*) and serving as a monitoring probe (*mon*).

We mainly used the *cpu* stressor, which contains more than 70 different stressor algorithms, and the default setting is to loop over all of them repeatedly. These algorithms perform different numeric computations, and together they stress of the various arithmetic units of the CPU. Nevertheless, we also tested the memory stressor, and two stressors using system-calls (executing timer calls and pipe operations).

Stress-ng can print the number of iterations it ran within the specified time limit with the option *--metrics-brief*. It cannot report per-process results, just the total for all the stressor processes of the same type. For continuous monitoring of the performance of stress-ng it must be run in an endless loop with short timeout of 20 s. This reporting period is much longer than the measurement periods typical for monitoring systems in production (1 s), but in our evaluation let stress-ng perform several hundred iterations in all scenarios to minimize quantization errors. In a real-life scenario, running VNFs under heavy load, a 1 s measurement period would lead to similar accuracy. The overhead of restarting stress-ng is negligible.

Based on extensive tests we decided to configure four stress-ng stressors during the tests. For both the *gen* and *mon* roles, we run the following operations to generate their load:

- CPU – integer and floating-point mathematical operations run in user mode
- Memory – *mmap()/munmap()* calls with 256 MB data
- Timer – sets one million timers each second, and counts how many of them are completed successfully
- Pipe – moving data through Linux pipes. The size of the pipe is 512 MB, and the data size is 4 KB (equals the memory page size).

The detailed parameter setup is shown in Table I. It can be seen that the parameters, and implicitly the load of the *mon* process is independent of the monitored *gen* process. Thus the cost of our solution is constant. In a real life deployment scenario the load level can be adjusted to the available resources.

TABLE I
THE PARAMETERS OF THE STRESS TOOL USED TO TEST THE SIDECAR SCENARIO

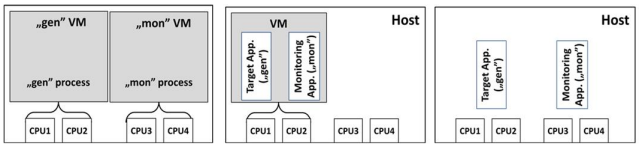
Stressor type	"gen" process	"mon" process
CPU	--cpu 1	--cpu 1
Virtual Memory	--vm 1	--vm 1 –vm-bytes 20
Timer	--timer 1	--timer 1
Pipe	--pipe 1	--pipe 1

C. Configuration of the virtual environments

During our measurements we used both KVM/QEMU VMs and Docker containers. The VMs used in our tests were provisioned with Vagrant, and depending on the scenario, we run a single VM or two VMs. When two VMs were provisioned, one VM acted as the target application, generating the load to be monitored (*gen*). The other VM acted as the monitoring VM (*mon*). We allocated two CPU cores

and 1 GB RAM for each. When a single VM was used (e.g., in Section V.A), only one of the VMs was started. When the stress-ng process was containerized (e.g., in Sections IV.A and IV.B), we used our custom Docker image, created from Ubuntu 18.04.1 LTS, and installed a stress-ng v.0.09.25. The Docker container was run with no resource limits.

Depending on the measurement setup, we had four arrangements. In the first one we had two VMs and in each VM we run a stress-ng process, as shown in Fig. 1 a) and the measurements on this setup are discussed in Section IV.A.



(a) (b) (c)

Fig. 1. Sidecar scenarios with a) two VMs, b) two containers run in single VMs, and c) with two containers run on the host, respectively.

Note that the pinning of VMs might differ from the one illustrated in Fig. 1 a), according to the details given in Section IV.A. The parameters of these two stress-ng processes were the ones already shown in Table I.

A second measurement setup used only one VM, both the *gen* and *mon* processes were containerized, and these two containers were run within the VM. This setup is shown in Fig. 1 b) and is discussed in Section IV.B. A third measurement setup without VMs used only Docker containers, where the *gen* and *mon* containers were run on the host. The containers shared all the resources of the hosts and this setup is illustrated in Fig. 1 c) and is discussed in Section IV.B. Finally, we had a fourth measurement setup, where two containers were run in a single pod. The measurements with this setup are discussed in Section IV.C.

We run our test on desktop PCs, the detailed hardware specification is shown in Table II.

TABLE II
THE HARDWARE USED DURING TO EVALUATE THE SCENARIOS

Name	CPU type	Frequency [GHz]	RAM [GBytes]
PC1	Intel Core i5-2400	3,1	8 (DDR3)
PC2	Intel Core2 Quad Q6600	2,4	6 (DDR2)
PC3	AMD Athlon 64 X2 5050e	2,6	6 (DDR2)
PC4	Core i5-3320M	2,6	8 (DDR3)

IV. EVALUATION OF THE PROPOSAL

In this section we run three set of experiments to evaluate our proposal from III.A in the test environment described in the previous section.

A. VM based deployments

The first sets of experiments were conducted with VM based deployments. The measurement setup is illustrated in Fig. 1 a), where machine *mon* is the sidecar VM that monitors its own performance, and tries to deduce the resources used by the *gen* process from the other VM, based on its own performance.

We limited the CPU usage of *stress-ng* with *cgroups* policies applied to the processes representing QEMU's virtual CPUs on the host. We used the *cpuset cgroup* to pin the vCPUs to specific physical CPUs, and the *cpu cgroup*'s *cfs_quota_ms* parameter to impose a quota on per-VM level. Each presented measurement point is the aggregation of 10 experiments.

When each VM only have 1 vCPU allocated, it can be the same cores for both VMs, or different. Fig. 2 shows the performance of *mon* when it shares a single CPU with *gen*. The different colors correspond to different loads on *gen*. When there is light load on *gen*, the measured performance of *mon* VM correlates with the load of *gen*. But when *gen* is at least 50% loaded, the performance of *mon* is independent of the load, this setup is thus not suitable for detecting overload on the telecom application.

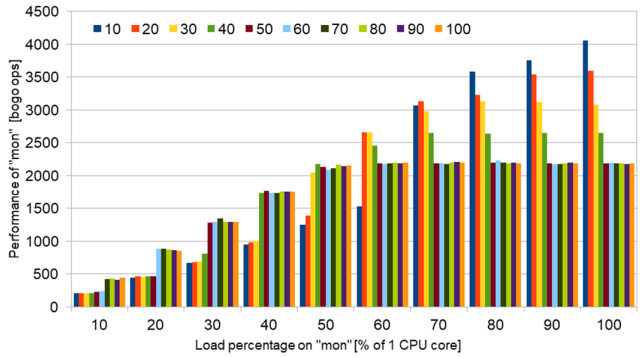


Fig. 2. Performance measured in bogo ops, when "gen" and "mon" share a single CPU. The colored bars correspond to different loads on "gen", expressed as % of 1 CPU core capacity.

When each VM only have 1 vCPU allocated, but they are mapped the different physical CPU cores, the performance figure differs from the previous case, as shown in Fig. 3. In this case when *gen* is getting close to the maximum load, the performance of *mon* gets a noticeable bump. Note however, that this bump starts at around 70% percent load on *gen*, which is still quite far from its maximum capacity. Another problem with this setup is that we are loading only 1+1 cores of a 4-core CPU; thus, the performance bump of *mon* comes from the raised CPU frequencies under heavy load. In a real deployment the applications usually try to put load on all available CPU cores, resulting in different performance profiles.

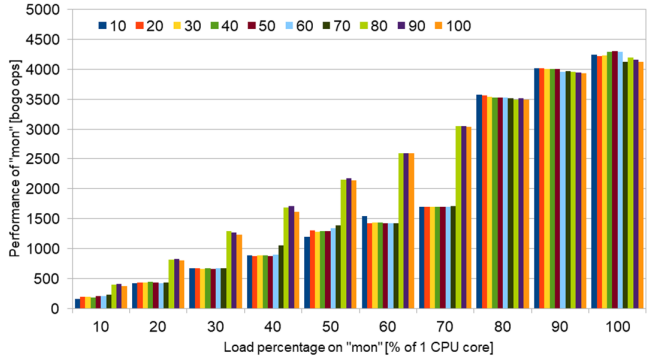


Fig. 3. Performance measured in bogo ops, when "gen" and "mon" run on different CPU cores. The colored bars correspond to different loads on "gen", expressed as % of 1 CPU core capacity.

When in our 4 core CPU host machines two vCPUs are allocated to both VMs, the CPU cores assigned to the VMs can be all different, only one shared, or both shared between the two VMs. The figures for the "all different" and the "all shared" CPU core scenarios look identical to the results shown in Fig. 2 and Fig. 3, respectively. This was the expected behavior and we do not show the results. Nevertheless, we observed a different behavior in the case when the VMs share one core, but they both have one independent core, as well. Fig. 4 shows that this scenario is quite like to the single shared CPU core scenario (i.e., Fig 2), but it inherits the sensitivity threshold of the single different CPU core scenario. The load percentages on the figure are doubled in this case, because maximum load for 2 CPUs is 200%.

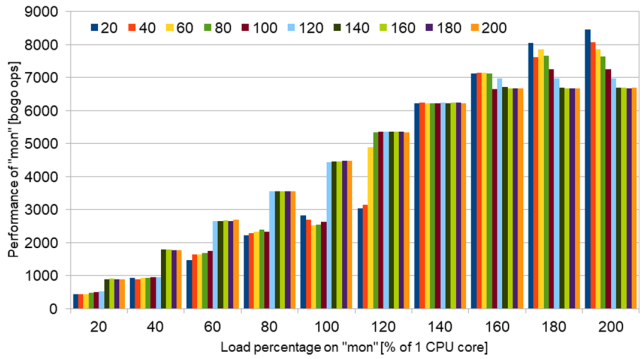


Fig. 4. Performance measured in bogo ops, when "gen" and "mon" share one of their CPU cores. The colored bars correspond to different loads on "gen", expressed as % of 1 CPU core capacity.

We also created a scenario, where *gen* had access to all four CPU cores, and *mon* had only one vCPU. Probably this scenario models the best a real deployment of a telco application getting the most computation resources possible, with a sidecar VM with limited CPU usage measuring it. Fig. 5 shows the results for this scenario (note that the maximum load of 400% corresponds to full utilization of 4 CPU cores). It is largely identical to the previous results: *mon* can detect changes in the load of *gen*, when that is low, however, when the load of *gen* is high, *mon* becomes blind.

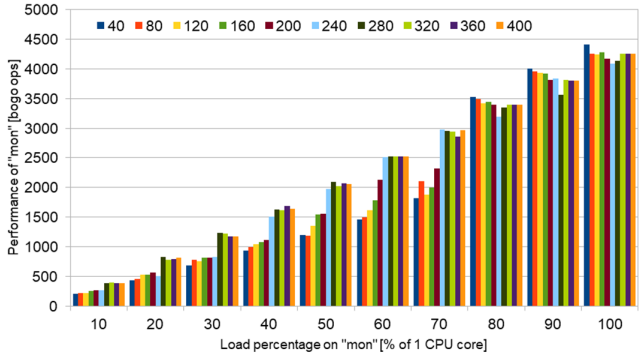


Fig. 5. Performance measured in bogo ops, when "gen" and "mon" share a single CPU core. The colored bars correspond to different loads on "gen", expressed as % of 1 CPU core capacity.

Note that in all the above scenarios *mon* can perform its load detection while generating small load itself. This is a nice property, as it allows running the performance monitoring sidecar with low impact on the telco application.

B. *Docker container-based deployments*

In this section we describe our results on testing the sidecar scenario when the processes were containerized. Similarly to the previous section, the container emulating the load of the target application was named *gen*, and the monitoring container was named *mon*.

In the case of container-based deployments we did not experience the dependence of the accuracy of load detection on the load level of the *mon* or the *gen* processes, as seen in the VM based deployments. Therefore in this section we compare the outcome of experiments with the same loads, but run on computers with different resource sets.

We compared two use cases: in the first case the containers run on the host (see Fig. 1 c), corresponding to a bare metal deployment of Docker containers. In the second one the two containers were run within a KVM/QEMU VM (see Fig. 1 b), modelling the widely used practice of deploying a container in a VM of a datacenter. The details of the VM, container setup, and the parameters of the load generator are all described in section III.

In these measurements the *stress-ng* was started at once (with the 4 stressors of different types set as shown in Table I), but we present them in four different charts: Fig. 6 for the CPU stressor, Fig. 7 for the memory stressor, Fig. 8 for the timer stressor and Fig. 9 for the pipe stressor.

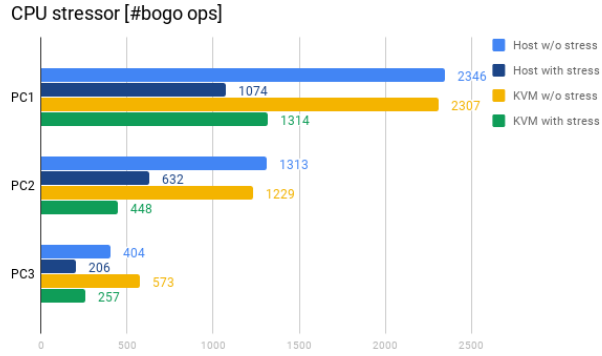


Fig. 6. Container-based scenario results with the CPU stressor.

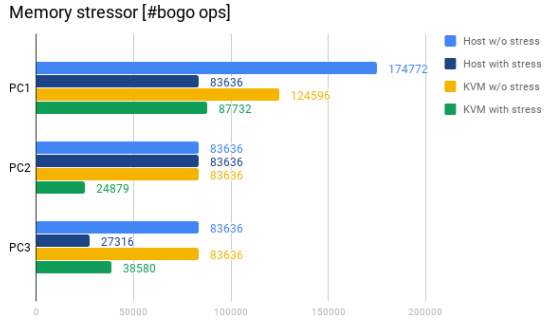


Fig. 7. Container-based scenario results with the memory stressor.

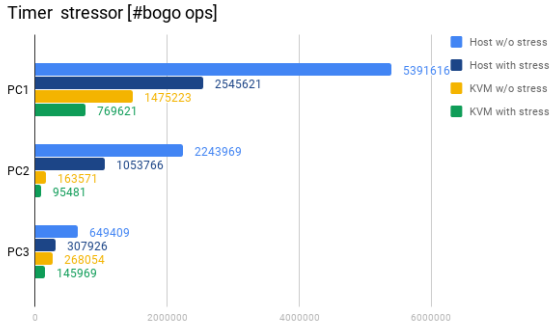


Fig. 8. Container-based scenario results with the timer stressor.

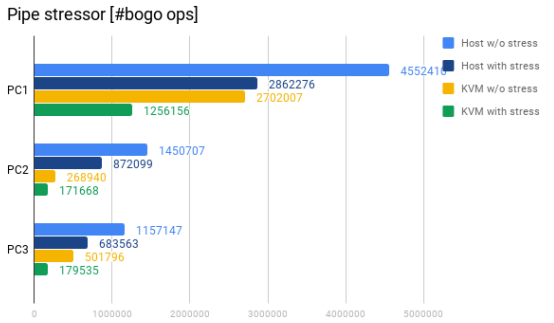


Fig. 9. Container-based scenario results with the pipe stressor.

In all four cases, for both host and VM based measurements it is clearly observable the effect of the stress on the *gen* container. It also can be seen that VM-based measurements result in lower values. However, the difference between the host-based and VM-based values depends on the stressor types: for the memory stressor the differences may be minimal (depends on the motherboard architecture and RAM type, not only on CPU type), whereas for the timer stressor we observed extreme differences.

For the stressors triggering *timer()* and *pipe()* system calls are much more sensitive to the computer architectures and react much more in terms of absolute value to the presence of load. Whereas this is useful to detect differences in both load and computational power, it has the drawback that it is volatile and has larger variance compared to the *cpu* and *memory* stressors.

In Table III we summarized the relative differences among the three PCs, calculated based on the *bogo ops*, as reported by the CPU stressor of *mon*.

TABLE III *CPU PERFORMANCE COMPARISON (RELATIVE TO PC1)*

Name	Measured by "mon" container	CPUboss.com benchmark values
PC1	1	1
PC2	0,56	0,46
PC3	0,17	0,24

In a separate column we show the *cpu score* based relative performance of the 3 CPUs, as provided by the *cpuboss.com* independent CPU benchmark site. It can be seen that our measurement accurately profile the 3 computers (note that the motherboard and RAM configurations correspond to the performance levels of the CPUs, thus this did not introduce further bias in the measurements).

C. Kubernetes based deployments

In the third experiment series we tested the sidecar scenario in a Kubernetes cluster. We deployed a pod running the two containers (*gen* and *mon*). Each container ran one stress-ng process each. The stressors were parameterized according to Table I, with the notable exception of starting 4 parallel CPU stressors in the *gen* container in order to allow it to consume as much CPU as it can.

During the tests, we started an external stress in a second pod, which stole resources from our pod. The *mon* container repeated the measurements in an infinite loop. The goal was to let the *mon* container measure the level of resource degradation.

The resource definition for the pod was set for CPU only. Within our pod, the *gen* container requested 1800 milli cores, and the *mon* container requested 200 milli cores of CPU, respectively. The external load that supposed to stole resources from our pod requested 1000 milli cores of CPU. The resource allocation policy was burstable (see Section II.A) and the pods were scheduled on PC2. The measurements have shown that the performances of the two containers (*mon* and *gen*) correlate. We verified the CPU usage on the host using the *top* tool. At the beginning of the experiment the pod generating the external load was not deployed, then we started the external load. The CPU consumption of the *gen* and *mon* containers before and after the external load is started is shown in Table IV. Initially the *gen* container uses as much resources as it can (3.8 CPUs). After the external load steals some resources (it gets 1.2 CPUs), the *gen* container can consume only ~60% of this resource (2.4 CPUs). The resource usage of the *mon* container scales down in a similar manner.

TABLE IV *THE CPU CONSUMPTION OF THE OBSERVED CONTAINERS DEPLOYED INTO A KUBERNETES CLUSTER, AS FUNCTION OF EXTERNAL LOAD*

External load?	CPU consumption of the "gen" container [milli cores]	CPU consumption of the "mon" container [milli cores]	CPU consumption of the "mon" container [bogo ops]
NO	3777	213	134
YES	2410	118	68

The 4th column of Table IV shows the measured values, as recorded by the "mon" container (expressed in *bogo ops*). The resource degradation level measured by the *mon* container is like the one observed at the host (3rd column) but is not exact match. This is because that the *stress-ng* load does not depend solely on the CPU usage. In practice this method must be calibrated to the proper application it is supposed to measure.

V. DISCUSSION OF RESULTS

The measurement results presented in this study were done on computers with four cores, and the results shown in the

previous section suggest that sidecar containers can detect if the main container is loaded just by monitoring the CPU frequencies, even if the two are pinned to different CPU cores.

A. The effects of the CPU frequency modifying mechanisms

The modern CPU architectures apply several optimization features, resulting in dynamic CPU resource availability that adapts to the load variations. Most of these features were introduced to increase the power consumption efficiency. The Intel CPUs implement frequency scaling in hardware, called *SpeedStep* technology. When a workload is deployed on one core, this technology raises the clock frequencies on all cores; the fewer cores are loaded, the higher their frequency can go.

Additionally to the above feature, a mechanism called *turbo frequency adjustment* aims to allow higher peak performances for short periods If multiple cores are loaded at the same time, their clock frequency drops below the maximum turbo frequency; thus, the overall computing capacity of the CPU doesn't scale linearly with the number of threads running.

We also ran some of the measurements detailed in section IV.A on a computing cluster, where the servers had CPU frequency scaling turned off in the BIOS. The measurement results confirmed that when the CPU frequencies are constant throughout the tests, the fluctuations presented earlier in that section are not present and the performance of the system scales linearly with the number of cores.

B. The effects of HyperThreading

Most Intel CPUs support the HyperThreading [16] technology, which allows a CPU core to share its computing resources between two threads, thus appearing as two virtual cores to the operating system. On Linux the CPU cores are ordered such that the second halves of the CPU cores are the hyperthreads of the first half of the cores, in the same order. We tested this experiment over PC4, which supports HyperThreading technology.

We repeatedly ran two simultaneous instances of *stress-ng* with one stressor process each for 20 seconds, as part of the KVM/QEMU-based measurement sets (see Section IV.A). Fig. 10 shows the measured CPU frequencies and the number of operations completed for various setups: only one stressor, both on the same core, on different cores, on the two hyperthreads of the same core. If both physical cores are loaded, the CPU frequency decreases by 100 MHz, which shows in the per-thread performance, but even in this case the CPU runs well above its nominal frequency. Running two stressors on the two hyperthreads of the same core yields higher performance than running them on the same logical core, but it is nowhere near the performance we get when using two separate cores.

Thus, HyperThreading can indeed improve the performance of parallel computations beyond the number of physical CPU cores, but it is more useful in improving the responsiveness on a desktop PC than increasing the computing power of a server.

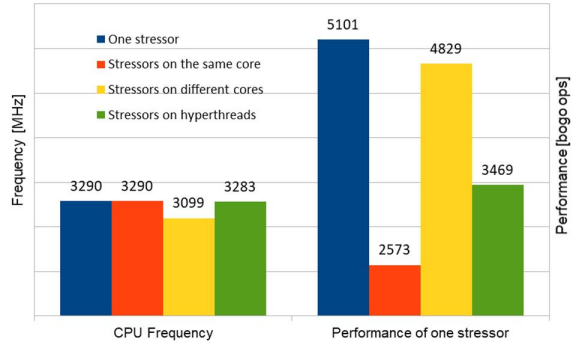


Fig. 10. HyperThreading results

Summarizing, if the monitoring process runs on the same CPU core as the monitored application, but on the other hyperthread, it can detect the load of the application while generating less interference than running on the same hyperthread. Of course, in a virtualized environment the processes running on the guest have no knowledge about HyperThreading of the host CPU; thus, exploiting it is usually not feasible.

C. The effects of different CPU architectures

The brief tests shown in this section already illustrates the dependence of CPU performance on the CPU architecture and setup.

Our measurements were taken on multiple different computers, but we were not able to cover every possible architecture. For example, AMD CPUs are known to scale the frequencies of the cores more independently of each other than Intel CPUs, and when there is more than one CPU in the machine, those also scale their frequencies independently of each other. These properties may affect the sensitivity of the sidecar measurements negatively. Heterogeneous architectures exist too: in the ARM world the so called *big.LITTLE* architecture is very popular: depending on the workload a low power or a high-performance CPU core may execute the task. In the future it might be worth investigating the possibility of using sidecar measurements on such architectures.

VI. CONCLUSION

In this paper we presented a measurement-based evaluation of the sidecar concept, aiming at evaluating the telecom application performance in a virtualized environment under dynamic load conditions. We considered several virtualization technologies and provided a quantitative analysis of the scenario.

According to our results the sidecar concept is viable. There is a correlation between the performance of the measurement application running in the sidecar and the resource usage of the main application running in a different VM or container. A good property of this measurement method is that the best sensitivity is achieved when the measurement application applies only slight load on the

system, thus creating low interference with the main application. The downside of this method is that it has low sensitivity when the main application is near full load, thus it cannot accurately predict an overload event. Running these measurements in a virtualized environment also adds challenges, as the visible resources not necessarily align with the resources that are physically available on that system.

ACKNOWLEDGMENT

The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications)".

The authors thank the valuable help, motivation and technical guidance of Attila Gál and Olga Papp from Ericsson Hungary. We also thank the help of László Sári, who supported us in setting up the measurement environment.

REFERENCES

[1] John, W., Moradi, F., Pechenot, B. and Sköldström, P., "Meeting the observability challenges for VNFs in 5G systems," *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pp. 1127-1130, 2017. DOI: 10.23919/INM.2017.7987445

[2] Burns, B., "How Kubernetes Changes Operations,"; *login:* The USENIX magazine, Vol. 40(5), 2015.

[3] Kernel Virtual Machine homepage – https://www.linux-kvm.org/page/Main_Page

[4] QEMU homepage – https://www.qemu.org/

[5] Libvirt, the virtualization API homepage – https://libvirt.org/

[6] Introduction to control groups (cgroups), RedHat documentation, https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/

[7] Namespaces - Overview of Linux namespaces, Linux Programmer's Manual, http://man7.org/linux/man-pages/man7/namespaces.7.html

[8] Docker homepage – https://www.docker.com/

[9] Kubernetes homepage - https://kubernetes.io/

[10] Configure Quality of Service for Pods, Kubernetes documentation, https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/

[11] Luong, D.H. et al., "Predictive Autoscaling Orchestration for Cloud-native Telecom Microservices," *2018 IEEE 5G World Forum (5GWF)*, pp. 153-158, 2018. DOI: 10.1109/5GWF.2018.8516950

[12] Van Rossem, S. et al., "Automated monitoring and detection of resource-limited NFV-based services," *2017 IEEE Conference on Network Softwarization (NetSoft)*, 2017. DOI: 10.1109/NETSOFT.2017.8004220

[13] Kraft S, Pacheco-Sanchez S, Casale G, Dawson S., "Estimating service resource consumption from response time measurements," *4th International ICST Conference on Performance Evaluation Methodologies and Tools*, pp. 1-10. 2009. DOI: 10.4108/ICST.VALUETOOLS2009.7526

[14] Khan M, Jin Y, Li M, Xiang Y, Jiang C., "Hadoop performance modeling for job estimation and resource provisioning," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 27(2), pp. 441-454, 2015. DOI: 10.1109/TPDS.2015.2405552

[15] stress-ng homepage – https://kernel.ubuntu.com/~cking/stress-ng/

[16] Marr et al., "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology Journal*, 2002.

Csaba Simon obtained his PhD degree at Budapest University of Technology and Economics, Department of Telecommunications and Media Informatics and he is working at the same Department since 2001. He is a member of the Balatonfüred Student Research Group. His research interests are mostly related to 5G systems and virtualization, IP QoS, peer-to-peer communications and network and service management. He was involved in several national and international research projects, covering his resarch topics. He is an active member of the Scientific Association for Infocommunications, Hungary, organising national conferences and being a contact for international relations and of the Sister and Related Societies Board at the IEEE ComSoc. He is the member of the International Working Group of the 5G Coalition, Hungary.

Markosz Maliosz received his PhD (2006) and MSc (1998) degrees in Computer Science from BME. He is a member of the Balatonfüred Student Research Group. He has participated in several national (OTKA-NKTH, TÁMOP, NFÜ) and EU-funded research projects (STREP, CELTIC, 5G PPP) and also worked in bilateral cooperation projects with Ericsson and Telia Research. His current research activity covers network virtualization and optimization focusing on industrial and cloud networking.

Miklós Máté received his MSc (2007) and PhD (2019) degrees in electrical engineering in the field of infocommunication systems at Budapest University of Technology and Economics (BME), Hungary. He is a research engineer in the High-Speed Networks Laboratory at the Department of Telecommunication and Media Informatics, BME. His research interests include intelligent transportation systems, distributed networks, and cloud technologies.

Dávid Balla is a PhD student at the University of Technology in Budapest, and also follows the PhD courses of the EIT Digital Doctoral School. He is a member of the Balatonfüred Student Research Group. He works at the High Speed Networks Laboratory at the university, and he is also the member of the research team at Ericsson Hungary. His main research topics are the physical and the software layers of cloud systems. During his master studies he worked with RDMA based interconnections and now he is dealing with Function as a Service and container based virtualization technologies.

Kristóf Torma graduated Budapest University of Technology and Economics in 2019. He is a member of the Balatonfüred Student Research Group. He joined the Faculty of Electrical Engineering and Informatics in 2020. His current research interest are cloud and container-based systems and their scaling behaviors, as well as scaling of IoT systems in Kubernetes.