Strong Temporal Isolation Among Containers in OpenStack for NFV Services

Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Riccardo Mancini, and Carlo Vitucci

Abstract—In this article, the problem of temporal isolation among containerized software components running in shared cloud infrastructures is tackled, proposing an approach based on hierarchical real-time CPU scheduling. This allows for reserving a precise share of the available computing power for each container deployed in a multi-core server, so to provide it with a stable performance, independently from the load of other co-located containers. The proposed technique enables the use of reliable modeling techniques for end-to-end service chains that are effective in controlling the application-level performance. An implementation of the technique within the well-known OpenStack cloud orchestration software is presented, focusing on a use-case framed in the context of network function virtualization. The modified OpenStack is capable of leveraging the special real-time scheduling features made available in the underlying Linux operating system through a patch to the in-kernel process scheduler. The effectiveness of the technique is validated by gathering performance data from two applications running in a real test-bed with the mentioned modifications to OpenStack and the Linux kernel. A performance model is developed that tightly models the application behavior under a variety of conditions. Extensive experimentation shows that the proposed mechanism is successful in guaranteeing isolation of individual containerized activities on the platform.

Index Terms—Cloud computing, containers, network function virtualization, temporal isolation

1 Introduction

Cloud Computing technologies have undergone a steady growth in the last decade, proving to be effective in the management of the complexity of nowadays distributed applications and services [1]. Cloud infrastructures evolved to complex systems offering a plethora of services that can be rapidly provisioned on-demand without human intervention, including [2], [3]: virtual machines, containers or even dedicated hosts; access to hardware-accelerated instances with GPUs or FPGAs; relational and NoSQL database services; in-memory caching solutions; performance monitoring, control and automation services; security services; specialized networking configurations; solutions for big-data processing including massive server-less deployments of custom processing topologies, as well as fully-managed platforms for the development, training and deployment of machine learning and artificial intelligence models; etc.

Public cloud computing has become increasingly adopted in many areas and businesses with computing requirements, with the only exception being those contexts where strong data confidentiality or tight latency requirements [1], [4] make it impossible to hand-over computations and data storage to a third-party remote infrastructure.

- *Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, and Riccardo Mancini are with the Scuola Superiore Sant'Anna, 56127 Pisa, Italy. E-mail: {tommaso.cucinotta, luca.abeni, mauro.marinoni, riccardo.mancini}@santannapisa.it.*
- *Carlo Vitucci is with the Ericsson AB, 164 83 Stockholm, Sweden. E-mail: carlo.vitucci@ericsson.com.*

Manuscript received 25 Feb. 2021; revised 25 Sept. 2021; accepted 26 Sept. 2021. Date of publication 28 Sept. 2021; date of current version 8 Mar. 2023. (Corresponding author: Tommaso Cucinotta.) Recommended for acceptance by C. Wu. Digital Object Identifier no. 10.1109/TCC.2021.3116183

However, the core Cloud Computing principle that a general-purpose computing infrastructure can be sliced into shares that are rapidly provisioned on-demand with minimal management effort [5] and can be elastically resized has become pervasive. It is being applied in several medium-to-big organizations in the form of the so-called *private* cloud computing: different departments or functions can rapidly instantiate computing services within a shared private infrastructure that can be quickly, adaptively, and automatically re-arranged to dedicate more processing, storage or networking power to the core activities that are exhibiting higher workloads at any time.

An important area where private cloud computing infrastructures are becoming increasingly used is the one of network operators and telecommunication systems [4], [6]. In this context, the technological evolution of communication and processing equipment is leading to the transition from a number of custom communication media and protocols, and dedicated appliances, to a converged networking infrastructure heavily based on TCP/IP and general-purpose computing servers. This enables the wide deployment of private cloud computing solutions in the context of what is known as Network Function Virtualization (NFV) [7]. The NFV paradigm allows replacing expensive physical appliances that have to be sized for peak-hour operations with elastically provisioned computing services (virtual machines or containers) that can be provisioned on-demand and dynamically adapted to the instantaneous load continuously changing over time.

A specific area where NFV is being heavily applied is the fifth-generation wireless systems (5G), where an unprecedented degree of flexibility is needed to handle the complexity of modern and future communication infrastructures supporting vertical markets such as automotive, Internet-of-Things (IoT), ultra-low latency applications [8], [9] and enhanced mobile broadband [10]. Future mobile scenarios

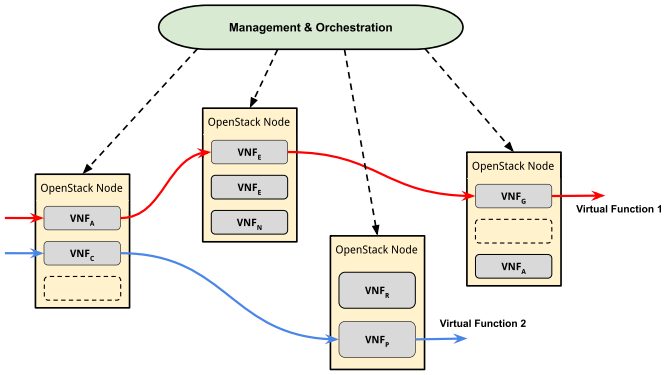


Fig. 1. End-to-end service deployment based on slice allocation.

will need infrastructures able to guarantee higher throughput, connection density and traffic capacity, better spectrum efficiency, higher network efficiency, and lower and more predictable latencies.

These constitute essential requirements for next-generation communication systems, where the 5G architecture combines cloud principles in order to achieve efficient and cost-effective deployment and control of a vast plethora of services over an end-to-end network.

1.1 Problem Presentation

In the context of the 5G architecture and NFV, a wide range of services exists with stable and predictable performance needs [8]. This is a requirement that is often in conflict with the idea of sharing physical resources due to the unavoidable temporal interferences among co-located services (a.k.a., the *noisy neighbor* problem). Therefore, the concepts of *network slice* and *slice isolation* are mandatory to provide resource slices with sufficiently stable performance to deploy entirely independent services with guaranteed service levels that do not suffer from undesirable and unpredictable interferences leading to performance drops or latency peaks.

In NFV, vertical services are built over a chain of virtualized network functions (VNFs), such as switching and routing, mobility management, signaling, tunneling and gateWaying, load balancing, traffic analysis and policing, and security services, all handled according to the ETSI NFV Management and Orchestration (MANO) guidelines [11] (see Fig. 1). In this context, slice allocation and isolation is the capability to allocate and isolate to a VNF the needed network, storage, and computing resources. Another essential characterization of the 5G architecture is that data center, core and access networks shall support quite different and customizable requirements. For example, 5G requests a widely distributed computing capability at the edge of the network to support Smart City and Machine Type Communications (MTC). Efficient resource usage and limited power consumption require fine-grained allocation and scheduling of slices over resources, so to achieve a proper degree of temporal isolation. Moving from the edge to data centers, the granularity of resource usage can be defined on a different scale.

It is remarkable that, in this area, the presence of high throughput and low-latency requirements, coupled with the possibility to use dedicated infrastructures with full control on the deployment environment, led to a progressive departure from traditional cloud architectures heavily based on virtual machines, in favor of lightweight virtualization solutions at the operating system (OS) level based on containers [12], [13], [14], [15], [16]. These do not suffer from the performance drawbacks of machine virtualization, at the cost of decreased flexibility, which is acceptable in the context. Also, the predictability requirements on the end-to-end performance recently led to the investigation of real-time scheduling techniques [17] to support a stable performance of co-located components [14], [18], [19]. The industrial interest in OS containers in the context of NFV is witnessed by the recent ETSI NFV MANO specification including an object model for OS container management and orchestration [20], complementing the plethora of existing approaches to container orchestration in the general cloud-computing domain [16], [21].

Issues of temporal interference among co-located containers or Virtual Machines (VMs) typically arise in shared infrastructures. Their weight increases when CPUs oversubscription is allowed or other resource bottlenecks are present, like disks or networks in data-intensive workloads. Concerning processing performance, interferences among co-located containers are typically managed via a 1-to-1 virtual CPU (vCPU) to physical CPU (pCPU) allocation (i.e., forbidding CPU over-subscription). Nevertheless, a minimum granularity equal to a single CPU is implied, resulting in a likely large infrastructure under-utilization. Consequently, infrastructure providers must select between: a) deployments with an amount of over-subscription that could lead to volatile single-service performance, due to the variable workload of the co-located ones on the same pCPU(s); b) deployments without over-subscription, presenting stable performance but also high infrastructure under-utilization. Both alternatives are affected by additional shortcomings in a wide range of application scenarios, including the nodes at the edge of the NFV network, where the QoS during traffic peak hours is affected by over-subscriptions and 1-to-1 allocations enormously raise power consumption, which dominates the energy budget for access networks.

1.2 Contributions

This paper presents a novel approach to the problem mentioned above, extending and complementing prior research by the same authors [14], [22], [23]. The proposed approach uses a fine-grained allocation of pCPU(s) achieved by using a real-time CPU scheduler built into the OS, which we realized modifying the `SCHED_DEADLINE` [24] real-time CPU scheduler in the mainline Linux kernel, so to provide hierarchical scheduling capabilities. This way, the OS kernel can guarantee the allocation of fine-grained shares of pCPU time to individual containers, providing the hosted services with a stable and predictable performance.

The low-level real-time CPU scheduling capability is leveraged in the cloud/NFV orchestration domain through modifications to the well-known OpenStack cloud management software and exposed to the MANO descriptor layer for NFV deployments through the Tacker service, resulting in proper support for isolating the performance of individual containerized VNFs.

The fundamental kernel-level mechanism for strong temporal isolation among containers enables the development of accurate performance models for distributed services and

applications, based on an approximated model of our proposed real-time CPU scheduler, resulting in an effective decision-making instrument to higher-level cloud/NFV orchestration layers for managing the underlying resources.

The approach is validated applying the methodology to a real test-bed where we performed extensive experimentation deploying, in the form of containers, both a synthetic distributed application with Poissonian traffic characteristics and a real IMS signaling application, using our modified OpenStack orchestrator leveraging our modified kernel-level scheduler. Experimental results show performance figures matching quite closely with our theoretical model expectations, in terms of average value and high percentiles of the response-time service distribution, confirming that the proposed approach allows for deploying time-sensitive cloud/NFV services with predictable performance levels.

Compared to the previously published material on the topic [14], [22], [23], besides reviewing the most important concepts and previously obtained results, this work presents for the first time: the modifications we applied to OpenStack and specifically to the Nova service in order to support real-time scheduling of containers through our hierarchical extension of the SCHED_DEADLINE policy within the Linux kernel (Section 5); the extensions needed at the MANO descriptors level to support the new features from the Tacker NFV management framework (Section 5.1); the support for real-time parallelizable software components within multi-core containers (Section 5.2); and results about the performance achieved when deploying a multitude of concurrent activities using the proposed architecture (Section 6).

1.3 Paper Structure

This paper is structured as follows. After a survey of the related research literature in Section 2, some background concepts useful for a better understanding of the paper are recalled in Section 3. Section 4 introduces the main bricks of the proposed approach to design distributed components with predictable performance levels. For instance, we elaborate on mechanisms for performance isolation based on an underlying EDF-based real-time scheduler for containers extending the Linux kernel and a performance model that is built for a specific class of distributed applications of interest. Section 5 provides implementation details showing how the OpenStack architecture has been modified to take advantage of the novel scheduling features, including the changes required to its Nova component, to the command-line interface, and the additional modifications that allowed for the integration within the NFV orchestration manager Tacker. Section 6 provides experimental results regarding the performance of a synthetic application workload under different conditions. These constitute an extensive validation of the correctness of the performance model built thanks to the proposed methodology and software architecture, highlighting also some limitations of the theoretical model. Finally, Section 7 provides our concluding remarks, outlining possible areas of further research in the future.

2 Related Work

Network operators are broadly developing NFV solutions leveraging on the cloud paradigm and are working on bringing it up to the edge of the network [8]. For example, the goal of Virtualized Radio Access Network (VRAN) is moving computational functionalities of the networking stack from the radio heads to dedicated servers stationed at the edge of the infrastructure [25], [26], [27], [28]. This solution has to deal with the compelling timing constraints involved, like the one of $4ms$ imposed in packets acknowledgment by the Hybrid ARQ (HARQ) [29] protocol. Timing constraints like these can be handled by analysis and profiling within the single network functions and performing an ad-hoc parallelization, as done by some authors for Cloud-RAN [28], or proper mechanisms can be used to enforce predictability in the underlying infrastructure at the bottommost software layers [30], if possible.

In the area of cloud computing and distributed service-oriented systems, the presence of particularly stringent timing constraints is traditionally tackled by: 1) dedicating physical resources to individual components or network functions (e.g., virtual machines deployed with their virtual cores pinned to dedicated physical cores that are not shared with others), or less commonly using dedicated physical hosts; 2) adopting high-performance computing paradigms and middleware components, including tuning of the operating system [31] especially in the presence of platforms with a non-uniform memory architecture (NUMA), adopting high-performance networking primitives heavily based on continuous polling and batching of I/O operations, and bypassing the OS kernel in favor of reduced functionality in user-space for faster I/O operations. For example, popular kernel-bypass techniques [32], [33] include the well-known Data-Plane Development Kit (DPDK) [34] originally by Intel, and the Netmap research project [35]. An interesting survey on fast packet I/O technologies for NFV can be found in [36]. It is useful to remind that, in the context of NFV, besides seeking for the maximum possible performance, dependability constitutes also a stringent requirement [37].

However, the just mentioned techniques come usually at non-negligible costs in terms of CPU workload, as the adopted software switching logic, heavily based on non-blocking (continuously polling) primitives, forces entire CPUs to stay busy at 100% utilization, even in conditions of moderate traffic. Therefore, there is still a strong interest in the industry in exploring the advantages of co-locating network functions that are unable to saturate entire physical CPUs individually, trying to control the possible temporal interference among them through proper mechanisms at the OS or hypervisor level.

For example, applying real-time scheduling techniques within the hypervisor allows for providing tuning mechanisms in resource allocation and then controlling the interferences of co-located services [38], [39]. Indeed, some authors proposed [39] to support real-time processing in the Xen bare-metal (Type-1) hypervisor with novel features, such as the application of hierarchical real-time scheduling techniques allocating to individual VMs accurate slices of the physical CPU execution time. The mechanism has also been integrated within OpenStack [18] replacing the hypervisor and the VM allocation mechanisms with RT-based ones, able to enforce timing isolation among colocated VMs. This way, it has been proposed [40] to exploit the capabilities of this RT-aware version of OpenStack as an enabling

technology to optimize the placement of network functions composing a service chain and obtain a higher resources utilization while respecting the timing constraints defined in the SLA.

Similar concepts appeared earlier in the context of the IRMOS European project [41], where real-time scheduling of virtual machines deployed with the KVM hosted (Type-2) hypervisor was integrated within a comprehensive cloud management solution for guaranteed resource provisioning to service-oriented real-time and multimedia applications. A fundamental brick of the IRMOS architecture was the so called Intelligent Service-Oriented Infrastracture (ISONI) [42], an abstraction layer able to deploy an arbitrary topology of VMs with precise individual as well as end-to-end QoS requirements, called a Virtual Service Network (VSN), within a computing and networking infrastructure capable of providing precise computational and networking guarantees. This was possible thanks to the combination of real-time scheduling of VMs with QoS-aware networking protocols. Carefully designed VM placement strategies [43], [44] allowed for deploying VSNs so as to satisfy either deterministic or probabilistic end-to-end QoS (and particularly latency) requirements for the deployed applications. However, the IRMOS framework used an ad-hoc architecture developed from scratch, providing no support for integration with current most common cloud infrastructures used in the SDN-NFV context, such as OpenStack or Kubernetes.

The just discussed solutions heavily rely on machine virtualization to gain flexibility in resource management, and take advantage of real-time scheduling techniques at the hypervisor level to employee temporal isolation and reduce unpredictabilities among co-located virtualized components. However, solutions exploiting classic machine virtualization are well-known [13], [14], [20], [45], [46] to cause significant processing and memory occupation overheads, resulting in a bulky software stack with duplicated resource management and scheduling functionality at the hypervisor and guest OSes level. Therefore, especially in the domain of NFV, where a network operator has its own private cloud infrastructure that is fully managed for deploying specialized VNF software, containerization is quickly superseding machine virtualization thanks to its lighter architecture and reduced overheads, paving the way towards higher efficiency and reduced end-to-end latencies greatly benefitting low-latency applications and services.

Additional works that are worth to mention are those adopting: reinforcement-learning techniques to tackle efficient placement of NFV service chains [47]; machine-learning algorithms for the elastic scaling of resources assigned to individual containers [48]; neural networks to grasp the complex relationships among the available deployment options and the expectable VM/service performance, so to facilitate VM placement and configuration decisions [49]. These kinds of activities exploit virtualization and can be extended to take advantage of virtualization solutions to increase isolation and reduce unpredictabilities.

However, the mentioned solutions propose genuine virtualization with all the related overheads, which can be avoided with a container-based approach. Interestingly, real-time scheduling of VMs in IRMOS was done through a patch enriching the Linux kernel with real-time hierarchical scheduling capabilities [38] based on *control groups* (`cgroups`), so it could also be applied to containers.

However, these enabling mechanisms are not enough to guarantee that the timing constraints of hosted services are respected. To reach this goal, they must be coupled with appropriate modeling and analysis techniques. The importance of the latter ones for designing predictable end-to-end cloud services in the context of network service-chains has been highlighted for example in [50], where resources are dynamically allocated by solving an optimization problem designed around an end-to-end performance model.

A number of authors also focused on the problem of optimizing VM placement in a cloud infrastructure to better host real-time workloads and mitigate interferences of co-located instances. For example, in [51] authors propose to tackle an energy-aware VM placement optimization problem via simulated annealing. In [52], authors propose to place VMs by recurring to machine learning models that have been trained on historical data about resource usage and availability within the infrastructure. In [53], authors propose an optimum VNF allocation solution specialized in elastic IMS functions leveraging the WebRTC protocol, with the ability to ensure precise QoS levels to users.

In [54], a model-based analytics approach is proposed to profile VNFs based on a network queueing model that is able to capture the burstiness of the workload, as well as effects due to interrupt coalescing and power management actions on the physical servers, to estimate network KPIs related to power consumption and communication latencies.

With respect to the state of the art, the proposed approach addresses the problem at different levels in an integrated way. It starts exploiting novel kernel mechanisms to improve isolation among containers and increase response times predictability. On top of it, an analysis has been provided to express the dependency between the reservations configuration parameters and performances (e.g., response time, latency), allowing more fine-grained decisions during the placement phase. Finally, these improvements have been integrated into standard cloud solutions to make them available for use within well-known infrastructures applied for SDN-NFV deployments.

3 Background Concepts

This section outlines some basic concepts on real-time scheduling mechanisms for the Linux kernel, virtualization techniques, and NFV deployment solutions based on OpenStack, which constitute a useful background for a better understanding of the sections that follow.

3.1 Real-Time Scheduling in the Linux Kernel

The Linux kernel provides a POSIX-compliant CPU scheduler [55], which implements different *scheduling policies* to select the tasks to be dispatched on the various CPU cores. The POSIX standard requires to implement 3 scheduling policies: `SCHED_FIFO`, `SCHED_RR` and `SCHED_OTHER`. `SCHED_FIFO` and `SCHED_RR` use a fixed priority scheduler, where a

numerical priority is associated to each task[1] and the tasks with the highest priorities are scheduled first, while SCHED_OTHER uses a system-dependent scheduling algorithm. On Linux, SCHED_OTHER is realized as a fair-share scheduler coupled with various heuristics to handle properly and boost the scheduling opportunities of interactive workloads compared to long-running batch activities.

Since SCHED_FIFO and SCHED_RR can be used to implement real-time applications (associating priorities to tasks according to Rate Monotonic [56], for example), they are scheduled in foreground respect to SCHED_OTHER (first, the scheduler checks if there are SCHED_FIFO or SCHED_RR tasks ready for execution, then, it select a SCHED_OTHER task only if no SCHED_FIFO nor SCHED_RR tasks are ready to run) and are generally known as "real-time scheduling policies".

The POSIX real-time scheduling policies are a good choice to schedule real-time tasks when their execution times and activation patterns are known in advance (and the real-time workload is almost static). However, they risk starving non-real-time (SCHED_OTHER) tasks when the real-time tasks consume more execution time than expected. Moreover, they require to re-arrange the tasks' priorities when new real-time tasks are created. The Linux kernel implements some non-standard mechanisms such as real-time throttling [57] and real-time control group scheduling [58] to address some of these issues, but such mechanisms reduce the predictability of the scheduling algorithm and make it much more difficult (if not impossible) to provide exact performance guarantees.

For this reason, the Linux kernel implements an additional real-time scheduling policy[2] named SCHED_DEADLINE, which is scheduled in foreground respect to all the other policies. The basic idea of this scheduling policy is to implement *reservation-based scheduling*, allowing each task to execute for at most a reserved runtime Q every period P and at the same time guaranteeing that each task is allowed to execute for its whole reserved time. As a result, each SCHED_DEADLINE can consume at most a fraction Q/P of the CPU time (where Q and P are the task's runtime and period) and is guaranteed to be able to use such a fraction of CPU time. This result is achieved by using a multi-processor variant [24] of the Constant Bandwidth Server (CBS) [59] algorithm to assign *scheduling deadlines* to tasks, and then schedule them through EDF [60].

In more detail, the scheduler tracks the amount of time used by each task (by using a *current runtime* which is initialized to Q and decreased when the task executes). Then, when a SCHED_DEADLINE task wakes up (becomes ready for execution), the scheduler checks if the task can use its current runtime and scheduling deadline without consuming more than Q/P of the CPU time; if not, a new scheduling deadline equal to $t + P$ (where t is the current time) is generated and the current runtime is reset to Q. When the current runtime of a task reaches 0, the task is "depleted", or "throttled", and cannot be scheduled until the time reaches its scheduling deadline. At this point, the scheduling deadline is postponed by P, and the current runtime is recharged to Q. SCHED_DEADLINE can be configured as a *global* scheduler, a *partitioned* scheduler, or even as a *clustered* one (with tasks partitioned across subsets of cores, then globally scheduled within each cluster of cores).

3.2 Virtualization and Containers

One of the foundational pillars of cloud computing is machine virtualization, by which a software layer takes control of the (hardware or software) resources available on a physical machine (processors, cores, RAM, storage, network interfaces) and distributes them among a set of isolated execution environments — the virtual machines (VMs). Each VM can potentially host a different *guest* OS, which has access to a subset of the resources available underneath. Sometimes the resources can also be temporally scheduled so that multiple VMs can use them in different time slices, as needed at run-time. In Type-1 virtualization, the hypervisor runs on bare metal, while in Type-2 virtualization, it is a software deployed on a *host* OS.

In *full* hardware virtualization, a software component named *hypervisor* emulates (or virtualizes) the presence of specific hardware devices and peripherals within the VMs; thus, guest OSes use their original, unmodified device drivers within the kernel, to handle said resources. If the CPU ISA is properly designed [61], virtualized software can manage to exploit the underlying capabilities of the physical CPU at full speed, as long as only user-space computational activities are carried on. However, when I/O is needed (disk or network access), or OS services are invoked (creation of new processes, needing modifications to the virtual memory pages), the hypervisor needs to continuously interrupt the guest execution (by means of *traps* causing *VM exits*) as a result of guest kernels trying to access I/O or special CPU registers, to emulate the corresponding effects in the context of each VM environment. This results in the advantage of deploying unmodified OS images, but on the other hand, it often results in unacceptable virtualization overheads.

Traditional means of mitigation consist in the use of hardware-assisted virtualization, which leverages on special capabilities of various hardware components to virtualize themselves. These are configured by the hypervisor so that each VM is offered a distinct set of virtual registers that can be programmed by the guest OS, using its in-kernel device driver, without any intervention by the hypervisor while using the virtualized peripheral. This happens with network interface cards (NICs) with SR-IOV capabilities, or with nested memory pages, for example.

On the other hand, virtualization overheads can also be mitigated by software means, recurring to *para-virtualization* [62], a widely used technique in modern OSes, that exploits a special API exposed by the hypervisor to guest OS kernels: special device drivers in the guest OS kernel now can directly and explicitly invoke a hypervisor service via a *hypercall* (similarly to how user-space software invokes a *system-call* in an OS), achieving an I/O performance that is normally very close to the one of non-virtualized environments.

However, para-virtualized set-ups still suffer from a number of non-optimal conditions due to the replication of

1. The term "task" is used in this paper to indicate a schedulable entity that can be either a single-threaded process or a thread of a multi-threaded process.

2. This is not a violation of the POSIX standard, which mandates the presence of SCHED_FIFO, SCHED_RR and SCHED_OTHER, but allows to implement additional scheduling policies.

several functions across the whole software stack. Indeed, often we can find functionality related to process scheduling, memory allocation, networking stacks and security, device drivers, and machine management services replicated both in the hypervisor (if Type-1) or the host OS (if Type-2), and in every hosted guest OS, on the same machine. Additionally, several guests may run instances of the same identical services independently from one another (as it often happens when guests are based on the same OS type and version). This creates a software stack that is often needlessly bulky (requiring more RAM and causing more cache misses than strictly needed).

Therefore, an increasing interest arose in lightweight OS-level virtualization solutions like *containers* [63], where a single host OS kernel is shared among multiple runtime environments. This can be done with a wide range of possible configurations, from a fully feature-rich container that runs a full user-space part of the OS (that has to be compatible with the host OS kernel) to the possibility of a very lightweight containerized application that runs exploiting the OS services, libraries and middleware available in the host OS. With containers, hosted applications not only process at the underlying hardware full speed, but they can also perform operations related to I/O, memory and process/thread management at the maximum speed allowed by the host OS, with the advantage of a much more lightweight set-up, compared to traditional virtualization.

It is quite understandable that, especially in the context of a private cloud infrastructure where high-performance VNFs are deployed to perform packet-processing in custom ways, possibly even bypassing the in-kernel stack to squeeze any single bit of performance out of the hardware, a containerized software stack can easily become more interesting than a virtualized one.

3.3 OpenStack Deployments for NFV

OpenStack is an open-source software for cloud computing infrastructure management. Due to its high modularity, its use as a VIM (*Virtual infrastructure Manager*) for NFV deployments is popular among telecommunication service providers and enterprises [64], [65], [66], [67], [68], [69]. In its simplest deployment, it requires one or more compute nodes, which will host the virtual machines or containers, and a controller node, which will take care of the management of the VMs and of the network.[3] Some of the drawbacks of the OpenStack architecture have been addressed in the literature, like the controller node acting as a single point of failure. For example, some authors proposed [70] to replace the central SQL backend supporting many of the orchestrator features with an entirely distributed peer-to-peer key/value store.

The most important OpenStack components for a NFV deployment are (see Fig. 2):

- *Nova* (compute service): manages the VM life-cycle from user request to the *Nova API* to the actual VM creation, e.g., with *KVM* using *libvirt*;
- *Neutron* (network service): manages the virtual networks with software switches like Linux bridge or Open vSwitch, and possible SDN enhancements, e.g., via OpenDayLight;
- *Tacker* (VNF manager and orchestrator): deploys VNFs on top of the VMs created by Nova.

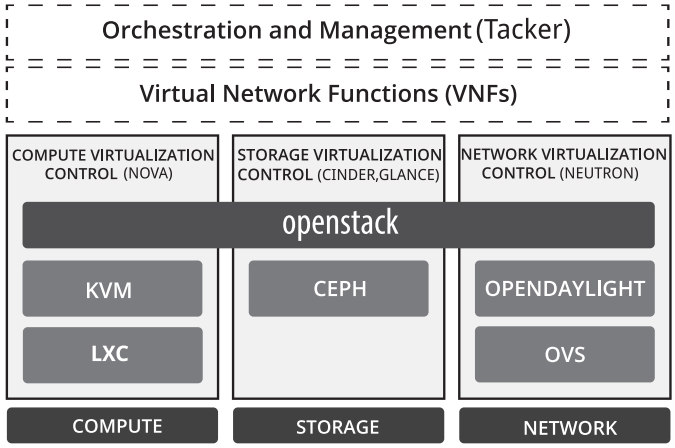


Fig. 2. Example of OpenStack for a NFV architecture.

Other components are also required for a working OpenStack deployment, e.g.: *Keystone* (identity), *Glance* (VM images management), *Horizon* (web-based dashboard), *Cinder* (block storage), *Ceph* (scalable software-defined storage), *Ceilometer* or *Monasca* (telemetry services).

4 Proposed Approach

This work aims at providing isolation among the performance of individual co-located containers (hosting CPU-intensive packet processing services) within an NFV platform. The containers' workloads lead to potential high contention of the CPU[4] and present a wide range of timing requirements determined by the specific due classes of traffic handled. Our approach revolves around a *real-time CPU scheduler* for containers co-located on the same physical node, reserving a well-defined (and controlled) amount of CPU time to each container as explained in Section 4.1. This is combined with reservations at the networking level, allowing to build a multi-resource performance model for distributed applications. A detailed modeling example is presented in Section 4.2, showing how the predictability obtained for the single containers can be exploited, and a probabilistic performance model of a simple synthetic packet processing service be built. Consequently, statistics of the response-time distributions are easily tied to the configurations of servers. Note that the theoretical model has been validated through extensive experimentation, as shown in Section 6, using synthetic workload, as well as a real IMS use-case scenario.

4.1 CPU Scheduling of Containers

Linux containers, as created via userspace tools like `lxc` and `lxd`[5] or docker,[6] are based on *namespaces* [71],

3. Network management can also be deployed as a separate node.

4. An extended version, including additional QoS control mechanisms at the networking, disk, or I/O layers, can be applied in data-intensive scenarios.

5. More information is available at: https://linuxcontainers.org/.

6. More information is available at: https://www.docker.com/.

controlling the resources' visibility, and *control groups*[7] (cgroups) allowing to control the amount of used resources and to schedule resources such as memory, physical CPUs, and similar. In particular, each container is associated with a "real-time cgroup" that allows controlling the amount of time used by real-time tasks (SCHED_FIFO and SCHED_RR) within the container. This is done by "throttling" real-time tasks once the specified fraction of time has been used.

The mechanism used in this paper, named "*container-based real-time scheduling*", extends the Linux scheduler to build theoretically-sound scheduling hierarchies through real-time cgroups (instead of merely throttling real-time tasks). In more details, the "Hierarchical CBS" (HCBS) scheduler [17][8] extends to task groups the SCHED_DEADLINE CPU scheduling class [24] that the Linux kernel already provides to support single-threaded real-time applications. As discussed in Section 3, in the original SCHED_DEADLINE implementation each task is attached with a CPU reservation, described by a *reservation runtime* Q guaranteed on the CPU every *reservation period* P. Our HCBS scheduler allows to attach such a SCHED_DEADLINE reservation to a group of tasks (a control group) instead of a single task.

Hence, the HCBS scheduler adds hierarchical scheduling capabilities [38], [39] to SCHED_DEADLINE, allowing this scheduling policy to schedule not only single tasks, but also groups of tasks (control groups). The HCBS implementation is based on the fact that the Linux kernel stores the scheduling context of a process or thread in a *scheduling entity* structure. Each task has a scheduling entity for each possible scheduling policy (including an *rt entity*, a *dl entity*, and so on). With HCBS, dl entities are associated not only to *SCHED_DEADLINE* threads and processes, but also to real-time control groups, and the scheduler works as follows:

- When a real-time task (process or thread scheduled by SCHED_FIFO or SCHED_RR) wakes up, the scheduler inserts it (or, better, its rt scheduling entity) in the ready real-time tasks queue (rt runqueue in Linux). If the task is not in any real-time control group, this rt runqueue is directly handled by the scheduler, but if the task is in a real-time control group this runqueue is associated with a dl scheduling entity. Hence, if this is the first task in the rt runqueue then the associated dl entity is inserted in the dl runqueue.
- When a real-time task blocks, it is removed from its rt runqueue. If such a runqueue becomes empty and the task is in a real-time control group, then the associated dl entity is removed from its dl runqueue
- When the scheduler has to select a task to schedule, it first checks if there are ready dl entities in the dl runqueue. If a dl entity is selected, then the scheduler checks if this scheduling entity is associated to a task (a SCHED_DEADLINE thread or process) or to a real-time control group. In the second case, the scheduler selects the highest priority real-time task from the rt runqueue associated to the dl entity.
- When a task served by a dl scheduling entity executes, for a time dt, the scheduler decreases the entity's runtime by dt. In a vanilla kernel, the only tasks served by dl entities are SCHED_DEADLINE tasks; when the HCBS scheduler is used, all the SCHED_FIFO and SCHED_RR tasks in a real-time control group are served by the group's dl entities.
- When the runtime of a dl scheduling entity arrives to 0, the entity is throttled and removed from the runqueue (the ready queue) until the end of the reservation period. If the entity is associated to a real-time control group, the scheduler tries to migrate real-time tasks of the group to a different core, where the group is served by an entity with runtime > 0.

Interested readers can refer to the original CBS [59] and SCHED_DEADLINE [24] papers to see all the deatails about how the scheduler handles the dl entities.

TABLE 1
List of Symbols Used for the Probabilistic Analysis

Symbol	Definition
m	Number of pCPUs (physical CPUs)
n	Number of real-time servers hosted on the same pCPU
Q_i	Runtime for the real-time control group hosting server i
P_i	Period for the real-time control group hosting server i
λ_i	Average inter-request time for requests towards server i
μ_i	Average processing time of requests towards server i
z_i	Size of requests for server i
R_i^e	Stochastic variable representing the RTT for a request to server i
t_i^S	Stochastic variable representing the time needed by a request to travel to server i from one of its clients
t_i^P	Stochastic variable representing the processing time for a request on server i
t_i^R	Stochastic variable representing the time needed by responses of server i to reach their requesting client
σ	Network bandwidth
δ	Network delay

Note that the i index may be dropped to make the notation easier to read.

4.2 Probabilistic Model

The performance of the container-based real-time scheduling approach described above can be analyzed by both using a deterministic approach based on the Compositional Scheduling framework (CSF) [39], [72] (which is able to provide hard real-time guarantees) or a relaxed probabilistic approach [14], which is more suitable in the cloud context.

The probabilistic analysis approach can be used, for example, to provide soft real-time guarantees to a set of containerized server applications (servers) running on a physical host with m identical CPU cores (referred to as "*pCPUs*"). This analysis, based on the definitions provided in Table 1, is performed assuming that each server i is hosted in a container scheduled using HCBS (with parameters Q_i and P_i), and a number n of different containerized servers are hosted on a single pCPU, on the machine. Moreover, each containerized server i receives a pattern of (aggregated) requests having exponential and i.i.d. inter-request times with average rate λ_i and exponential and i.i.d. request processing times with average rate μ_i. The latter corresponds to processing times of

7. More information is available at: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

8. Source available at https://github.com/lucabe72/LinuxPatches/tree/Hierarchical_CBS-patches.

individual requests when an entire CPU is dedicated to the server while its speed has been locked to its maximum value, with hyperthreading disabled, if present. Therefore, the performance model we build does not depend on Dynamic Voltage and Frequency Switching (DVFS) mechanisms. Proper modeling of the effects of DVFS on performance might be the subject of future works. Furthermore, μ_i does not include possible queueing times if requests queue up before being processed.

The complete analysis is reported in our prior work [14], however here we report a summary of the most important results, based on the fact that the end-to-end round-trip time (RTT) for a request can be modeled as a stochastic variable

$$R_i^e = t_i^S + t_i^P + t_i^R,$$

where: t_i^S is the time to *send* the request from the client to the server i; t_i^P is the time to *process* the request within server i, including possible queuing time (server response time); t_i^R is the time a response of server i needs to reach the client.

Our probabilistic analysis allows to approximate the average RTT $E[R_i^e]$ and its ϕth percentile $P_\phi[R_i^e]$. This approximation is based on well-known results from real-time theory and queuing theory, as detailed below.

4.2.1 Processing Times Under HCBS Real-Time Scheduling

The first observation we make is that, if a server with a per-request processing time of C_i time units (if executing on a dedicated physical core) is deployed on a real-time container scheduled by a reservation (Q_i, P_i) with P_i small enough compared to C_i, and the scheduling parameters for HCBS servers co-located on the same pCPU satisfy the condition

$$\sum_i \frac{Q_i}{P_i} \leq 1, \tag{1}$$

then the server response time t_i^P can be approximated as

$$t_i^P \cong \frac{C_i}{Q_i/P_i}, \tag{2}$$

under a simple no queueing assumption.

Proof. It is well-known [24] that, if n real-time tasks are scheduled on a single CPU under a reservation-based scheduler like the HCBS, with scheduling parameters satisfying the condition in Eq. (1), then each task is guaranteed to be scheduled on the CPU for Q_i time units in each time window of duration P_i. This means that, if the computation requires a processing time C_i when running on the CPU in isolation and without any reservation, then its completion time t_i^P under a (Q_i, P_i) reservation is bounded by:

$$\left\lfloor \frac{C_i}{Q_i} \right\rfloor P_i \leq t_i^P \leq \left\lceil \frac{C_i}{Q_i} \right\rceil P_i,$$

with $\lceil \cdot \rceil$ and $\lfloor \cdot \rfloor$ denoting the ceiling and floor functions, respectively. For reservation periods Q_i sufficiently small with respect to C_i, this results in t_i^P being approximated reasonably well by Eq. (2). □

This means that server i, under a (Q_i, P_i) reservation, behaves as if deployed on a virtual CPU that is slower by a factor of Q_i/P_i w.r.t. the CPU it is deployed onto. In practice, the period P can be set to as little values as a few tens of ms, but amounts smaller than $1ms$ would cause excessive scheduling overheads (and context switches).

Note that, if multiple tasks are being scheduled within each reservation using a second-level scheduler based on fixed-priority, as allowed by our HCBS scheduler, then the analysis is more complex [72] but a "fluid approximation" similar to the above Eq. (2) can still be used.

The second point of our theoretical analysis relates to the characterization of the processing performance of servers hit by Poissonian traffic: consider a server i under a (Q_i, P_i) HCBS reservation, with sufficiently small reservation period P_i and all reservations co-located on the same pCPU respecting condition in Eq. (1). If the server is hit by traffic with i.i.d. and exponentially distributed inter-arrival times with average rate λ_i, and characterized by i.i.d. and exponentially distributed processing times with average processing rate μ_i (when executing on a dedicated CPU without reservation), under the stability assumption $\lambda_i < \mu_i Q_i/P_i$, is characterized by the following average value and ϕth percentile of its response-time t_i^P distribution

$$E[t_i^P] = \frac{1}{\mu_i \frac{Q_i}{P_i} - \lambda_i}, \; P_\phi[t_i^P] = -\frac{ln(1-\phi)}{\mu_i \frac{Q_i}{P_i} - \lambda_i}. \tag{3}$$

Proof. It is well-known in the field of queueing theory [73] that, for a M/M/1 system with average arrival rate λ_i and average serving rate μ_i, the average and ϕth percentile of the response-time R_i distribution turns out to be

$$E[R_i] = \frac{1}{\mu_i - \lambda_i}, \; P_\phi[R_i] = -\frac{ln(1-\phi)}{\mu_i - \lambda_i}. \tag{4}$$

Now, considering that the M/M/1 queueing system is actually running within a (Q_i, P_i) HCBS reservation with the condition in Eq. (1) being satisfied, from the result in Eq. (2) above it is clear that the system can only process requests at the reduced average rate $\mu_i Q_i/P_i$, not the full rate μ_i. Therefore, the result in Eq. (3) easily follows. □

4.2.2 Transmission Times

The approximated times needed by requests and responses to reach their destinations (t_i^S and t_i^R) are computed by splitting them in queuing times (times during which network packets are queued waiting to be transmitted), transmission latencies (times needed by a packet to travel between client and server or vice-versa) and transmission times (times needed to transmit a packet of size z_i on a medium with speed σ). We can distinguish two cases depending on whether we can neglect network enqueueing times or not.

If the network is not congested, the transmission latencies are almost constant (and can be assumed to be equal to a constant δ), and the queuing and transmission times are negligible. Moreover, it is possible to employ end-to-end QoS/reservation techniques at the networking level, such as the well-known IntServ standard [74], or, considering the limited geographic and organizational extension of typical NFV infrastructures, time-triggered Ethernet [75], [76]. This way, applications may be granted precise guarantees on the

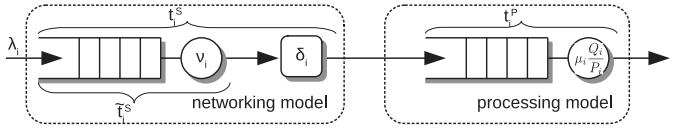


Fig. 3. Queueing model representing networking and processing.

experienced latency and available throughput on an end-to-end path. Therefore, it is still possible to use well-approximating constants δ and σ for the end-to-end latency and throughput experienced by a given application or VNF). Hence, the RTT can be approximated as $R_i^e = t_i^P + 2\delta$, and, considering the results in Eq. (3) above, its average and ϕth percentile can be approximated as

$$\begin{aligned} E[R_i^e] &= 2\delta + \frac{1}{\mu_i \frac{Q_i}{P_i} - \lambda_i}, \\ P_\phi[R_i^e] &= 2\delta - \frac{\ln(1-\phi)}{\mu_i \frac{Q_i}{P_i} - \lambda_i}. \end{aligned} \tag{5}$$

In the situation in which queuing times and transmission times are not negligible, instead, if we assume a per-server exponential distribution of the request sizes z_i, and their independence from the processing times, we can obtain the following approximations:

$$\begin{aligned} E[R_i^e] &= 2\delta + \frac{1}{\nu_i - \lambda_i} + \frac{1}{\mu_i \frac{Q_i}{P_i} - \lambda_i}, \\ P_\phi[R_i^e] &\leq 2\delta - \frac{\ln(1-\sqrt{\phi})}{\alpha_i}, \end{aligned} \tag{6}$$

where $\nu_i = \frac{\sigma}{E[z_i]}$ is the average transmission rate for the packets of server i and $\alpha_i \triangleq (\frac{1}{\nu_i - \lambda_i} + \frac{1}{\mu_i \frac{Q_i}{P_i} - \lambda_i})^{-1}$, under the stability assumptions of $\nu_i > \lambda_i$ and $\mu_i \frac{Q_i}{P_i} > \lambda_i$.

Proof. Assume an exponential distribution of the request sizes z_i^S, implying a similar distribution of the transmission times t_i^S with an average transmission rate $\nu_i = \frac{\sigma_i}{E[z_i^S]}$, and response-time approximated as $t_i^R \cong \delta_i$, as due to, e.g., sending back to the client just a success/error code (symmetrically, we can think of a $t_i^S \cong \delta_i$ and exponentially distributed z_i^R and t_i^R, as due to e.g., replying with data to a very short request packet). With exponentially distributed service times and under scheduling parameters (Q_i, P_i), and as condition in Eq. (1) is respected, Eq. (2) implies an average service rate of $\mu_i \frac{Q_i}{P_i}$.

Then, as depicted in Fig. 3, the system can be approximated as a sequence of two M/M/1 queues, which, under the stability condition of

$$\lambda_i < \nu_i \wedge \lambda_i < \mu_i \frac{Q_i}{P_i}, \tag{7}$$

has the steady-state behavior of two independent M/M/1 queues, with the process of arrivals at the server input queue being Poissonian with the same parameter λ_i. Therefore, denoting with $\tilde{t}_i^S$ the time needed for networking where the constant term δ_i has been removed, again, from well-known results on M/M/1 systems, we have

$$\begin{aligned} R_i^e &= 2\delta_i + \tilde{t}_i^S + t_i^P, \quad E[\tilde{t}_i^S] = \frac{1}{\nu_i - \lambda_i}, \quad E[t_i^P] = \frac{1}{\mu_i \frac{Q_i}{P_i} - \lambda_i} \\ &\Rightarrow E[R_i^e] = 2\delta_i + \frac{1}{\nu_i - \lambda_i} + \frac{1}{\mu_i \frac{Q_i}{P_i} - \lambda_i}. \end{aligned} \tag{8}$$

The computation of (a bound for) the percentile $P_\phi[R_i^e]$ is more involved: we start from the definition $P_\phi[R_i^e] = D \Leftrightarrow \Pr[R_i^e \leq D] = \phi$. As R_i^e is the sum of two exponential distributions with different parameters, we can easily get to a closed-form lower bound for the R_i^e cumulative distribution by splitting it proportionally to the expected values of the two components. Formally, using the easy-to-prove lower-bound

$$\Pr[X + Y \leq z] \geq \Pr\left[X \leq z\frac{E[X]}{E[X]+E[Y]} \wedge Y \leq z\frac{E[Y]}{E[X]+E[Y]}\right],$$

we have

$$\Pr[R_i^e \leq D] = \Pr[\tilde{t}_i^S + t_i^P \leq D - 2\delta_i] \geq \left(1 - e^{-\alpha_i(D-2\delta_i)}\right)^2, \tag{9}$$

where $\alpha_i \triangleq (\frac{1}{\nu_i - \lambda_i} + \frac{1}{\mu_i \frac{Q_i}{P_i} - \lambda_i})^{-1}$, and $\tilde{t}_i^S$ and t^P have been assumed to be i.i.d. and independent from one another. Now we can compute the minimum deadline D guaranteeing $\Pr[R_i^e \leq D] \geq \phi_i$:

$$\begin{aligned} \Pr[R_i^e \leq D] &\geq \left(1 - e^{-\alpha_i(D-2\delta_i)}\right)^2 \geq \phi_i \\ D &\geq 2\delta_i - \frac{\ln(1-\sqrt{\phi_i})}{\alpha_i}. \end{aligned} \tag{10}$$

Therefore, Eq. (6) easily follows. □

Note that the theoretical hypothesis of infinite transmission speed $\nu_i \to \infty$ leads to an expression of $P_\phi[R^e]$ similar to Eq. (5) where $\sqrt{\phi_i}$ replaces ϕ_i, providing an insight into the implications of the approximated bound of Eq. (9). Additional details on the above results can be found in [14].

5 Software Architecture

This section summarizes changes that have been designed and realized for the Tacker NFV management layer and the OpenStack framework to build a prototype for the model conceptualized in Section 4. The discussion refers to a simplified deployment of OpenStack that has been used, consisting of simply three physical nodes: a controller node, a compute node configured for instantiating VMs via KVM, and a compute node configured for instantiating containers through `lxc`. The setup has been deployed using Kolla, an automatic deployment framework built upon Ansible that deploys OpenStack services in separated Docker containers.

The reference development and deployment platform has been Linux Ubuntu 16.04 LTS, with OpenStack Rocky including Nova (management of physical compute nodes and VMs), Neutron (network management), Keystone (authentication), Glance (image provisioning), Ceilometer (monitoring and alarming), Heat (orchestration), Horizon (web-based dashboard), Tacker (NFV management), with

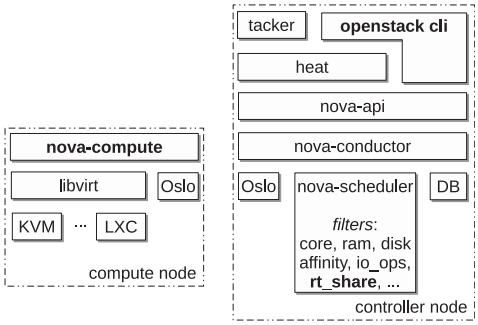


Fig. 4. Overview of modifications to OpenStack components.

OpenStack configured as Virtualized Infrastructure Management (VIM). Version 18.1.0 of Nova has been modified and deployed. All components run on the controller node, except for Nova, which also has additional parts running on the compute nodes. OpenStack has been configured to use MariaDB as the reference DataBase system (running on the controller node as well).

Fig. 4 highlights the major sub-components of the OpenStack/Nova architecture that are relevant for our work: `nova-conductor` handles interactions among Nova subcomponents; `nova-scheduler` allocates VMs to compute nodes through the use of allocation *filters*; `DB` is the DataBase (MariaDB, in our deployment scenario) used by OpenStack to manage persistent information; `oslo` is a set of libraries (used throughout OpenStack components) including logging, JSON processing, configuration management, and messaging services; `nova-compute` runs on each compute node to instantiate and manage VMs through `libvirt`, which is a component external to OpenStack used to handle various virtualization technologies (including Xen, KVM, `lxc` and `lxd` containers); `nova-api` runs on the controller node to accept requests from either the command-line client `openstack cli`, or the web-based dashboard Horizon.

Our modifications allowed us to deploy `lxc` containers, with attached a per-container `(runtime, period)` specification, over physical compute nodes with the HCBS hypervisor scheduler as described in Section 4.1. The components introduced above have been modified as follows:

- Each physical compute node has been provided with additional information on the available and used real-time computational bandwidth, adding the `rt_share` and `rt_share_used`columns in the `compute_nodes` table in the Nova DB schema;
- Each deployed instance has been provided with additional real-time scheduling parameters through `rt_runtime_us` and `rt_period_us`new columns in the `instance_extra` table of the DB schema;
- The scheduling parameters have been added as additional attributes of an OpenStack instance type (flavor), by using two new `extra_specs` attributes: `hw:cpu.rt_runtime_us` and `hw:cpu.rt_period_us`;
- The `cpu.rt_runtime_us` and `cpu.rt_period_us`per-instance attributes have been added to the scheduler hints that can be specified on instance creation;
- A new `scheduler.filters.RTShareFilter`, subclassing `BaseHostFilter`, has been added to the set of Nova host filters, that passes a compute node only if supports the HCBS hypervisor scheduler functionality and its residual `free_rt_share = rt_share - rt_share_used`is greater than or equal to the required `vcpus * rt_runtime_us / rt_period_us`;
- The `scheduler.HostState` Python class, representing a compute node resources availability within the Nova scheduler, has been extended with the `rt_share` and `rt_share_used`fields, as well as with the capability to update these fields based on the deployed instances;
- The `virt.libvirt.LibVirtDriver` Python class has been modified to apply the instance `rt_runtime_us` and `rt_period_us` parameters to the CPU `cgroup` just created for the instance;
- A new boolean option, `hcbs_scheduler_available`, has been added to the Nova configuration file. When set to false, the node will advertise a total `rt_share` of 0 in its `HostState`so that it will not be chosen by the `RTShareFilter`.

As a result, a new flavor with a scheduling runtime of, say, $100ms$ every period of $200ms$, 1 vcpu and 512 Mb of RAM can be created with:

```
  openstack flavor create flavor_rt1 --vcpus 1 --ram 512 \
--property hw:cpu.rt_runtime_us=100000 \
--property hw:cpu.rt_period_us=200000
```

Then, a new instance can either inherit the scheduling parameters specified in the flavor definition, or override them. For example, the following command overrides the default $100ms$ scheduling runtime of the flavor with $120ms$:

```
openstack server create --flavor flavor_rt1 \
--hint cpu.rt_runtime_us=120000
```

Going up in the software stack, with the presented set of changes, it is possible to specify real-time scheduling parameters within a Heat Orchestration Template file by either 1) using for the new instance a flavor with attached real-time scheduling parameters, or 2) specifying new real-time scheduling parameters as scheduler hints, which would anyway override the flavor parameters, if present.

5.1 Real-Time Reservations in MANO Descriptors

Standard MANO descriptors defined in the TOSCA specification allow for specifying the processing requirements of a VDU under deployment, as a `flavor` property of the VDU and/or via properties within the `nfv_compute` descriptor.

When using a flavor to specify the VDU requirements, it is easy to make the H-CBS real-time scheduling features accessible to the higher-layer NFV framework and MANO descriptors: once one has defined one or more flavors, with associated the desired scheduling parameters `cpu.rt_runtime_us` and `cpu.rt_period_us`, as described above, the flavor can be directly mentioned in a VDU specification, e.g.,:

```
topology_template:
  node_templates:
    VDU_A:
      type: tosca.nodes.nfv.VDU.Tacker
      properties:
        flavor: flavor_rt1
        ...
```

On the other hand, one can specify a VDU requirements also using directly the `nfv_compute` descriptor, whose type is `tosca.datatypes.compute_properties`. Using such a solution permits the specification of such requirements as: `num_cpus`, `mem_size`, `disk_size` and `cpu_allocation`, among others. The latter is a map of type `tosca...CPUAllocation`, including such attributes as:

- `socket_count`, `core_count`, `thread_count`: supplementary details on the desired topology connecting the required computational elements;
- `cpu_affinity`: distinguishes between virtual cores pinned down onto dedicated physical cores or allowed to migrate among shared physical cores;
- `thread_allocation`: defines the mapping of virtual cores onto hyper-threads available in the underlying physical host.

In this case, we support our new H-CBS scheduling features simply by extending the `CPUAllocation` map type by adding the ability to specify the `cpu.rt_runtime_us` and `cpu.rt_period_us` scheduler hints when setting up the underlying Heat Orchestration Template implementing the NFV specification.

For example, the following sample VDU template syntax can be used to request a container with 4 cores, scheduled under a real-time reservation of $60ms$ every $100ms$:

```
topology_template:
  node_templates:
    VDU_A:
      type: tosca.nodes.nfv.VDU.Tacker
      capabilities:
        nfv_compute:
          properties:
            num_cpus: 4
            mem_size: 4096 MB
            disk_size: 20 GB
            cpu_allocation:
              cpu.rt_period_us: 100000
              cpu.rt_runtime_us: 60000
```

5.2 Multi-Core Containers

When deploying complex software components able to leverage on the parallelism available on multi-core architectures, it is essential to support multi-core virtualized runtime environments properly. In cloud and NFV environments, we often need to deploy containers with possibly a limited intrinsic parallelism degree over considerably large multi-core physical systems with plenty of CPUs.

Therefore, in order to support fine-grained allocation and real-time scheduling of the available physical cores across many containers, we extended our real-time EDF-based scheduler for the Linux kernel: it is possible to specify a per-CPU runtime that is granted by the kernel to the container processes. It is also possible to specify a runtime of zero for one or more CPUs; this way, it is possible to restrict the CPUs each container is actually deployed onto.

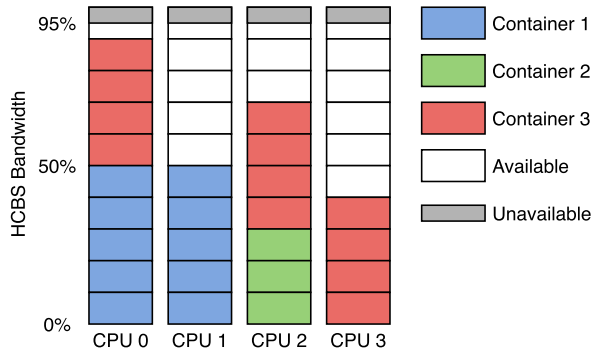


Fig. 5. Example allocation of multi-core containers in a 4-CPU scenario.

Our modifications to the OpenStack Nova and Nova-LXC include the possibility to use the information on the number of virtual cores a container is created with, along with the specified real-time scheduling runtime and period: then, the container is configured with associated a CPU reservation spanning across only a subset of the available physical CPUs, where the per-CPU runtime is set to the desired value, whilst on the other CPUs it is set to zero.

In order to support this feature, we had to realize an allocation policy within Nova that keeps track of how much real-time bandwidth has been allocated on each physical CPU of each host so that we can choose what CPUs each new container can be deployed onto, in order not to overcommit the real-time bandwidth available on each physical CPU (whose maximum is set to a default value of 95% that can be changed through a configuration parameter if desired). For the sake of simplicity, the adopted policy is "worst-fit", in which the CPUs with the highest available bandwidth are chosen.

For example, consider the case depicted in Fig. 5 in which there are 4 available CPU cores that initially have no HCBS containers, and three containers having the following computational requirements: 1) 50% bandwidth on 2 CPU cores; 2) 30% bandwidth on 1 CPU core; 3) 40% bandwidth on 3 CPU cores. The first container is allocated to the first two available CPUs (0,1), the second one to the first fully available CPU (2), the third one on the three CPUs with the highest available bandwidth (i.e., 3,2,0). The drawback of this simple allocation is that, for example, if a new single-core container with 75% bandwidth needs to be allocated, there would be no CPUs with enough available bandwidth. However, it could be possible to "make room" for the new real-time container by "defragmenting" the available real-time bandwidth moving existing containers around (in the example, it would be sufficient to move the reserved bandwidth of container 3 from CPU 3 to CPU 1). Our underlying HCBS supports dynamic changes of the assigned per-core runtime. However, for applications with tight timing requirements, migration of real-time reservations among cores should be done using an appropriate *mode-change protocol* [77], [78], in order not to disrupt their run-time behavior (essentially due to the additional L1 cache misses the migrated threads would incur on the new CPU). A typical way of doing this would be by leveraging on the moment a real-time task suspends waiting for I/O or for a timer to fire. Integrating this kind of logic in the framework, possibly leveraging on dynamic partitioning heuristics directly integrated within the scheduler [79], [80], is left as possible future work on the topic.

TABLE 2
Main Characteristics of the Hardware Platforms Used for the Experiments

Section	Component	CPU	RAM
6.2	DistWalk Server DistWalk Client	i5-4590S @ 3.00GHz	8GB
6.3	DistWalk Server DistWalk Client	i5-4590S @ 3.00GHz	8GB
6.4	Kamailio Server SIPp Client	i7-4790K @ 4.00GHz i7-7700HQ @ 2.80GHz	16GB

All CPUs are Intel(R) Core(TM) with the detailed model.

6 Experimental Results

In this section, the accuracy of the model presented in Section 4.2 in presence of the mechanism detailed in Section 4.1 will be evaluated, using our implementation as presented in Section 5. This is a different kind of performance evaluation with respect to the one performed by the most relevant previous works such as RT-OpenStack [18], that focused on sets of *independent* periodic real-time tasks.[9]

This section is organized as follows: Section 6.1 gives additional details about the experimental setup; Section 6.2 presents the results in the case of negligible networking time; Section 6.3 shows results in the case of constrained network bandwidth and exponential packet size; finally, Section 6.4 presents the results of a possible NFV application (i.e., a SIP connection manager).

6.1 Experimental Setup

In order to test the proposed configuration, an OpenStack deployment consisting of a compute node and a controller node is used (which is required by OpenStack, but does not influence the results of the experiments). The tested containers are deployed on the compute node, and they host the application server. The client is hosted by a third node outside the OpenStack deployment. In the experiments of Sections 6.2 and 6.3, we used DistWalk,[10] an open-source distributed application able to impose a configurable client-server networking traffic and processing workload [23]. In Section 6.4, the open-source Kamailio SIP server and `SIPp` SIP client were used.

Different machines have been used throughout the experiments shown in the subsections that follow, with the hardware characteristics detailed in Table 2.

Moreover, the compute node ran a modified version of Linux with the HCBS patches in all experiments. The server reservation period is always set to $P = 2ms$, a small enough value that enables the approximation of Eq. (2). Also, for simplicity, the server has been constrained to use only one CPU. Compute and client nodes have been connected through an L2 switch with $1Gbps$ Ethernet. A token bucket traffic shaper (from the `tc` tool) was used to limit network bandwidths below $1Gbps$. The token bucket was set with the smallest possible buffer size[11] and its latency[12] has been set to $100ms$ to prevent excessive packet drops.

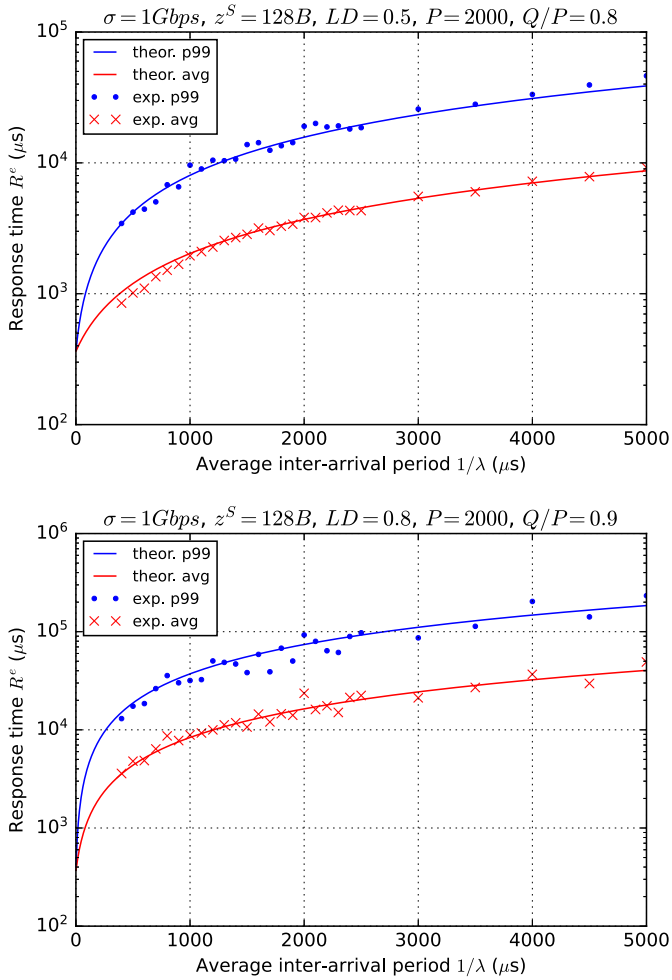


Fig. 6. Experimental response-time statistics (markers) obtained varying the inter-arrival period, at a CPU load of $LD = 0.5$ (top plot) and $LD = 0.8$ (bottom plot). Lines represent the theoretical expectations of Eq. (5).

Furthermore, in both machines, hyper-threading and CPU frequency switching have been disabled to increase accuracy. At the same time, high-resolution timers and HRTICK[13] have been enabled, and HZ[14] has been set to $1ms$, for the same reason. In what follows, all measurements have been obtained with 10000 samples per run.

6.2 Negligible Networking Time

In this first set of experiments, sending and receiving times are negligible (Eq. (5)). Two configurations of computational workload and reservation have been tested, varying the average inter-arrival time of requests from $100\mu s$ to $5ms$. In order to assume negligible sending and receiving times with regards to processing time, the packet size was set to $z^S = 128$. In this experiment, the average client-server latency was $\delta = 363/2 = 181.5\mu s$, as measured by gathering

9. This paper, instead, focuses on applications composed of real-time tasks interacting through a client-server paradigm. We already verified the correct scheduling of independent periodic real-time tasks in previous papers [17].

10. Available on GitHub: https://github.com/tomcucinotta/distwalk.

11. $buffer = min(mtu, \frac{rate}{HZ})$, since the timer resolution is $1/HZ$.

12. i.e., the maximum amount of time a packet is allowed to stay in the token bucket before being dropped.

13. High-resolution timers allow for precise preemption points.

14. HZ is the rate of the kernel timer used, e.g., by the scheduler tick.

TABLE 3
Model Accuracy with Regards to Experiments for Both Average and 99th Percentile, Expressed as Root Mean Squared Error (RMSE) and Mean Absolute Percentage Error (MAPE), for Multiple Configurations of Packet Size (z^S), CPU Load (LD) and Computational Bandwidth (Q/P)

z^S (B)	LD	Q/P	Average		99th Percentile	
			RMSE	MAPE	RMSE	MAPE
128	0.5	0.8	171.53	5.41%	2195	8.79%
128	0.7	0.9	515.24	5.02%	3477.3	9.57%
128	0.8	0.9	2995.3	10.37%	18875	15.71%
2048	0.5	0.8	198.6	8.01%	1363.7	6.75%
2048	0.7	0.9	657.88	5.57%	6938.3	15.93%
2048	0.8	0.9	2251.2	13.01%	14528	21.26%

10K samples with `ping`. Both compute and client nodes ran on an identical machine as detailed in Table 2.

Fig. 6 shows the experiment results, where markers represent the obtained average and the 99th percentiles of the response-times, while the continuous lines represent the respective theoretical values. In these plots, an increment of the average inter-arrival times (X-axis) indicates a corresponding raise of the response times since the average processing times on the server are also increased due to the constant computational workload (LD).

The figure shows that the response-time distribution statistics match quite closely with the expectation coming from the theoretical model. This can also be seen from the table in Table 3 where RMSE and MAPE are reported for different configurations. We can notice how the best accuracy is achieved at lower load ratios ($LDR = (LD)/(Q/P)$), e.g., in the top plot where $LDR = 0.5/0.8 = 0.625$. Increasing LDR, the accuracy decreases, e.g., in the bottom plot where $LDR = 0.8/0.9 \cong 0.889$. This behavior can be explained as we are approaching the instability region ($LDR > 1$).

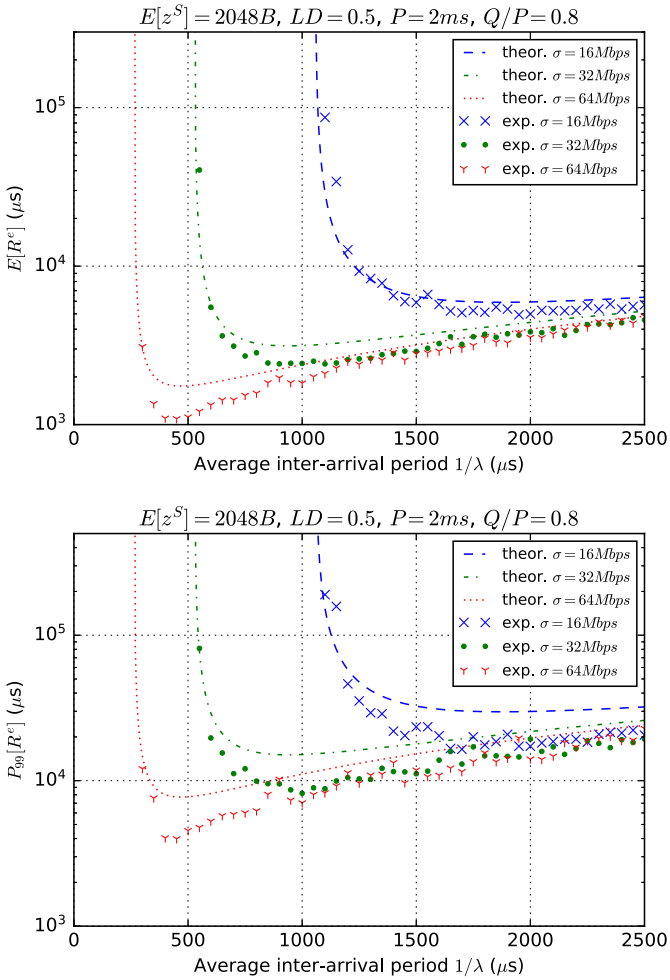


Fig. 7. Response-time statistics (top: average, bottom: 99th percentile) at varying average inter-arrival periods (on the X axis) and network bandwidths (different curves). Lines represent the theoretical expectations (note that the theoretical 99th percentile is a conservative approximation), while markers show experimental results.

6.3 Non-Negligible Networking Time With Exponentially Distributed Packet Sizes

The next set of experiments, taken from [23], shows the results under a constrained network bandwidth with exponentially distributed packet sizes. The average latency and experimental setup are the same as in the previous section.

Fig. 7 compares the obtained average and 99th percentile of the response times with different network bandwidths. Notice that a decrement in the average inter-arrival times (shown on the X axis) does not always cause a decrease in response time as in the previous case. Instead, starting from a particular value of the inter-arrival period, the response times increase because packets get queued, causing an increase in the network delays that are higher than the benefit from reducing the processing times.

Experimental points for the average fall slightly below the expected value. This can be explained by the fact that we are simulating a lower bandwidth using a token bucket, which lets packets through at link speed if there are enough tokens, e.g., when the load is low as in the right part of the plot. In fact, the experimental points follow the line of 1Gbps before rising up when the load increases.

Also in the 99th percentile plot values are below the theoretical line, but this time it was expected, since it is a conservative approximation. Furthermore, the figure reveals that the approximation is less accurate near the "bending point" of the curve, where the distance between the theoretical line and the points increases. However, just after this "bending point", approaching the instability region $BWR > 1$, it can be seen that a number of experimental response times statistics exceed the predicted values, confirming that the model has some weaknesses at high loads.

6.4 IMS Test Case

The experiment in this section, taken from [14], is used to evaluate the performance of a SIP connection manager, broadly adopted for VoIP applications, which is a possible NFV application that may benefit from the proposed approach. We picked Kamailio[15] to handle SIP traffic produced by the SIPp[16] tool, and compared the experimental cumulative distribution functions (ECDF) of the RTTs.

In this experiment, the average client-server latency was $\delta = 252/2 = 126\mu s$, as measured gathering 10K samples

15. More details available at: http://kamailio.org/.
16. More details available at: http://sipp.sourceforge.net/.

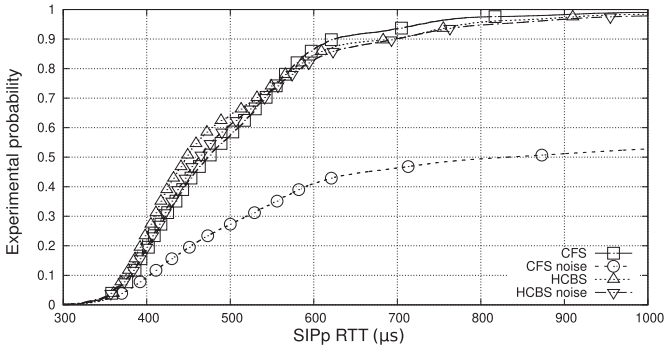


Fig. 8. ECDF plots of SIPp authentication RTT using the CFS algorithm and HCBS, with and without interfering tasks.

using `ping`. The application server and client were deployed on machines with the hardware characteristics detailed in Table 2. The client executes SIPp as a high-priority `SCHED_FIFO` task to prevent interference with other running tasks, while the server hosts the Kamailio and Mysql in a 1-vCPU container. The SIPp configuration fixes the number of registrations every $100ms$ to 257, and requests the reporting to an output file of the measured round-trip time for each request, which involves the interaction with the Mysql database and two packet round-trips.

Fig. 8 presents the results for the following cases: *CFS*, the default Linux scheduler, and *HCBS*, our proposed approach, both in the case of no interfering process (no noise) or with noise. In *CFS*, the CPU time is uniformly split among the active tasks, allowing Mysql and Kamailio to exploit all the time of the underlying pCPU. In *HCBS*, Mysql and Kamailio are run within a HCBS container as `SCHED_RT` tasks. The HCBS container is configured with a runtime of $1.4ms$ and a period of $2ms$, which is enough to satisfy the average load (around 60%) with some safety margin. In both cases, the *noise* plots correspond to the scenario in which other four periodic rt-app[17] tasks are running in the system for $6ms$ every $100ms$. In the case of *HCBS noise*, these are executed within another HCBS server as `SCHED_RR` tasks, with runtime of $24ms$ and a period of $100ms$. These tasks have a strong effect on the resulting performance of the system since they collectively claim 24% of the pCPU time. As a result, less than 70% of SIP RTTs are below $2ms$.

The average performance of *HCBS* is slightly better than *CFS*, since `SCHED_DEADLINE` is the highest priority scheduling class, but the 99th percentile obtained with *HCBS* is higher than the one obtained with *CFS* ($1287\mu s$ versus $980\mu s$). This result can be explained by the throttling that the `SCHED_DEADLINE` entity receives when its budget is exhausted. In the case of *HCBS noise*, the ECDF is only marginally influenced by the rt-app, excluding a little overhead introduced by an increased number of context switches, thanks to the reservation mechanism. Its 99th percentile is equal to $1268\mu s$ (higher than *HCBS* due to measurement noise), *outperforming CFS noise* that is able to reach that percentile *one order of magnitude later*, at $10614\mu s$.

Summarizing, the shown experimental results highlight that an IMS signalling application may exhibit very unstable and degraded response times in presence of co-located "noisy" containers when under a standard scheduling policy. On the other hand, the use of our proposed HCBS scheduler keeps the performance more stable and predictable, allowing for the design and deployment of NFV services with a reliable performance and timing behavior.

7 Conclusions and Future Work

This paper introduced a novel approach for resource allocation to NFV services based on real-time scheduling of co-located containers, and its prototype implementation based on OpenStack with Tacker, leveraging a patch to the Linux kernel. The proposed architecture provides stable and predictable QoS. This has been showcased building a model based on queueing networks of a synthetic application, where experimental results matched with theoretical expectations at low-to-moderate overall system utilizations, while also highlighting some limitations of the modeling framework mostly visible at high loads. The proposed solution has been demonstrated to be effective with a real IMS use-case based on the Kamailio server and the SIPp client.

As future work, we plan a better integration with NFV MANO standards, experimentation on common software components for access networks, such as OpenAirInterface,[18] extensions to our theoretical analysis removing parts of the assumptions, and supporting differentiated QoS mechanisms in accessing cloud storage services such as [81], often involved in end-to-end NFV scenarios.

References

[1] R. Buyya, C. Vecchiola, and S. T. Selvi, *Mastering Cloud Computing: Foundations and Applications Programming*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Pub. Inc., 2013.

[2] S. Mathew, "Overview of Amazon Web Services – AWS whitepaper," AWS, Seattle, WA, USA, White Paper, Apr. 2021. [Online]. Available: https://d1.awsstatic.com/whitepapers/aws-overview.pdf

[3] Google, "Compare AWS and Azure services to Google Cloud," Apr. 2021. [Online]. Available: https://cloud.google.com/free/docs/aws-azure-gcp-service-comparison

[4] M. Stolic, J. A. Connary, W. Yan, A. Mukherjee, M. Salaheldin, and R. Piasecki, "5G automation architecture white paper," Cisco, San Jose, CA, USA, White Paper, 2020. [Online]. Available: https://www.cisco.com/c/dam/m/en_us/customer-experience/collateral/5G-automation-architecture-white-paper.pdf

[5] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," Special Publication (NIST SP), Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Sep. 28, 2011.

[6] W. John *et al.*, "The future of cloud computing: Highly distributed with heterogeneous hardware," May 2020. [Online]. Available: https://www.ericsson.com/499e1f/assets/local/reports-papers/ericsson-technology-review/docs/2020/the-future-of-cloud-computing.pdf

[7] NFV Industry Specif. Group, "Network functions virtualisation," ETSI, Sophia Antipolis, France, Introductory White Paper, 2012.

[8] C. Vitucci and A. Larsson, "Flexible 5G edge server for multi industry service network," *Int. J. Adv. Netw. Serv.*, vol. 10, no. 3–4, pp. 55–65, 2017.

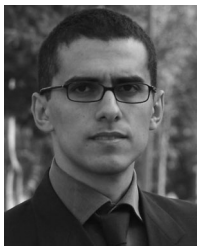
[9] F. Voigtländer, A. Ramadan, J. Eichinger, C. Lenz, D. Pensky, and A. Knoll, "5G for robotics: Ultra-low latency control of distributed robotic systems," in *Proc. Int. Symp. Comput. Sci. Intell. Controls*, Oct. 2017, pp. 69–72.

17. See: https://github.com/scheduler-tools/rt-app/.

18. More information at: http://www.openairinterface.org/

[10] J. Ordonez-Lucena , P. Ameigeiras, D. Lopez, J. J. Ramos-Munoz , J. Lorca, and J. Folgueira, "Network slicing for 5G with SDN/NFV: Concepts, architectures, and challenges," *IEEE Commun. Mag.*, vol. 55, no. 5, pp. 80–87, May 2017.
[11] ETSI, "GS NFV-MAN 001 V1.1.1: Network Functions Virtualisation (NFV); management and orchestration," Dec. 2014. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/nfv-man/001_099/001/01.01.01_60/gs_nfv-man001v010101p.pdf
[12] M. Bogo, J. Soldani, D. Neri, and A. Brogi, "Fine-grained management of cloud-native applications, based on TOSCA," *Internet Technol. Lett.*, vol. 3, no. 5, 2020 Art. no. e212.
[13] J. Anderson, H. Hu, U. Agarwal, C. Lowery, H. Li, and A. Apon, "Performance considerations of network functions virtualization using containers," in *Proc. Int. Conf. Comput., Netw. Commun.*, Feb. 2016, pp. 1–7.
[14] T. Cucinotta, L. Abeni, M. Marinoni, A. Balsini, and C. Vitucci, "Reducing temporal interference in private clouds through real-time containers," in *Proc. IEEE Int. Conf. Edge Comput.*, 2019, pp. 124–131.
[15] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," *SIGOPS Operatig Syst. Rev.*, vol. 41, no. 3, pp. 275–287, Mar. 2007.
[16] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," *IEEE Trans. Cloud Comput.*, vol. 7, no. 3, pp. 677–692, Jul.–Sep. 2019.
[17] L. Abeni, A. Balsini, and T. Cucinotta, "Container-based real-time scheduling in the Linux kernel," *SIGBED Rev.*, vol. 16, no. 3, pp. 33–38, Nov. 2019.
[18] S. Xi *et al.*, "RT-OpenStack: CPU resource management for real-time cloud computing," in *Proc. IEEE 8th Int. Conf. Cloud Comput.*, 2015, pp. 179–186.
[19] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: Towards real-time hypervisor scheduling in Xen," in *Proc. 9th ACM Int. Conf. Embedded Softw.*, 2011, pp. 39–48.
[20] ETSI, "Network Functions Virtualisation (NFV) Release 4; Management and Orchestration; Requirements for service interfaces and object model for OS container management and orchestration specification," Nov. 2020. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV-IFA/001_099/040/04.01.01_60/gs_NFV-IFA040v040101p.pdf
[21] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. 10th Eur. Conf. Comput. Syst.*,Apr. 2015, pp. 1–17.
[22] T. Cucinotta, M. Marinoni, A. Melani, A. Parri, and C. Vitucci, "Temporal isolation among LTE/5G network functions by real-time scheduling," in *Proc. 7th Int. Conf. Cloud Comput. Serv. Sci.*, Funchal, Madeira, Portugal, Apr. 2017, pp. 368–375.
[23] R. Mancini ., T. Cucinotta ., and L. Abeni ., "Performance modeling in predictable cloud computing," in *Proc. 10th Int. Conf. Cloud Comput. Serv. Sci.*, 2020, pp. 69–78.
[24] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli, "Deadline scheduling in the Linux kernel," *Softw.: Pract. Experience*, vol. 46, no. 6, pp. 821–839, 2016.
[25] A. Maaref, J. Ma, M. Salem, H. Baligh, and K. Zarin, "Device-centric radio access virtualization for 5G networks," in *Proc. IEEE Globecom Workshops*, Dec. 2014, pp. 887–893.
[26] X. Costa-Perez, J. Swetina, T. Guo, R. Mahindra, and S. Rangarajan, "Radio access network virtualization for future mobile carrier networks," *IEEE Comm. Mag.*, vol. 51, no. 7, pp. 27–35, Jul. 2013.
[27] B. Haberland *et al.*, "Radio base stations in the cloud," *Bell Labs Tech. J.*, vol. 18, no. 1, pp. 129–152, Jun. 2013.
[28] V. Q. Rodriguez and F. Guillemin, "Performance analysis of VNFs for sizing cloud-RAN infrastructures," in *Proc. IEEE Conf. Netw. Function Virtualization Softw. Defined Netw. (NFV-SDN)*, 2017, pp. 1–6.
[29] 3GPP, "3rd generation partnership project; Transport requirement for CU-DU functional splits options; R3–161813 (document for discussion)," 3GPP, Sophia Antipolis, France, *Rep. R3-162005*, Aug. 2016.
[30] T. Cucinotta, L. Abeni, M. Marinoni, and C. Vitucci, "The importance of being os-aware - In performance aspects of cloud computing research," in *Proc. 8th Int. Conf. Cloud Comput. Serv. Sci.*, Mar. 2018, pp. 626–633.
[31] M. Navrátil, L. Bailey, and C. Boyle, "Red Hat Enterprise Linux 7 performance tuning guide," Jun. 2020. [Online]. Available: https://tinyurl.com/24nxfdjk
[32] G. Ara, T. Cucinotta, L. Abeni, and C. Vitucci, "Comparative evaluation of kernel bypass mechanisms for high-performance Inter-container Communications," in *Proc. 10th Int. Conf. Cloud Comput. Serv. Sci.*, 2020, pp. 44–55.
[33] R. Kawashima, H. Nakayama, T. Hayashi, and H. Matsuo, "Evaluation of Forwarding efficiency in NFV-Nodes Toward predictable service chain performance," *IEEE Trans. Netw. Service Manage.*, vol. 14, no. 4, pp. 920–933, Dec. 2017.
[34] Data Plane Development Kit (DPDK). Accessed: Sep. 5, 20149. [Online]. Available: https://www.dpdk.org/
[35] L. Rizzo and M. Landi, "netmap: Memory mapped access to network devices," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 422–423, Oct. 2011.
[36] G. Lettieri, V. Maffione, and L. Rizzo, "A survey of fast packet I/O technologies for network function virtualization," in *Lecture Notes in Computer Science*. Berlin, Germany: Springer., 2017, pp. 579–590.
[37] D. Cotroneo, L. De Simone, and R. Natella, "NFV-Bench: A dependability benchmark for network function virtualization systems," *IEEE Trans. Netw. Serv. Manage.*, vol. 14, no. 4, pp. 934–948, Dec. 2017.
[38] F. Checconi, T. Cucinotta, D. Faggioli, and G. Lipari, "Hierarchical multiprocessor CPU reservations for the Linux kernel," in *Proc. 5th Int. Workshop Operating Syst. Platforms Embedded Real-Time Appl.*, Jun. 2009, pp. 1–8.
[39] J. Lee *et al.*, "Realizing compositional scheduling through virtualization," in *Proc. IEEE 18th Real Time Embedded Technol. Appl. Symp.*, Apr. 2012, pp. 13–22.
[40] Y. Li, L. T. X. Phan, and B. T. Loo, "Network functions virtualization with soft real-time guarantees," in *Proc. IEEE INFOCOM 35th Annual IEEE Int. Conf. Comput. Commun.*, Apr. 2016, pp. 1–9.
[41] T. Cucinotta *et al.*, "Virtualised e-Learning with real-time guarantees on the IRMOS platform," in *Proc. IEEE Int. Conf. Serv.-Oriented Comput. Appl.*, Dec. 2010, pp. 1–8.
[42] M. Kessler, A. Reifert, D. Lamp, and T. Voith, "A service-oriented infrastructure for providing virtualized networks," *Bell Labs Tech. J.*, vol. 13, no. 3, pp. 111–127, Fall 2008.
[43] K. Konstanteli, T. Cucinotta, K. Psychas, and T. A. Varvarigou, "Elastic admission control for federated cloud services," *IEEE Trans. Cloud Comput.*, vol. 2, no. 3, pp. 348–361, Jul.–Sep. 2014.
[44] K. Konstanteli, T. Cucinotta, K. Psychas, and T. Varvarigou, "Admission control for elastic cloud services," in *Proc. IEEE 5th Int. Conf. Cloud Comput.*, 2012, pp. 41–48.
[45] J. Bhimani, Z. Yang, M. Leeser, and N. Mi, "Accelerating big data applications using lightweight virtualization framework on enterprise cloud," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2017, pp. 1–7.
[46] H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure devops," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2016, pp. 202–211.
[47] G. Li, H. Zhou, B. Feng, Y. Zhang, and S. Yu, "Efficient provision of service function chains in overlay networks using reinforcement learning," *IEEE Trans. Cloud Comput.*, early access, Dec. 23, 2019, doi: 10.1109/TCC.2019.2961093.
[48] K. Rzadca *et al.*, "Autopilot: Workload autoscaling at Google scale," in *Proc. 15th Eur. Conf. Comput. Syst.*, Apr. 2020, pp. 1–16.
[49] G. Kousiouris, T. Cucinotta, and T. Varvarigou, "The effects of scheduling, workload type and consolidation scenarios on virtual machine performance and their prediction through optimized artificial neural networks," *J. Syst. Softw.*, vol. 84, no. 8, pp. 1270–1291, 2011.
[50] V. Millnert, E. Bini, and J. Eker, "Cost minimization of network services with buffer and end-to-end deadline constraints," *SIGBED Rev.*, vol. 14, no. 4, pp. 39–45, Jan. 2018.
[51] H. R. Faragardi, S. Dehnavi, T. Nolte, M. Kargahi, and T. Fahringer, "An energy-aware resource provisioning scheme for real-time applications in a cloud data center," *Softw.: Pract. Experience*, vol. 48, no. 10, pp. 1734–1757, 2018.
[52] S. Sotiriadis, N. Bessis, and R. Buyya, "Self managed virtual machine scheduling in cloud systems," *Inf. Sci.*, vol. 433–434, pp. 381–400, 2018.
[53] D. T. Nguyen, K. K. Nguyen, S. Khazri, and M. Cheriet, "Real-time optimized NFV architecture for internetworking WebRTC and IMS," in *Proc. 17th Int. Telecommun. Netw. Strategy Planning Symp.*, Sep. 2016, pp. 81–88.
[54] R. Bolla, R. Bruschi, F. Davoli, and J. F. Pajo, "A model-based approach towards real-time analytics in NFV infrastructures," *IEEE Trans. Green Commun. Netw.*, vol. 4, no. 2, pp. 529–541, Jun. 2020.
[55] *The Open Group Base Specifications Issue 6*, IEEE Std 1003.1, 2004 Edition, The IEEE and The Open Group, 2004.

[56] C. L. Liu and J. Layland, "Scheduling alghorithms for multiprogramming in a hard real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.
[57] J. Corbet. "SCHED_FIFO and realtime throttling," Sep. 2008. [Online]. Available: https://lwn.net/Articles/296419/
[58] Real-Time group scheduling. Apr. 2017. [Online]. Available: https://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt
[59] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proc. IEEE Real-Time Syst. Symp.*, Madrid, Spain, Dec. 1998, pp. 4–13.
[60] M. L. Dertouzos, "Control robotics: The procedural control of physical processes," *Inf. Process.*, vol. 74, pp. 807–813, 1974.
[61] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, 1974.
[62] P. Barham *et al.*, "Xen and the art of virtualization," *SIGOPS Operating Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003.
[63] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: A new facility for resource management in server systems," in *Proc. 3rd Symp. Operating Syst. Des. Implementation*, 1999, vol. 99, pp. 45–58.
[64] C. Vitucci, T. Cucinotta, R. Mancini, and L. Abeni, "Implementation and deployment of a server at the edge using openstack components," in *Proc. 19th Int. Conf. Netw.*, Apr. 2020, pp. 22–29.
[65] F. Callegati, W. Cerroni, C. Contoli, R. Cardone, M. Nocentini, and A. Manzalini, "SDN for dynamic NFV deployment," *IEEE Commun. Mag.*, vol. 54, no. 10, pp. 89–95, Oct. 2016.
[66] X. Ge *et al.*, "OpenANFV: Accelerating network function virtualization with a consolidated framework in openstack," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 353–354.
[67] R. Mijumbi, J. Serrat, J. Gorricho, S. Latre, M. Charalambides, and D. Lopez, "Management and orchestration challenges in network functions virtualization," *IEEE Commun. Mag.*, vol. 54, no. 1, pp. 98–105, Jan. 2016.
[68] M. Paolino, J. Fanguède, N. Nikolaev, and D. Raho, "Turning an open source project into a carrier grade vswitch for NFV: Vosyswitch challenges results," in *Proc. IEEE Int. Conf. Netw. Infrastruct. Digit. Content (IC-NIDC)*, 2016, pp. 22–27.
[69] B. Han, V. Gopalakrishnan, G. Kathirvel, and A. Shaikh, "On the resiliency of virtual network functions," *IEEE Commun. Mag.*, vol. 55, no. 7, pp. 152–157, Jul. 2017.
[70] A. Lebre, J. Pastor, A. Simonet, and F. Desprez, "Revising openstack to operate fog/edge computing infrastructures," in *Proc. IEEE Int. Conf. Cloud Eng.*, Apr. 2017, pp. 138–148.
[71] M. Kerrisk . "Namespaces in operation, part 1: namespaces overview," Jan. 2013. [Online]. Available: https://lwn.net/Articles/531114/
[72] I. Shin and I. Lee, "Compositional real-time scheduling framework," in *Proc. 25th IEEE Int. Real-Time Syst. Symp.*, Dec. 2004, pp. 57–67.
[73] A. Willig, "A short introduction to queueing theory," Jul. 1999. [Online]. Available: http://www.cs.ucf.edu/~lboloni/Teaching/EEL6785_Fall2010/slides/QueueingTheory.pdf
[74] J. T. Wroclawski, "The Use of RSVP with IETF integrated services," RFC 2210, Fremont, CA, USA, Sep. 1997. [Online]. Available: https://rfc-editor.org/rfc/rfc2210.txt
[75] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer, "The time-triggered Ethernet (TTE) design," in *Proc. 8th IEEE Int. Symp. Object-Oriented Real-Time Distrib. Comput.*, 2005, pp. 22–33.
[76] C. Liu, F. Li, G. Chen, and X. Huang, "TTEthernet transmission in software-defined distributed robot intelligent control system," *Wirel. Commun. Mobile Comput., Special Issue Softw.-Defined Ind. Internet Things*, vol. 2018, pp. 1–13, Jul. 2018. [Online]. Available: https://www.hindawi.com/journals/wcmc/2018/8589343/
[77] J. Real and A. Crespo, "Mode change protocols for real-time systems: A survey and a new proposal," *Real-Time Syst.*, vol. 26, pp. 161–197, 2004.
[78] L. Chen and L. T. X. Phan, "SafeMC: A system for the design and evaluation of mode-change protocols," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2018, pp. 105–116.
[79] L. Abeni and T. Cucinotta, "EDF scheduling of real-time tasks on multiple cores: Adaptive partitioning vs. global scheduling," *SIGAPP Appl. Comput. Rev.*, vol. 20, no. 2, pp. 5–18, Jul. 2020.
[80] A. Mascitti, T. Cucinotta, and L. Abeni, "Heuristic partitioning of real-time tasks on multi-processors," in *Proc. IEEE 23rd Int. Symp. Real-Time Distrib. Comput.*, 2020, pp. 36–42.
[81] R. Andreoli, T. Cucinotta, and D. Pedreschi, "RT-MongoDB: A NoSQL database with differentiated performance," in *Proc. 11th Int. Conf. Cloud Comput. Serv. Sci.*, 2021, pp. 77–86.



Tommaso Cucinotta received the MSc degree in computer engineering from the University of Pisa, Italy, and the PhD degree in computer engineering from Scuola Superiore Sant'Anna (SSSA), Pisa. He has been investigating on real-time scheduling for soft real-time and multimedia applications, and predictability in infrastructures for cloud computing and NFV with SSSA. He has been MTS in Bell Labs, Dublin, Ireland, investigating on security and real-time performance of cloud services. He has been a software engineer with Amazon Web Services, Dublin, Ireland, where he worked on improving the performance and scalability of DynamoDB. Since 2016, he has been an associate professor with SSSA and the head of Real-Time Systems Lab, since 2019.

Luca Abeni received the MSc degree in computer engineering from the University of Pisa and the PhD degree from Scuola Superiore Sant'Anna (SSSA) in 2002. He is investigating on real-time operating systems and scheduling algorithms for QoS control in multimedia with SSSA. From 2003 to 2006, he was with Broadsat S.R.L., developing IPTV applications and audio or video streaming solutions over wired and satellite (DVB - MPE) networks. Then, as a full-time researcher he joined the University of Trento , where he became an associate professor. Since 2017, he has been an associate professor with Real-Time Systems Lab, SSSA.

Mauro Marinoni received the MSc and PhD degrees in computer engineering from the University of Pavia, Italy, in 2003 and 2007, respectively. He is currently a research affiliate with Scuola Superiore Sant'Anna. Since 2007, he has been with the Real-Time Systems Lab, where he was an assistant professor from 2009 to 2020. He has been local coordinator of the FP7 JUNIPER and the Eurostars RETINA projects, and has been involved in several industrial projects in various domains.

Riccardo Mancini received the BSc degree in computer engineering from the University of Pisa, where he is currently working toward the MSc degree in computer engineering. He is currently a Honors student with SSSA. Since 2019, he has been collaborates with the RETIS investigating on predictability in cloud computing. His research interests include cloud computing, GP-GPU acceleration, and big data.

Carlo Vitucci received the MSc degree in telecom engineering from the University of Rome, Italy. In 1996, he joined Ericsson R&D in Rome as an embedded firmware designer, working with real-time operating systems and middleware architectures. In 2008, he joined Quixant, a gaming company, as a specialist embedded software designer. Since 2014, he has been with Ericsson, Stockholm, as senior embedded software architect. His research interests include next-generation networks and fault-management systems.