Performance Modeling in Predictable Cloud Computing

Riccardo Mancini, Tommaso Cucinotta[a] and Luca Abeni[b]
Scuola Superiore Sant'Anna, Pisa, Italy

Keywords: Cloud Computing, Real-time Scheduling, Temporal Isolation, Performance Modeling, Network Function Virtualization.

Abstract: This paper deals with the problem of performance stability of software running in shared virtualized infrastructures. The focus is on the ability to build an abstract performance model of containerized application components, where real-time scheduling at the CPU level, along with traffic shaping at the networking level, are used to limit the temporal interferences among co-located workloads, so as to obtain a predictable distributed computing platform. A model for a simple client-server application running in containers is used as a case-study, where an extensive experimental validation of the model is conducted over a testbed running a modified OpenStack on top of a custom real-time CPU scheduler in the Linux kernel.

1 INTRODUCTION

The relentless evolution of information and communication technologies brought to a wide diffusion of cloud computing technologies (Armbrust et al., 2010) in the last decade. These are being adopted more and more in industrial and commercial settings, for various reasons. For example, they enable flexible and efficient use of hardware resources that are multiplexed over multiple customers (tenants). Moreover, they decrease the problems related to hardware obsolescence and the need of having expensive data centers operated by specialized personnel, sized for peak-hour workloads. As a result, the ICT (Information and Communications Technology) infrastructure and services, operated 24/7, can be rented on-demand as needed.

Not only *public cloud* providers are seeing a continuous growth of their business, but *private cloud* computing is also emerging as an increasingly used paradigm within an organization. This allows to optimize the use of ICT infrastructures by multiplexing a number of heterogeneous services on the same physical infrastructure providing computing, networking and storage services. In this context, *hybrid cloud* computing is emerging as an increasingly interesting solution taking the best of the two approaches (Armbrust et al., 2010).

A set of technologies that played a key and enabling role in cloud computing have been the ones related to *virtualization* of resources. *Disk virtualization* has been historically used to support disk partitioning, aggregation and reliability (think of RAID) in mainframes and personal computers. *Network virtualization* has been at the heart of employing separation and security in networking for data centers. *Machine virtualization* has been greatly leveraged to build nowadays' cloud services able to host a number of heterogeneous Operating Systems (OSes) on the same physical nodes. Virtualization technologies pose the foundation for a flexible and adaptable use of the underlying infrastructure, enabling seamless migration of virtual machines and services throughout the physical infrastructure as needed, including the possibility to *live migrate* them, causing unnoticeable down times in the range of a few hundred milliseconds.

1.1 Problem Presentation

This explosion in the use and adoption of cloud computing services led to a corresponding evolution of the users' needs. Nowadays, cloud services are not only used for storage or batch activities, but they are increasingly used for on-line and (soft) real-time applications, where customers and users of the shared infrastructure exhibit higher and higher requirements in terms of *responsiveness* and *timeliness* of the hosted applications and services.

While the use of virtualization leads to the increase in flexibility and security (Ardagna et al.,

[a] https://orcid.org/0000-0002-0362-0657
[b] https://orcid.org/0000-0002-7080-9601

Mancini, R., Cucinotta, T. and Abeni, L.
Performance Modeling in Predictable Cloud Computing.
DOI: 10.5220/0009349400690078
In *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*, pages 69-78
ISBN: 978-989-758-424-4
Copyright © 2020 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

2015) of multi-tenant cloud infrastructures, it hurts performance due to its associated overheads. This problem, particularly nasty in machine virtualization, has been tackled either in hardware by adding acceleration capabilities available in *hardware-assisted virtualization*, or in software by employing solutions based on *para-virtualization*, requiring modifications and customization of the guest OS. More recently, an increasingly interesting alternative of the latter kind is the one of using lightweight OS-level virtualization, a.k.a., *containers*. These are effectively an extension of an OS kernel services with additional encapsulation capabilities that let users run completely independent user-space software stacks. While multiple containers on the same physical machine share the same OS kernel, achieving lower security and resilience levels when compared to machine virtualization, they are also capable of accessing the underlying hardware at bare-metal performance. As a result, containerization technologies are used in a number of application domains, ranging from *serverless* computing (Akkus et al., 2018) (especially when dealing with big-data processing services with such solutions as AWS Lambda and Fargate or Google Functions) to Network Function Virtualization (NFV) (Aditya et al., 2019; Cucinotta et al., 2019).

Traditional cloud services tackle the problem of performance control by deploying *scalable* virtualized services, able to dynamically request and obtain additional physical resources from the infrastructure as needed, due to the typically dynamic nature of the submitted workloads over time. This approach can be effective only if the performance[1] of the single instance, e.g., the single virtual machine (VM) or container, is sufficiently stable. However, multi-tenancy in public clouds is well-known to cause unforeseeable interferences among co-located workloads. A similar problem occurs also in private cloud infrastructures, when hosting services from different heterogeneous departments of an organization. In that case, over-provisioning, i.e. scheduling two (or more) containers on a single CPU core, may be used to achieve a higher degree of efficiency in the use of the underlying infrastructure. For example, if two containers are known to run, in average, less than half of the CPU time each, they could be placed on the same CPU core to reduce costs, at the cost of predictability.

Unfortunately, scaling horizontally a service after detecting some performance degradation is not sufficient to grant a predictable performance to interactive cloud applications with tight timing requirements, due to the time needed to employ the monitor/decide/scale control loop. For example, a new VM might require minutes in order to boot and be ready to join an elastic group.

The classical way to provide a predictable performance/quality of service to a container or VM is by recurring to hardware partitioning. This means that virtual cores (that is, the cores of a VM or container) are pinned onto physical cores, and subsets of the available physical cores are dedicated to specific VMs or containers. This kind of approach forbids any possibility of over-provisioning, decreasing the effectiveness of a cloud infrastructure to manage its resources efficiently. Some researchers investigated on the use of real-time scheduling in the hypervisor or OS kernel for virtualized services (Abeni et al., 2018; Cucinotta et al., 2009; Xi et al., 2011).

This last strand of research is the context in which our investigation is located, as explained in what follows.

1.2 Contributions

This paper deals with the problem of performance modeling in shared containerized infrastructures, presenting an extensive and detailed validation of a performance model from the literature (Cucinotta et al., 2019). Such a model is based on a real-time scheduling strategy providing temporal isolation at the processing level, jointly with traffic shaping for isolation of multiple networking flows. The model validated in this paper currently consider only single-core containers, but extensions to multi-core/multi-processor containers are currently being considered.

Since the scheduler has already been shown to provide temporal isolation between containers (Abeni et al., 2018), the performance evaluation can be performed in isolation, running one single container per node (the container-based real-time scheduler guarantees that the performance of a container is not affected by the interference of other containers running on the same node). Several experiments have been performed in order to extensively validate all the aspects of the considered probabilistic analysis, including the networking model (that did not receive too much attention in previous works).

Moreover, this work presents a realistic deployment by using OpenStack containers to orchestrate the distributed client-server application used for the experiments.

[1] Expressed, for example, in terms of amount of work performed in a time interval.

2 RELATED WORK

There is an increasing interest in providing a stable and predictable Quality of Service (QoS) to applications running in cloud computing environments, representing an interesting issue for researchers, both from a theoretical and a practical point of view. Some prior works focused on addressing these issues through a better placement of network functions on virtualized nodes (Rankothge et al., 2017; Sang et al., 2017) and their dynamic migration (Gilesh et al., 2018), or using elastic auto-scaling strategies that cannot provide performance guarantees (Ali-Eldin et al., 2012; Roy et al., 2011; Fernandez et al., 2014).

Other works advocated the use of real-time scheduling techniques to make the execution of virtualized and distributed computing environments more predictable, and easier to analyze (Xi et al., 2015; Xi et al., 2011; Lee et al., 2012; Drescher et al., 2016; Cucinotta et al., 2010). However, the theoretical analysis proposed in such works has often been extremely simplistic, for example only considering worst-case behaviors and ignoring communications among different virtualized activities (Li et al., 2016).

The mentioned real-time scheduling techniques are based on *CPU reservations*, which are similar to the well-known traffic shaping techniques (Georgiadis et al., 1996) used in computer networks. Both reservation-based scheduling and traffic shaping are based on the idea of enforcing a limit to the fraction of resources a service/connection can use, and traffic shapers (based on token bucket or similar) are often used in cloud computing to limit the network bandwidth a VM can use.

A promising first step towards more realistic application models can be based on queuing theory (Allen, 1978; Gross and Harris, 1985) that enables probabilistic analysis (instead of simply considering the worst case). Works going in this direction exist, for example performing the probabilistic analysis of client-server applications (Cucinotta et al., 2017) or extending such an analysis to containerized execution environments (when real-time scheduling techniques are applied (Abeni et al., 2018)) to achieve predictable QoS in private clouds (Cucinotta et al., 2019).

However, the correctness and accurateness of the analysis has been verified through experiments limited to simplified setups, omitting (for example) network delays (which the theoretical model and the analysis can account for).

Finally, it is worthwhile to mention that some authors (Mian et al., 2013) tried to build performance models for applications running in public clouds, e.g., through the use of linear classifiers. However, using effective techniques for temporal isolation among co-located services, one could increase accuracy of this kind of models.

3 SYSTEM MODEL

This section presents a summary of the model from literature (Cucinotta et al., 2019) that is going to be validated in the presented experiments.

The model assumes some kind of isolation between applications, that can be achieved (for example) by using a container-based real-time scheduler (Abeni et al., 2018). This scheduler, based on some modifications to the `SCHED_DEADLINE` policy, provides the *real-time containers* abstraction.

Using real-time containers, each containerized application is reserved a *runtime* Q (meaning that it is allowed to execute for a time Q and is guaranteed to receive such an amount of execution time) every *period* P, under the condition that $\sum_i Q_i/P_i \leq 1$ among all real-time containers hosted on each CPU core.

These real-time containers are used to execute a set of server applications $S_1 \ldots S_n$ receiving (and serving) requests from clients through network connections.

A server S receives a pattern of requests modeled as a Poisson stochastic process. Namely, requests of size z^s are sent by the client with exponential and i.i.d. inter-request times with average rate λ and exponential and i.i.d. packet processing times with average rate μ (the processing time is measured when a whole CPU core is dedicated to the processing of requests).

The end-to-end Round-Trip Time (RTT) for requests is thus a stochastic variable $R^e = t^S + t^P + t^R$, where t^S is the time needed to send a request from the client to the server, t^P is the time needed by the server to process the request and t^R is the time needed by the response to go back from the server to the client.

When a server S is deployed as a real-time container with runtime Q and period P, its processing time t^P can easily be approximated if P is sufficiently small:

$$t^P \cong \frac{R}{Q/P}. \tag{1}$$

where R is the processing time when the server runs on a dedicated physical CPU core in isolation (without reservation or equivalently with $Q = P$).

The transmission and response times can be detailed as:

$$t^S = q^S + \delta + \frac{z^S}{\sigma}$$
$$t^R = q^R + \delta + \frac{z^R}{\sigma} \quad (2)$$

where:

- q^S (q^R) is the queuing time during which a request is waiting to be transmitted (sent back), δ is the client-server transmission latency (measurable as, e.g., half the ping time between the client and the server)
- z^S/σ (z^R/σ) is the time needed to transmit a request (reply) of size z^S (z^R) on a medium with speed σ.

If the network is not congested, δ has very little variability, the queuing times when sending requests and replies are negligible ($q^S \cong q^R \cong 0$), z^S is constant and $z^R \cong 0$ (because the server sends back to the client just a success/error code), then Eq. (2) simplifies in $t^S \cong \delta + \frac{Z}{\sigma}, t^R \cong \delta$ (where $Z = Z^s$ is the networking time). Hence, the transmission and response times can be substantially ignored, adding back the constant $2\delta + \frac{Z}{\sigma}$ to the processing time t^P in the final expression of the RTT. Considering for example a negligible bandwidth requirement $Z \simeq 0$, Poissonian arrivals with average rate λ and exponentially distributed service times with average rate μ, the average end-to-end response-time $E[R^e]$ and its ϕth percentile $P_\phi[R^e]$ can be approximated as:

$$E[R^e] = 2\delta + \frac{1}{\mu\frac{Q}{P} - \lambda}$$
$$P_\phi[R^e] = 2\delta - \frac{\ln(1-\phi)}{\mu\frac{Q}{P} - \lambda}. \quad (3)$$

If the request sizes z^s are distributed exponentially with average $E[s]$ (so the transmission times t^S are also exponentially distributed with rate $\nu = \frac{E[s]}{\sigma} > \lambda$) and $t^R \cong \delta$, then the average round-trip time $E[R^e]$ and its ϕth percentile $P_\phi[R^e]$ can be approximated as:

$$E[R^e] = 2\delta + \frac{1}{\nu - \lambda} + \frac{1}{\mu\frac{Q}{P} - \lambda}$$
$$P_\phi[R^e] \leq 2\delta - \frac{\ln\left(1 - \sqrt{\phi}\right)}{\alpha} \quad (4)$$

where $\alpha \triangleq (\frac{1}{\nu-\lambda} + \frac{1}{\mu\frac{Q}{P}-\lambda})^{-1}$. Note that the formula for $P_\phi[R^e]$ is a conservative bound, where $\nu \to \infty$ leads to an expression similar to Eq. (3), just with $\sqrt{\phi}$ rather than ϕ, providing an insight into the approximation implications. Refer to (Cucinotta et al., 2019) for additional details.

4 IMPLEMENTATION AND EXPERIMENTAL SETUP

To validate the performance model presented in (Cucinotta et al., 2019) (as recalled in Section 3), an implementation of the container-based real-time scheduler used in the original paper has been used. While in previous works the containers were created and configured manually, in this paper the practical usability of the technique is demonstrated by deploying the containers through OpenStack[2].

The experiments have been carried out using `distwalk`[3], a simple yet realistic open-source distributed application able to impose a configurable client-server networking traffic and processing workload on the server. In particular, the application client sends requests to the server with a random interarrival rate (λ) and random request size (z^S). The server then simulates the execution of the request for a random amount of CPU time ($1/\mu$) before sending a reply packet to the client of random size (z^R). The distribution of the random variables λ, μ, z^S and z^R can be configured from the client application interface (e.g. constant, uniform, exponential).

4.1 Implementation

The container-based real-time scheduling patches from (Abeni et al., 2018) enable the user to set runtime and period for every *cgroup*. To use this new feature through OpenStack, some modifications to its "compute" component have been necessary. Nova is the OpenStack "compute" component, that allows to create virtual servers (based on different kinds of VMs or containers). If the virtual server is implemented using containers, Nova uses libvirt to interact with the user-space tools used to manage the containers, such as LXC[4].

The modifications to Nova added the following functionalities:

- Real-time parameters for the containers can be set directly from the command line user interface;
- User-defined parameters are set in the container's *cgroup* at container creation time;
- A simple allocation algorithm (worst-fit) is used in order to choose the CPU in which to schedule

[2]More information can be found at: https://www.openstack.org/.

[3]Available on GitHub: https://github.com/tomcucinotta/distwalk.

[4]More information can be found at: https://linuxcontainers.org/lxc/

the container (only single CPU containers are considered in this proof-of-concept setup);

- Parameters are periodically retrieved from the container *cgroup* to monitor the utilization.

To implement such functionalities, the following modifications to the Nova component have been required:

- New database columns have been added, to store the container-specific real-time parameters and the compute server metrics;
- A new flavor property and a new scheduler hint have been added, to allow setting the real-time parameters at both creation time and flavor definition;
- The Nova scheduler has been modified to choose only RT-enabled compute servers when deploying a real-time container, making sure that the resulting set of real-time containers in the server is schedulable;
- The container creation method of the *Libvirt Driver* has been modified to set the correct *cgroup* parameters as well. This has been done by directly setting the *cgroup* parameters through the filesystem (an alternative solution is to modify *Libvirt* too, adding the real-time parameters in the virtual server description);
- The container creation method of the *Libvirt Driver* has also been modified to choose the CPU in which to pin the real-time container using a worst-fit strategy;
- The possibility to read the real-time parameters from the *cgroup* has been added for monitoring the server utilization. This has been done by adding a new monitoring method in the *Libvirt Driver*.

4.2 Experimental Setup

As shown in fig. 1, the experimental setup consisted of three nodes:

- A compute node, in which the application's server ran;
- A client node, in which the client ran
- A controller node, which is required by OpenStack to deploy and manage the virtual machines but has no influence on the experiments.

The application's server ran inside a real-time container on a computer with Linux kernel v5.1.0. The server container was restricted to use only one CPU core and its reservation period was always set to

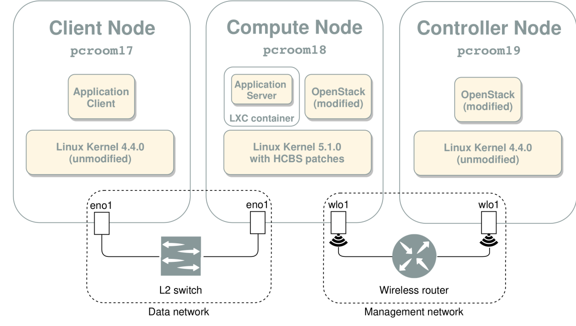


Figure 1: Schematization of the Experimental Setup. In Reality, All Nodes Have Both *wlo1* and *eno1* Interfaces but Some Are Not Relevant to the Experiments and Thus Have Been Omitted for Simplicity Sake.

$P = 2ms$, a value small enough to enable the approximation of Eq. (1). The client ran on another machine with Linux kernel v4.4.0. Both machines were identical, with an Intel(R) Core(TM) i5-4590S CPU @ 3.00GHz and 8GB of RAM. They both had a $1Gbps$ NIC connected to an L2 switch. The client-server latency has been measured using `ping` and gathering 10K samples, obtaining $\delta = 363/2 = 181.5\mu s$. Furthermore, in both machines high resolution timers and HRTICK[5] were enabled and HZ[6] was set to 1000 ($1ms$), in order to obtain more accurate results.

In what follows, every measurement has been obtained from 10K samples over each run, with the affected machines running with CPU frequency switching and hyper-threading disabled.

Network bandwidths lower than $1Gbps$ were obtained using a token bucket traffic shaper set-up using the `tc` tool with the smallest possible buffer size[7] and latency set to $100ms$[8].

In the following figures, LD represents the CPU load ($\frac{\lambda}{\mu}$), z^S is the packet size, the average inter-arrival period corresponds to $1/\lambda$ and the response time is R^e, as in Section 3. Furthermore, we introduce the *computational load ratio* (LDR) and the *network bandwidth ratio* (BWR) which represent the saturation level of the resources and are defined as the amount of used resources divided by the available amounts (note that $0 \leq LDR, BWR \leq 1$):

[5] Use high-resolution timers to deliver accurate preemption points.

[6] Compile-time constant of the Linux kernel that defines the internal timer rate, used, for example, by the scheduler.

[7] Due to timer resolution of 1/HZ, we have $buffer = min(mtu, \frac{rate}{HZ})$.

[8] Latency is the maximum amount of time a packet can sit in the TBF before being dropped. We chose this value since it is sufficiently high to avoid excessive packet drops in our use case.

$$LDR = \frac{LD}{Q/P}$$
$$BWR = \frac{\lambda E[z^S]}{\sigma}. \quad (5)$$

5 EXPERIMENTAL VALIDATION

In the following, the accuracy of the performance model, in the studied case of one server and one client, is evaluated in various configurations, by first considering negligible networking time, then one-way communications from the client to the server with constant and exponentially distributed packet sizes under constrained network bandwidth. Finally, effects of high network bandwidth saturation are highlighted.

5.1 Negligible Networking Time

The first set of experiments deals with negligible sending and receiving times (Eq. (3)). Various values of the computational workload and reservation have been tested with an average request inter-arrival time ranging from $100\mu s$ to $5ms$. The small packet size of $z^S = 2048$ bytes and the high network bandwidth of $\sigma \approxeq 1Gbps$ made networking time negligible with regards to processing time[9]. Results are shown in Figure 2, where the continuous lines represent the theoretical average and 99th percentiles of the response-time distributions, while the markers represent the respective experimental values obtained from the real platform. Note that increasing the average inter-arrival times (X-axis) implies a corresponding increase of the average processing times on the server, thus of the response times, since the computational workload (LD) is kept constant in each plot (as detailed on top of each plot).

As visible, the experimental average and 99th percentile of the response times match quite closely with theoretical expectations. Furthermore, going from the top plot with $LDR = 0.5/0.8 = 0.625$ to the middle one with $LDR = 0.7/0.9 \approxeq 0.778$ to the bottom one with $LDR = 0.8/0.9 \approxeq 0.889$, we can observe that increasing LDR causes a decrease in stability and predictability of the experimental response times. This is due to the fact that we get closer to the instability region ($LDR > 1$). This effect has been already noted and discussed in (Cucinotta et al., 2019).

In the following, we discuss results obtained in scenarios with non-negligible networking times.

[9]Networking time: $z^S/\sigma \approxeq 16\mu s$

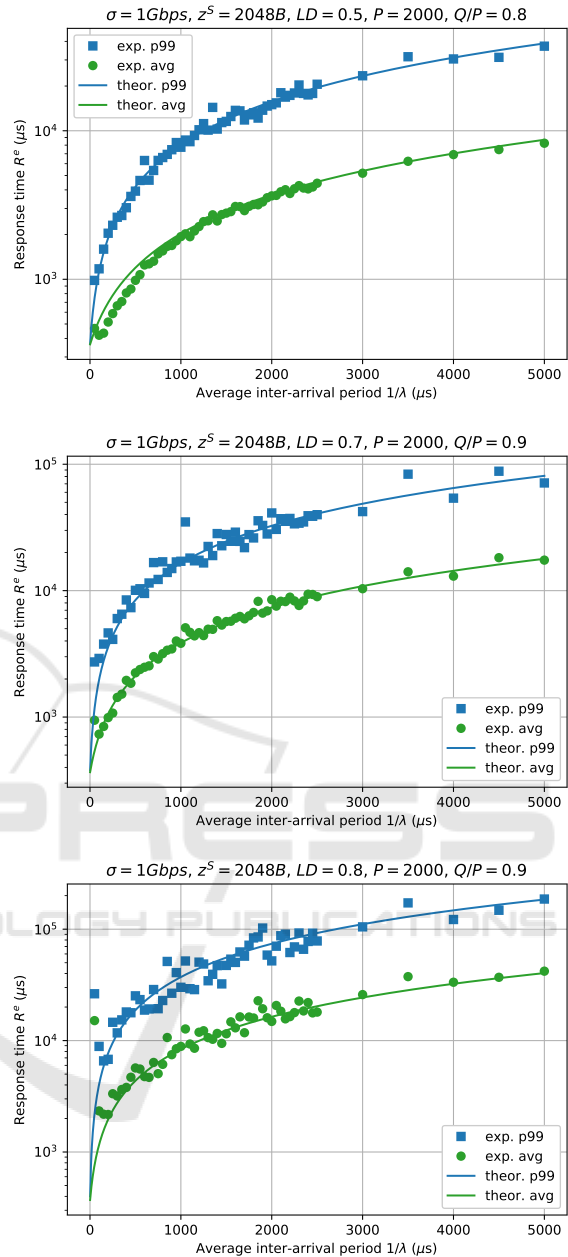


Figure 2: Experimental Response Times (Markers) for Various Configurations of Load and CPU Reservation, Compared with Theoretical Expectations of Eq. (3) (Lines). Both Average (Green) and 99th Percentile Are Shown.

5.2 Non-negligible but Constant Packet Sizes

The following set of experiments tests the model under constrained network bandwidth. In these experiments, request packet sizes are kept constant, thus they are not exponentially distributed as considered

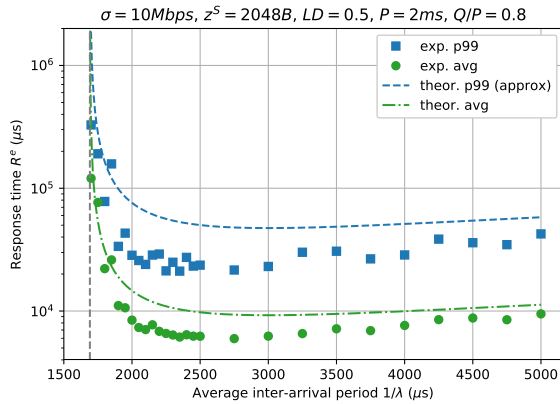


Figure 3: Experimental Response Times (Markers) for Different Inter-Arrival Times Compared with the Theoretical Expectations (Lines) of Eq.(4), for Both Average (Green) and 99th Percentile (Blue). Note That the Sent Packet Size Is Constant and That Theoretical 99th Percentile Is a Conservative Approximation.

in the model of Eq.(4).

In Figure 3, the computational workload *LD* and CPU reservation Q/P are constant, while request inter-arrival times range from $100\mu s$ to $5ms$. Note that, differently from the previous set of experiments, from a certain point decreasing the average inter-arrival times (on the X axis) causes an increase of the response times. This is due to the fact that packets get queued causing a significant increase in network delays and therefore a corresponding increase in the response times that is much higher than the benefit from the reduction in the processing times.

It can be noted that the model overestimates the response times. In the case of the 99th percentile, this was expected since Eq. 4 is a conservative approximation of it. However, the overestimation is evident also for the *average* response times. This can be explained by the fact that packet sizes are not exponentially distributed, as considered in the model, where higher response times are due to the presence of bigger requests.

Figure 4 compares the results obtained with different network bandwidths ($\sigma = 16, 32, 64\,Mbps$). As in the previous figure, experimental values fall below the expected ones for the very same reasons. In addition, it can also be noted that the 99th percentile approximation is more conservative near the "bending point", than it is farther from it, where it is more accurate.

In addition, the leftmost regions of both plots show that, even near the instability region due to network saturation (marked by min $1/\lambda$ in the plot), the model still predicts the response times with a good accuracy.

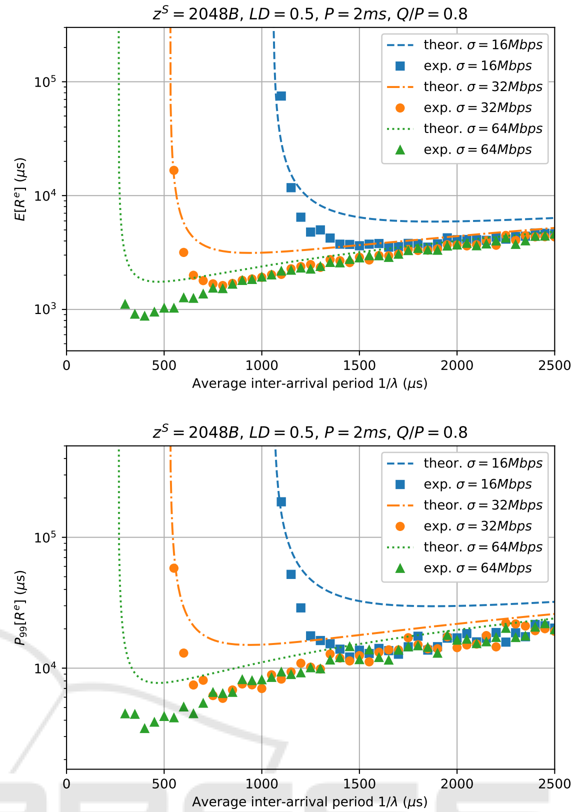


Figure 4: Average (Top Plot) and 99th Percentile (Bottom Plot) Response Times Obtained Experimentally (Markers) and Theoretically Expected from Eq.(4) (Dashed Lines), for Different Network Bandwidths and Average Inter-Arrival Periods.

5.3 Non-negligible Networking Time with Exponentially Distributed Packet Sizes

The following set of experiments tests the model under a constrained network bandwidth with exponentially distributed packet sizes.

In Figure 5, the computational workload *LD* and CPU reservation Q/P are constant, while the average request inter-arrival time $1/\lambda$ ranges from $100\mu s$ to $5ms$ (on the X axis). The same considerations as in the previous case apply here, with the difference that this time the average experimental values match closely with the expected values, since this time the model is accurate. Instead, the 99th percentile values still fall below the theoretical lines since the $P_\phi[R^e]$ approximation of Eq. 4 is conservative, as discussed earlier.

Figure 6 compares the obtained average and 99th percentile of the response times with different network bandwidths. As in Figure 5, it can be noted that the 99th percentile approximation is less accurate in

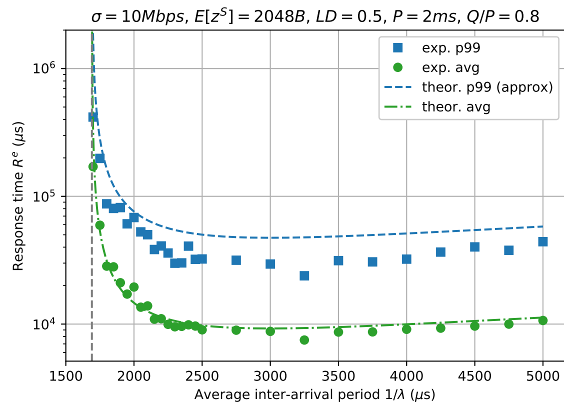


Figure 5: Experimental Response Times (Markers) for Different Inter-Arrival Times Compared with Theoretical Expectations of Eq.(4) (Lines), for Both Average (Green) and 99th Percentile (Blue). Note That the Theoretical 99th Percentile Is a Conservative Approximation.

the "bending point", even though this effect is less noticeable than before, since this time we have exponentially distributed packet sizes, reflecting better our model assumptions.

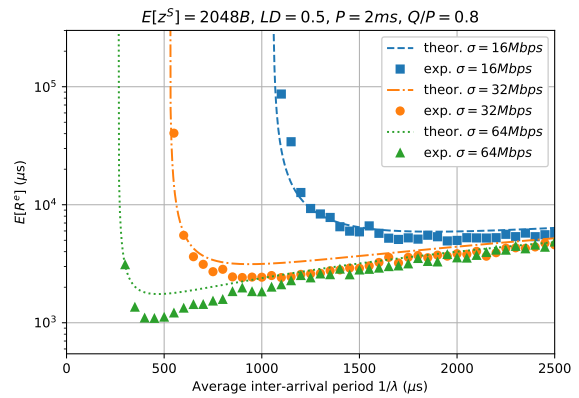
However, note that in this case, approaching the instability region $BWR > 1$, we can see a number of experimental response times statistics that exceed the predicted values, confirming that our model suffers of some limitations in this area.

5.4 Network Utilization Comparison

This set of experiments highlights the effect of very high network utilization, close to saturation conditions (bandwidth ratio BWR approaching 1).

In Figure 7, the 99th percentile of the response times at different *loads* (LD) and *load ratios* (LDR) for two different *network bandwidth ratios* (BWR), 0.5 and 0.9, are shown, along with the theoretical conservative approximation of Eq. 4.

Results highlight that, when the BWR is kept low (top plot), values are below the conservative approximation and also pretty stable. However, when approaching saturation of the available bandwidth (bottom plot, with $BWR = 0.9$), values become unpredictable and can also be observed over the conservative approximation line. Indeed, high network utilization saturating the available bandwidth can generate packet drops, which are not modeled.



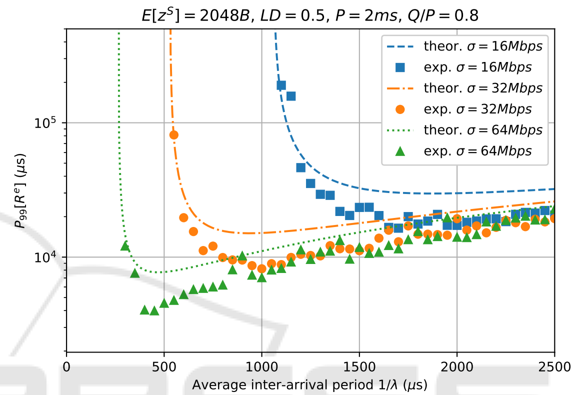


Figure 6: Average (Top Plot) and 99th Percentile (Bottom Plot) Response Times Obtained Experimentally (Markers) and Theoretically Expected from Eq.(4) (Dashed Lines), for Different Network Bandwidths and Average Inter-Arrival Periods. Note That the Theoretical 99th Percentile Is a Conservative Approximation.

6 CONCLUSIONS AND FUTURE WORK

In this paper, the problem of validating a container-based performance model using real-time scheduling has been addressed. In particular, the accuracy and effectiveness of performance analysis for real-time containers (based on real-time scheduling of the CPU, coupled with traditional traffic shaping techniques) from literature (Cucinotta et al., 2019) has been evaluated through several experiments performed on a real implementation of the technique (exploiting custom modifications to OpenStack and the Linux kernel).

The results showed that a simple client-server application with Poissonian traffic characteristics can be tightly modelled thanks to the adopted mechanisms. Our experimentation has shown that in most cases the considered performance model matches the experimental results. However, in this paper we also high-

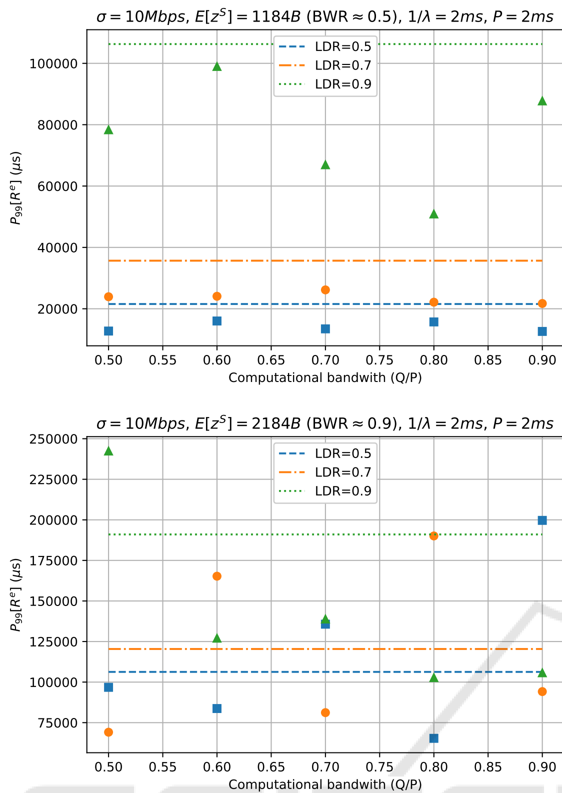


Figure 7: Experimental 99th Percentile Response Times (Markers) for Different Loads (LD) and Load Ratios (LDR) at Two Different Network Bandwidth Ratios (BWR), Compared with the Theoretical Expectations of Eq.(4) (Dashed Lines).

lighted some limitations of the model arising when getting closer to the instability region (saturation of the reserved/available computational or networking bandwidth), in addition to the well-known limitation due to non-negligible scheduling overheads as happening with too small CPU reservation periods.

This shows that real-time containers really enable a predictable performance for the hosted software components, so that we can build abstract, high-level performance models useful for designing applications with strong end-to-end QoS guarantees.

As a future work, the model validation presented in this paper can be extended in various ways. First, scenarios with more concurrent clients could be taken into consideration. This would also require some minor modifications to the performance model.

Second, the isolation capabilities of our proposed architecture has to be validated under more complex interference scenarios with a multitude of workload types. For example, mechanisms to control storage access and its associated model could be added – at least when using SSD drives.

Third, the studied model considers containers using only a single CPU core. However, in many NFV scenarios the containers run concurrent servers using multiple threads for handling the clients requests. Hence, leveraging multiple CPU cores per container would be more realistic. This is among the planned extensions of this work in our future investigations.

Finally, the performance of some real virtualized network function could be analyzed – e.g., deploying Open Air Interface[10], Kamailio[11] or similar open-source software.

REFERENCES

Abeni, L., Balsini, A., and Cucinotta, T. (2018). Container-based real-time scheduling in the linux kernel. In *Proceedings of the Embedded Operating System Workshop 2018 (EWiLi'18)*, Torino, Italy.

Aditya, P., Akkus, I. E., Beck, A., Chen, R., Hilt, V., Rimac, I., Satzke, K., and Stein, M. (2019). Will serverless computing revolutionize nfv? *Proceedings of the IEEE*, 107(4):667–678.

Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., and Hilt, V. (2018). SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA. USENIX Association.

Ali-Eldin, A., Tordsson, J., and Elmroth, E. (2012). An adaptive hybrid elasticity controller for cloud infrastructures. In *2012 IEEE Network Operations and Management Symposium*, pages 204–212.

Allen, A. O. (1978). *Probability, Statistics, and Queueing Theory with Computer Science Applications*. Academic Press, Inc., Orlando, FL, USA.

Ardagna, C. A., Asal, R., Damiani, E., and Vu, Q. H. (2015). From security to assurance in the cloud: A survey. *ACM Comput. Surv.*, 48(1):2:1–2:50.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. (2010). A view of cloud computing. *Commun. ACM*, 53(4):50–58.

Cucinotta, T., Abeni, L., Marinoni, M., Balsini, A., and Vitucci, C. (2019). Reducing temporal interference in private clouds through real-time containers. In *Proceedings of the IEEE International Conference on Edge Computing*, Milan, Italy. IEEE.

Cucinotta, T., Anastasi, G., and Abeni, L. (2009). Respecting temporal constraints in virtualised services. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 2.

Cucinotta, T., Checconi, F., Zlatev, Z., Papay, J., Boniface, M., Kousiouris, G., Kyriazis, D., Varvarigou,

[10]More information can be found at: https://www.openairinterface.org/.

[11]More information can be found at: https://www.kamailio.org/w/.

T., Berger, S., Lamp, D., Mazzetti, A., Voith, T., and Stein, M. (2010). Virtualised e-Learning with real-time guarantees on the IRMOS platform. In *2010 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–8.

Cucinotta, T., Marinoni, M., Melani, A., Parri, A., and Vitucci, C. (2017). Temporal Isolation Among LTE/5G Network Functions by Real-time Scheduling. In *Proceedings of the 7th International Conference on Cloud Computing and Services Science*, pages 368–375, Funchal, Madeira, Portugal.

Drescher, M., Legout, V., Barbalace, A., and Ravindran, B. (2016). A flattened hierarchical scheduler for real-time virtualization. In *Proceedings of the 13th International Conference on Embedded Software*, EMSOFT '16, pages 12:1–12:10, New York, NY, USA. ACM.

Fernandez, H., Pierre, G., and Kielmann, T. (2014). Autoscaling web applications in heterogeneous cloud infrastructures. In *2014 IEEE International Conference on Cloud Engineering*, pages 195–204.

Georgiadis, L., Guerin, R., Peris, V., and Sivarajan, K. N. (1996). Efficient network qos provisioning based on per node traffic shaping. *IEEE/ACM Transactions on Networking*, 4(4):482–501.

Gilesh, M., Sanjay, S., MadhuKumar, S., and Lillykutty, J. (2018). Selecting suitable virtual machine migrations for optimal provisioning of virtual data centers. *ACM Applied Computing Review*, 18:22–32.

Gross, D. and Harris, C. M. (1985). *Fundamentals of Queueing Theory (2Nd Ed.).* John Wiley & Sons, Inc., New York, NY, USA.

Lee, J., Xi, S., Chen, S., Phan, L. T. X., Gill, C., Lee, I., Lu, C., and Sokolsky, O. (2012). Realizing compositional scheduling through virtualization. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pages 13–22.

Li, Y., Phan, L. T. X., and Loo, B. T. (2016). Network functions virtualization with soft real-time guarantees. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9.

Mian, R., Martin, P., Zulkernine, F., and Vazquez-Poletti, J. L. (2013). Towards building performance models for data-intensive workloads in public clouds. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 259–270, New York, NY, USA. ACM.

Rankothge, W., Le, F., Russo, A., and Lobo, J. (2017). Optimizing resource allocation for virtualized network functions in a cloud center using genetic algorithms. *IEEE Transactions on Network and Service Management*, 14(2):343–356.

Roy, N., Dubey, A., and Gokhale, A. (2011). Efficient autoscaling in the cloud using predictive models for workload forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 500–507.

Sang, Y., Ji, B., Gupta, G. R., Du, X., and Ye, L. (2017). Provably efficient algorithms for joint placement and allocation of virtual network functions. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9.

Xi, S., Li, C., Lu, C., Gill, C. D., Xu, M., Phan, L. T. X., Lee, I., and Sokolsky, O. (2015). RT-Open Stack: CPU Resource Management for Real-Time Cloud Computing. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 179–186.

Xi, S., Wilson, J., Lu, C., and Gill, C. (2011). RT-Xen: Towards real-time hypervisor scheduling in Xen. In *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*.

SCIENCE AND TECHNOLOGY PUBLICATIONS