On the use of Linux Real-Time Features for RAN Packet Processing in Cloud Environments

Luca Abeni[1], Tommaso Cucinotta[1], Balázs Pinczel[2], Péter Mátray[2], Murali Krishna Srinivasan[2], and Tobias Lindquist[2]

[1] Scuola Superiore Sant'Anna, Pisa, Italy
{luca.abeni,tommaso.cucinotta}@santannapisa.it
[2] Ericsson {balazs.pinczel,peter.matray}@ericsson.com
{murali.krishna.xk.srinivasan,tobias.lindquist}@ericsson.com

Abstract. This paper shows how to use a Linux-based operating system as a real-time processing platform for low-latency and predictable packet processing in cloudified radio-access network (cRAN) scenarios. This use-case exhibits challenging end-to-end processing latencies, in the order of milliseconds for the most time-critical layers of the stack. A significant portion of the variability and instability in the observed end-to-end performance in this domain is due to the power saving capabilities of modern CPUs, often in contrast with the low-latency and high-performance requirements of this type of applications. We discuss how to properly configure the system for this scenario, and evaluate the proposed configuration on a synthetic application designed to mimic the behavior and computational requirements of typical software components implementing baseband processing in production environments.

1 Introduction

Networking infrastructures are experiencing a huge paradigm shift, with an ever-increasing need to support, among others, mobile scenarios with higher and higher performance requirements, both in terms of networking bandwidth and of predictable or ultra-low latency. This requires a great degree of flexibility and adaptation in the management of physical resources, where a number of lessons learnt from the domain of cloud computing are being applied in the context of networking infrastructures. For example, this is witnessed by the recent rise of network function virtualization (NFV) [3, 4] (often coupled with software-defined networking (SDN) [9]).

A NFV infrastructure hosts a number of Virtualized Network Functions (VNFs) that need to process packets with low latency. In 5G mobile scenarios, this latency has to be controlled even in the milliseconds-scale, to support properly ultra-reliable low-latency communications (URLLC) [2, 7], one of the key characteristics of 5G architectures enabling mobile communications in modern and future use-cases in such areas as industrial manufacturing and factory automation, robotics and automotive. For example, individual VNF components hosted in an NFV infrastructure may sometimes have available a "budget" in

terms of processing latency [6] that can become as little as $1ms$, which is the case of baseband packet processing, the use-case we focus on in the present paper.

One of the advantages of applying cloud principles to NFV infrastructures, is the ability to host a diverse set of workloads with heterogeneous requirements within a shared physical infrastructure. This is a geo-distributed and multi-site data center equipped with flexible storage solutions and general-purpose servers, where different VNF components are often co-located on the same physical servers in the form of virtual machines or containers.

However, due to the strict timing requirements of the scenarios mentioned above, it is of paramount importance to be able to guarantee that the end-to-end performance of hosted applications is not impaired by: a) the virtualization layer, often used to achieve the needed flexibility in management of the physical resources throughout the NFV infrastructure; b) the temporal interferences due to the co-location of multiple VNF components onto the same servers.

The second problem is well-known and often referred to as the "noisy neighbour" problem. It is generally tackled in both general cloud and NFV infrastructures by recurring to a number of custom configurations of the virtualized or containerized environment for hosting guests, typically by [10]: disabling over-provisioning in the virtual to physical CPU allocation, and applying a static mapping among them (core pinning); similarly, disabling memory over-commitment and dynamic allocation (ballooning); deploying data-intensive components with greater risks of interference in different NUMA nodes, so to minimize the interferences among their data paths in the hierarchical memory subsystem, i.e., preventing contention at the L2-cache access and memory-controller levels; disabling hyperthreading. However, some of these configurations (such as, for example, disabling hyperthreading) are not recommended in cloud environments because they end up decreasing the CPU throughput and the possibility to run multiple applications on the cloud nodes.

This paper provides an experimental evaluation of the impact of various hardware and OS tuning features mentioned above.We focus on an industrial use-case scenario tied to low-latency packet processing for 5G/URLLC, namely baseband packet processing. The results show that with careful tuning of the hardware and software configuration it is possible to remove most of the sources of interference and tail-latency (making it possible to host such an application with the required level of time-predictability) without compromising the CPU throughput and the performance of other applications running in the cloud. While previous works [5] disabled features like hyperthreading to achieve more predictable response times, this work shows how hyperthreading can be left enabled without compromising the real-time performance of the baseband software.

2 Scheduling the BaseBand Application

In RAN packet processing, multiple frequencies of the available spectrum are used to handle communications, where the time is divided into Transmission

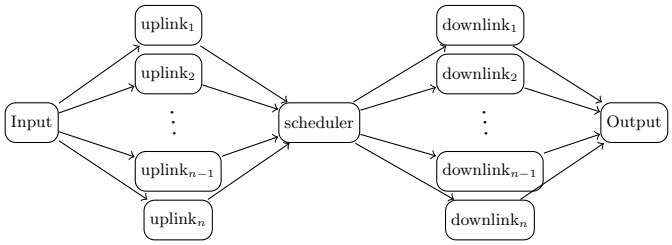

Fig. 1. Model of the baseband application as a DAG.

Time Intervals (TTIs) all having the same duration. In each TTI, data is received by the various radio cells, and new data is prepared to be transmitted to them.

The baseband software considered as a test-case for this work is designed to serve n cells and is a multi-threaded application, described by the directed acyclic graph (DAG) shown in Figure 1. This is composed of: an I/O thread, responsible for communications with the remote radio units ("Input" and "Output" nodes in the figure); n uplink threads processing data received from one of the cells; a scheduler thread, coordinating the use of the spectrum and time slots within each TTI; n downlink threads preparing data to be sent to one of the cells. In practice, there is an uplink thread and a downlink thread per cell.

For the sake of simplicity, assume that the I/O thread periodically activates all the uplink threads at the beginning of each TTI. After executing for some time, each uplink thread sends an activation to the scheduler thread and then blocks until the beginning of the next TTI (next activation from the I/O thread). After receiving an activation from all the n uplink threads, the scheduler thread wakes up, executes for some time, and then activates all the downlink threads. If the end of the TTI arrives before all the uplink threads sent an activation to the scheduler thread, a timeout fires and the scheduler thread is activated anyway. Each downlink thread, after being activated by the scheduler thread, executes for some time and then terminates; all the n downlink threads should terminate before the end of the period.

Hence, we measure an end-to-end response time from the activation of the first uplink thread to the termination of the last downlink thread. This end-to-end response time should be smaller than an end-to-end relative deadline D which is equal to the TTI duration. If such a deadline is seldom missed, then higher-layer protocols can recover by re-transmits, but if this occurs too frequently, then the transmissions degrade in quality or even fail. Looking at Figure 1, we can see that if C_{ul}, C_{sched}, and C_{dl} are the WCETs (worst case execution times) of the uplink, scheduler and downlink threads, then the maximum end-to-end response time, equal to the longest path from I/O to the downlink termination (maximum makespan of the DAG), is $C_{ul} + C_{sched} + C_{dl}$. So, the parallel task will respect the deadline D if $C_{ul} + C_{sched} + C_{dl} \leq D$. This response time can be obtained by scheduling all the threads as soon as they are

activated, and using n CPU cores (a core sequentially executes an uplink thread, the scheduler thread and a downlink thread, the other $n-1$ cores execute an uplink thread and a downlink thread). Hence, the end-to-end response time can be minimized by having only one active real-time thread per core, avoiding temporal scheduling; in this case, a `SCHED_FIFO` scheduling policy is considered to be the best option to run this application: if each thread is assigned the maximum real-time priority, then it is scheduled as soon as it activates, as required. More advanced scheduling policies such as `SCHED_DEADLINE` [8, 1] could be useful when the CPU scheduler has to schedule multiple real-time threads on the same CPU core, or when it is necessary to limit the fraction of CPU time consumed by a real-time application. Pinning real-time threads to specific CPU cores can be useful to avoid migrations and reduce the scheduling overheads, or to cope with the unpredictabilities caused by hyperthreading, as shown in Section 3.

When there are no uplink/scheduler timeouts, the scheduler executes only after all the uplink threads are finished, and the downlink threads execute only after the scheduler thread is finished; so, if the total end-to-end time is smaller than 1 TTI, then this property is respected using n cores only. If an uplink thread takes more than 1 TTI (scheduler timeout) or the total end-to-end response time is larger than 1 TTI (uplink threads are activated while downlink threads are still active), then $2n$ cores could potentially be needed.

For certain scenarios, a typical pattern of execution times could look like the following: the execution times of the uplink threads are generally shorter than $500\mu s$ (except for a few rare outliers), the execution times of the scheduler thread are generally smaller than $100\mu s$, and the execution times of the downlink threads are generally smaller than $300\mu s$. However, there are a few sources of non-determinism causing fluctuations in these numbers, i.e., radio link quality, channel coding, cell load, and others. Assuming $C_{ul} = 500\mu s$, $C_{sched} = 100\mu s$, and $C_{dl} = 300\mu s$, we have $C_{ul} + C_{sched} + C_{dl} = 900\mu s$, so a TTI of $1ms$ can be supported using n CPU cores.

If, instead, execution times distributions with longer tails are assumed, then $2n$ CPU cores might be needed.

3 CPU Configuration

The goal of this work is to run a virtual baseband application on a large server based on multiple Intel x86 CPUs with a large number of cores. These modern CPUs are designed to maximize the average performance/throughput and reduce power consumption by using various mechanisms. The three most important ones are Dynamic Voltage and Frequency Scaling (DVFS), CPU idle states, and hyperthreading. In particular, DVFS and idle states allow reducing power consumption when the server is not fully loaded, while hyperthreading allows doubling the number of *logical* CPU cores seen by applications.

The hyperthreading technology allows the OS kernel to see a single CPU core (referred to as *hardware core* in the following) as two *siblings* (also referred to as *logical cores* or *hardware threads*). This means that if a CPU is composed of

n hardware cores the OS can use $2n$ siblings to schedule the application tasks. Technically, this result is achieved by duplicating the hardware resources that store the state of each core (such as the CPU registers). Other hardware resources such as the ALU, the caches, and similar, instead, are not duplicated and are shared by the siblings executing on the same physical core. The two siblings executing on the same physical core risk competing for the execution resources that are not duplicated; hence, running an application on a sibling can slow down the execution of applications on the other one. This makes the execution unpredictable, and this is why hyperthreading is often disabled when real-time performance and determinism are important.

When a CPU is idle (it has no tasks to execute), some hardware components can be turned off to save some energy. Modern CPUs allow to achieve this result by entering different *idle states*; for example, Intel CPUs can be in different "C-states" (named, `C0`, `C1`, etc...). Increasing the state number, a C-state is said to be "deeper", stops more hardware components, and allows saving more energy. Returning from an idle state to `C0` takes some time, which increases with the state number (deeper C-states have longer exit and entry latencies). This is why modern operating system kernels such as Linux allow disabling some (or all) of the idle states. When all the idle states are disabled and a CPU is idle, it executes a busy loop in the idle task.

In Intel CPUs, C-states are per physical core (so, a hardware thread cannot enter an idle state if the other sibling of the physical core is executing machine instructions), however, Linux allows disabling C-states for individual logical cores. When a sibling is idle and can enter an idle state, it is stopped (so, a single sibling remains active on the physical core) but the C-state of the physical core does not change until its other sibling also needs to enter an idle state.

Finally, the DVFS mechanism allows lowering the working frequency of CPU cores (and consequently the voltage at which the CPU is driven) to save some energy. Obviously, the frequency of a core should be reduced when the core is not fully loaded. Generally, the OS is responsible for selecting the most appropriate working frequency for the various CPU cores, based on an estimation of the system workload or on some constraints imposed by the applications running in the system. The various frequency/voltage configurations supported by a CPU are often known as Operating Performance Points (OPP) or Power States (P-states); even if in modern Intel CPUs the P-state concept is more advanced than a simple frequency/voltage configuration, the Linux kernel internally maps P-states to CPU frequencies, making it possible to give the CPU hints regarding the frequencies at which its cores should work. Obviously, when the DVFS mechanism is active the CPU speed becomes less predictable: even if the frequency scaling algorithm is configured to always select the maximum possible speed when a real-time task is active, the time needed to switch frequency can negatively affect the real-time performance. This is why DVFS is generally disabled (and the CPU is driven at an almost constant frequency — even disabling mechanisms such as the "*turbo mode*") when real-time performances need to be guaranteed.

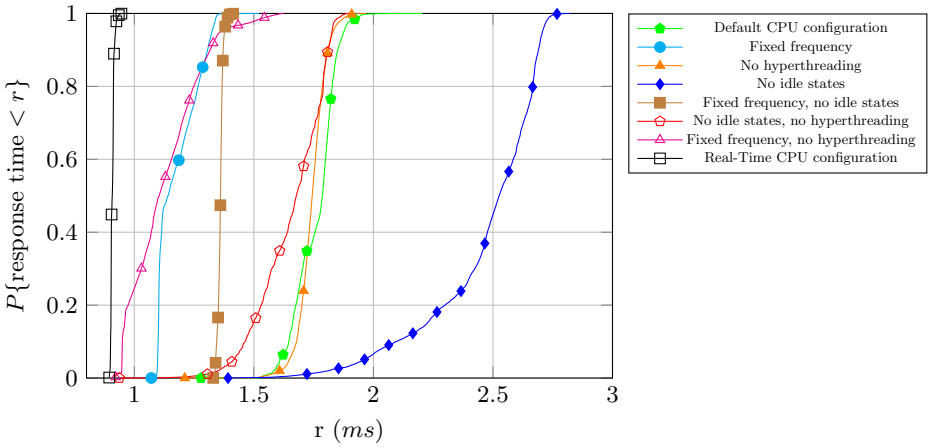

Fig. 2. CDF of the end-to-end response time with deterministic execution times and various CPU configurations.

As mentioned, the general guidelines for executing real-time applications on modern CPUs recommend to disable frequency scaling, idle states and hyperthreading. Some preliminary results (see below) confirm that with this configuration (named "Real-Time CPU" configuration in the following) the CPU can support deterministic execution of the baseband real-time application. However, in a cloud environment it would be useful to keep hyperthreading enabled, to give more CPU time to non-real-time applications running in background.

To experiment with various possible hardware and software configurations using a deterministic and reproducible real-time workload, we implemented a synthetic application that, from a task scheduling perspective, behaves similarly to a softwarized baseband software (same number of threads and synchronization among them), but allows users to control the execution times of the various threads. Basically, the synthetic application reproduces the thread synchronization used by the real baseband software, but the various threads consume CPU time by executing busy loops instead of decoding/encoding the radio signal (the loop counters are calibrated so that each thread executes for the desired amount of time). All the tests have been performed on a server equipped with a dual Intel(R) Xeon(R) CPU E5-2640 v4 running at $2.40GHz$ (having 10 physical cores per CPU; with hyperthreading enabled there are 40 siblings).

First of all, we tested various CPU configurations with the synthetic application configured for 2 cells (2 uplink threads and 2 downlink threads) and deterministic execution times $C_{ul} = 500\mu s$, $C_{sched} = 100\mu s$, and $C_{dl} = 300\mu s$. Figure 2 reports the Cumulative Distribution Function (CDF) of the end-to-end response times measured with different CPU configurations ranging from the "Default CPU" configuration (DVFS, turbo mode, idle states and hyperthread-

ing are enabled) to the "Real-Time CPU" configuration (DVFS, turbo mode, idle states and hyperthreading are all disabled).

Looking at the figure, there are some interesting results to be noticed. First of all, it can be seen that the "Real-Time CPU" configuration works as expected, and the response times are always very close to the theoretical value of $900\mu s$ (the CDF plots the probability to measure an end-to-end response time smaller than the value r on the x-axis, hence an almost vertical line indicates almost deterministic response times). Another interesting fact to be noticed is that the "Fixed Frequency no idle states" configuration also exhibits deterministic response times, but they are larger than the theoretical value (around $1350\mu s$ instead of $900\mu s$). All the other plots show a much larger execution-time variation, and the result changes from run to run, while the "Real-Time CPU" and "Fixed Frequency no idle states" configurations generate reproducible results.

Another interesting thing to be noticed in the figure is the "No idle states" curve, which looks strange since it shows that disabling the CPU idle states increase the response times and make them less deterministic. This strange behaviour can be explained by noticing that this configuration disables all the idle states (C-states deeper than `C0`) for all the CPUs seen by the Linux kernel (which, in this case, are logical cores). In this configuration, all the idle siblings will execute a busy loop in the idle task. Hence, if the real-time application is executing on the first sibling of a physical core and the second sibling of such physical core is idle, then the real-time application will experience the interference of the idle loop running on the second sibling!

This also explains the increased response times incurred when using the "Fixed frequency no idle state" configuration. To address this issue, *idle states should be disabled only on the logical cores executing real-time applications* (in this way, when the other sibling is idle, it is stopped and does not interfere with the execution of the real-time application). To do this, the real-time application has to be pinned to a limited number of siblings (so that it is possible to know in advance on which siblings the real-time application will execute and to disable idle states only on them). This configuration will be referred to as "Fixed frequency no idle states on RT cores" in the following.

Of course, the "Fixed frequency no idle states on RT cores" configuration can improve the real-time performance when some cores are idle, but does not offer significant advantages when all the logical cores are heavily loaded. To get good real-time performance in this situation (without resorting to disabling hyperthreading completely) it would be necessary to make sure that while a real-time thread is executing on a sibling nothing is scheduled on the second sibling of its physical core. This result can be achieved by using a functionality that has been recently introduced in the Linux kernel, named *Core Scheduling*[3].

With Core Scheduling it is possible to assign "*cookies*" to threads, and the kernel CPU scheduler will make sure that only tasks with the same cookie executes simultaneously on the same physical core (so, if a thread with cookie $\mathcal{C}$ is

[3] https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/core-scheduling.html

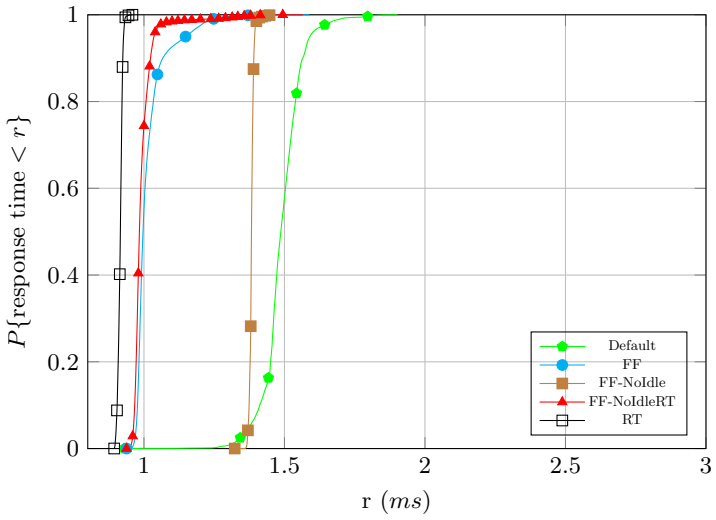

Fig. 3. CDF of the end-to-end response time with deterministic execution times.

executing on the first sibling of a physical core, then only threads with cookie $\mathcal{C}$ can execute on the second sibling — if no other thread with cookie $\mathcal{C}$ is ready for execution, the second sibling is left idle).

Although Core Scheduling has been originally developed to address security issues (mitigating hardware bugs such as L1TF[4]), it can be used to increase the predictability of real-time applications. By assigning unique cookies to the real-time threads (and leaving non-real-time threads with no cookies), it is possible to make sure that real-time threads do not share their physical cores with any application (and hence do not suffer of any interference due to hyperthreading). This configuration will be named "Core Scheduling" in the following.

4 Experimental Results

An extensive set of experiments has been performed on the server described in Section 3 to evaluate the previously discussed configurations and the effectiveness of the core scheduling mechanism.

Figure 3 reports the CDFs of the end-to-end response times for the most stable CPU configurations when the threads execution times are assumed to be deterministic (and equal to the worst-case values). For the sake of simplicity, from now on "FF" represents the "Fixed Frequency" CPU configuration, "FF-NoIdle" represents the "Fixed frequency, no idle states" configuration, and "FF-NoIdleRT" represents the "Fixed frequency, no idle states on RT cores"

[4] https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/l1tf.html

Table 1. Summary of the naming used for various CPU configurations.

Name	Description
Default	DVFS, turbo mode, idle states and hyperthreading enabled
FF	Fixed Frequency (DVFS and turbo mode disabled)
FF-NoIdle	Fixed Frequency, no idle states (DVFS, turbo mode and idle states disabled)
FF-NoIdleRT	Fixed Frequency, no idle states on RT cores (DVFS and turbo mode disabled, idle states disabled only on the cores where the real-time threads execute)
Core	Core scheduling (as above, but core scheduling is used to ensure that when a real-time thread executes on a physical core, no other thread can execute on the other sibling of the physical core)
RT	Real-Time CPU configuration (DVFS, turbo mode, idle states and hyperthreading disabled)

configuration (Table 1 summarizes these symbols). In this experiment, the baseband software is the only application running in the server, which is otherwise idle, hence most of the response times are lower respect to Figure 2. As already noticed in Section 3, the Real-Time CPU configuration is very effective in providing deterministic response times near to the theoretical value of $900\mu s$. "FF-NoIdle" also results in deterministic response times, but introduces an additional delay due to the interference of the idle loop with the siblings where the real-time threads are executing. Notice that the "RT" and "FF-NoIdle" curves are identical to the ones of Figure 2, confirming the determinism of these configurations. "FF-NoIdleRT" reduces the response times of "FF-NoIdle" by pinning the real-time threads on siblings 0 and 2, and disabling the CPU idle states only on these two siblings (as previously described).

The experiment has also been repeated using randomly-distributed execution times for the real-time threads, instead of considering their worst-case values (see Figure 4). To account for some outliers executing for more than the expected WCETs, the probability distributions of the execution times[5] have some tails larger than C_{ul}, C_{sched} and C_{dl} (hence, the average execution times are smaller than in the previous experiments while the maximum execution times are larger — although very infrequent).

When the system is loaded with some background non-real-time applications, things look more interesting and both "FF" and "FF-NoIdleRT" result in response times comparable with the "FF-NoIdle" configuration (so, not good for real-time). This is where core scheduling can help. To investigate this setup, we executed some more experiments pinning the real-time threads to siblings 0 and 2 and running a CPU-intensive application on siblings 20 and 22 (the two

[5] In this case, gaussian distributions have been used for the sake of simplicity.

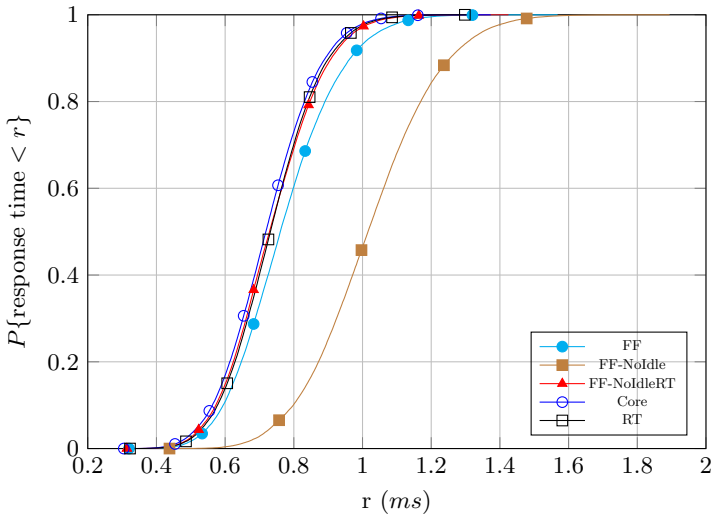

Fig. 4. CDF of the end-to-end response time with gaussian execution times and no additional load.

Table 2. CPU throughput for non-real-time applications running in background.

	ffmpeg4	ffmpeg2	Prime4	Prime2
FixFreq	1496 (100%)	897 (100%)	84794 (100%)	69091 (99.9%)
No idle states	1001 (61.9%)	853 (95%)	70390 (83%)	69102 (100%)
No idle on RT cores	1253 (83.7%)	886 (98.7%)	78168 (92.2%)	69084 (99.9%)
Core Scheduling	1195 (80%)	837 (93.3%)	76117 (89.7%)	67780 (98%)
Real-Time CPU	917 (61.3%)	789 (88%)	61882 (72.9%)	58417 (84.5%)

hardware threads sharing physical cores with siblings 0 and 2). We selected two different CPU-intensive applications to run in background: a synthetic benchmark using n threads to compute prime numbers and a more realistic application transcoding some audio and video in background (using ffmpeg). The results are reported in Figure 5 and show that the "RT" and the "Core" configuration result in very similar response times (which are basically identical to the ones shown in Figure 4). All the other configurations result in large response times due to the interference of the non-real-time application caused by hyperthreading.

To evaluate the "cost" of this isolation Table 2 reports the total number of frames transcoded by the `ffmpeg` instance or the total number of prime numbers found by the synthetic application. Two different setups have been tested: non-real-time application scheduled on siblings 0, 2, 20 and 22 (indicated as "Prime2" and "ffmpeg2") or scheduled on siblings 4, 6, 20 and 22 (indicated as "Prime4" and "ffmpeg4"). Notice how core scheduling allows to find a good

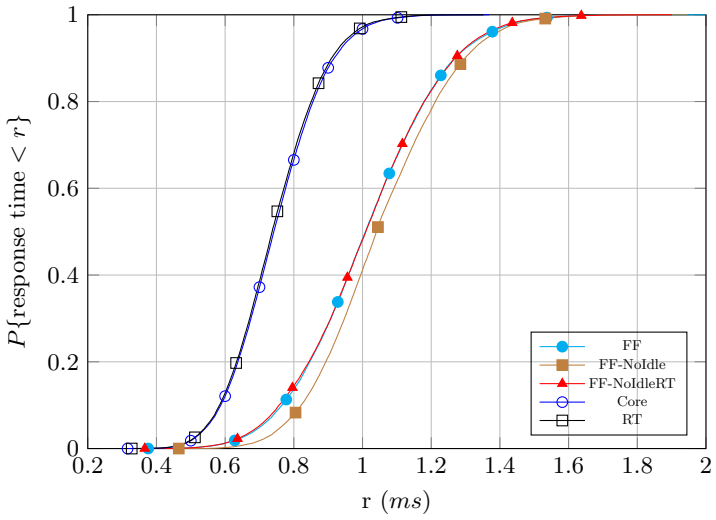

Fig. 5. CDF of the end-to-end response time with gaussian execution times and non-real-time load in background.

trade-off between real-time and non-real-time performance, increasing the CPU throughput respect to the Real-Time configuration without compromising real-time performance.

5 Conclusions

This paper evaluated the performance of a cloud node when serving a cRAN application characterized by strict temporal constraints. While the current approach to cloudify this kind of baseband applications relies on disabling hyperthreading and reducing the CPU time left to other applications running in the cloud, this paper showed how it is possible to find a reasonable trade-off between real-time performance and cloud throughput without disabling hyperthreading, by properly using advanced kernel features such as Core Scheduling.

As a future work, we will investigate scalability issues and power consumption. In this regard, some preliminary results seem to indicate that core scheduling allows to find a good trade-off between real-time performance and power consumption. We also plan to take advantage of more advanced scheduling policies, such as `SCHED_DEADLINE`, to reduce the number of CPU cores used by real-time threads and to host multiple real-time applications on the same node.

References

1. Abeni, L., Balsini, A., Cucinotta, T.: Container-based real-time scheduling in the linux kernel. SIGBED Review **16**(3), 33–38 (October 2019)

2. Ericsson: 5G wireless access: an overview – Ericsson White Paper 1/28423-FGB1010937 (Apr 2020)
3. ETSI: Network Functions Virtualisation – Introductory White Paper – An Introduction, Benefits, Enablers, Challenges & Call for Action. Tech. rep., SDN and Openflow World Congress, Darmstadt, Germany (2012), https://portal.etsi.org/nfv/nfv_white_paper.pdf
4. ETSI: Network Functions Virtualisation (NFV) – Update White Paper – Network Operator Perspectives on Industry Progress. Tech. rep., SDN and Openflow World Congress, Frankfurt, Germany (2013), http://portal.etsi.org/nfv/nfv_white_paper2.pdf
5. Foukas, X., Radunovic, B.: Concordia: Teaching the 5g vran to share compute. In: Proceedings of the 2021 ACM SIGCOMM 2021 Conference. p. 580–596. SIGCOMM '21, Association for Computing Machinery, New York, NY, USA (2021)
6. Giannone, F., Gupta, H., Kondepu, K., Manicone, D., Franklin, A., Castoldi, P., Valcarenghi, L.: Impact of ran virtualization on fronthaul latency budget: An experimental evaluation. In: IEEE Globecom Workshops. pp. 1–5 (Dec 2017)
7. Le, T.K., Salim, U., Kaltenberger, F.: An overview of physical layer design for ultra-reliable low-latency communications in 3gpp releases 15, 16, and 17. IEEE Access **9**, 433–444 (2021). https://doi.org/10.1109/ACCESS.2020.3046773
8. Lelli, J., Scordino, C., Abeni, L., Faggioli, D.: Deadline scheduling in the linux kernel. Software: Practice and Experience **46**(6), 821–839 (2016)
9. Open Networking Foundation (ONF): ONF SDN Evolution. ONF TR-535, ONF (2016), http://www.opennetworking.org/wp-content/uploads/2013/05/TR-535_ONF_SDN_Evolution.pdf
10. Suchánek, M., Navrátil, M., Bailey, L., Boyle, C.: Performance Tuning Guide – Monitoring and optimizing subsystem throughput in RHEL 7 (Aug 2021), https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/performance_tuning_guide/index