Period Estimation for Linux-based Edge Computing Virtualization with Strong Temporal Isolation

Luca Abeni, Tommaso Cucinotta, and Daniel Casini
Scuola Superiore Sant'Anna, Pisa, Italy.
{name.surname}@santannapisa.it

***Abstract*—Virtualization of edge nodes is paramount to avoid their under-exploitation, allowing applications from different tenants to share the underlying computing platform. Nevertheless, enabling different applications to share the same hardware may expose them to uncontrolled mutual timing interference, as well as timing-related security attacks. Strong timing isolation through `SCHED_DEADLINE` reservations is an interesting solution to facilitate the safe and secure sharing of the processing platform: nevertheless, `SCHED_DEADLINE` reservations require proper parameter tuning that can be hard to achieve, especially in the case of highly dynamic environments, characterized by workloads that need to be served without knowing any accurate information about their timing. This paper presents an approach for estimating the periods of `SCHED_DEADLINE` reservations based on a spectral analysis of the activation pattern of the workload running in the reservation, which can be used to assign and refine reservation parameters in edge systems.**

***Index Terms*—edge computing, Linux scheduling, resource reservation, isolation**

I. Introduction

The edge computing paradigm is of key importance to keep computations closer to where data originates, with advantages in terms of latency, energy consumption, and privacy with respect to Cloud-based approaches that provide data to be sent to the Cloud for being elaborated [1], [2]. In this context, it becomes crucial to efficiently share edge computing nodes among diverse applications coming from different tenants. A representative example comes from the context of mobility, where autonomous vehicles need to offload computationally intensive tasks to the closest edge server on their path. A common virtualization approach consists of just partitioning a subset of the computational resources and assigning them to an application in an exclusive way.

Nevertheless, such a coarse-grained partitioning can lead to underutilization of computing platforms, especially in the presence of lightweight workloads, which anyway need to receive an integer fraction of the physical cores to execute.

In many cases, edge nodes are equipped with the Linux operating system, which affirms as an excellent solution to achieve performance while remaining compatible with a plethora of software stacks (e.g., think of frameworks for artificial intelligence) that are needed by many practical use cases [3] and which are often incompatible with special purpose operating systems (e.g., VxWorks or FreeRTOS).

When using Linux, its `SCHED_DEADLINE` kernel scheduler [4] is an excellent solution for providing the fine-grained virtualization of computational resources. Indeed, `SCHED_DEADLINE` implements a *resource reservation* mechanism that allows encapsulating each application in a virtual platform (VP) composed of several virtual CPUs (vCPUs), each with a guaranteed fraction of the overall CPU bandwidth and a bounded latency (service-delay) [5], [6].

`SCHED_DEADLINE` is a mechanism to obtain (fine-grained) resource partitioning but also acts as an *enforcement* mechanism, meaning that it guarantees that each application (i.e., VP) receives no more than its allocated processing bandwidth. This way, it shields applications running on the same processing platforms from their mutual timing interference. Therefore, resource enforcement is especially useful in contexts where different applications do not trust each other, and there is the possibility that a misbehaving application (e.g., due to bugs in the code or cyber-attacks) can harm the timing requirements of other co-located applications.

Furthermore, `SCHED_DEADLINE` is very general, and its VP abstraction can fit the case of a single Linux thread, a virtual machine (VM) running on Linux as a host operating system (e.g., leveraging a type-2 hypervisor such as QEMU/KVM), and a container (in this case, a patch non-included in mainline Linux is required [7]). Nevertheless, to obtain the desired bandwidth and latency for a `SCHED_DEADLINE`-based VP, it is required to configure a budget (also called *runtime*) of Q time units that are assigned to a specific vCPU over a period of P time units for each of the vCPUs of the VP. Therefore, also *the number of vCPUs of the VP* is another configuration parameter. However, setting these parameters is totally non-trivial, especially for applications arriving dynamically at an edge node without any knowledge of its timing information, such as execution times, activation patterns, and required parallelism.

This Paper. This paper targets the problem of finding a proper parameter tuning for `SCHED_DEADLINE` reservations in edge platforms in which workloads (tasks) arrive dynamically (as a result of an offloading request from a mobile device, for example) without any associated information about their timing.

A tool to estimate the tasks' periods (based on previous work [8]) using a Linux daemon thread with a remote procedure call (RPC) interface is developed and evaluated experimentally under different scenarios. We extensively discuss how to shape reservation servers under different configurations, including complex configurations (such as virtual machines with multiple tasks running in the same vCPUs), which are not addressed in previous work. Furthermore, we discuss how to

use such a dynamic estimation method for reservation periods in an edge scenario in which mobile devices send offloading requests to edge nodes.

II. Background on Linux

Linux is arguably one of the most popular operating system kernels and runs in many different contexts, ranging from embedded systems and edge computing up to high-performance computing while encompassing several application areas, including robotics, industrial automation, and even space [9]. Since 2014 (kernel version 3.14), Linux has featured the `SCHED_DEADLINE` real-time scheduler, which implements the earliest deadline first (EDF) scheduling algorithm combined with the constant bandwidth server (CBS) reservation algorithm [5]. Linux provides multiple scheduling policies, implemented by scheduling classes which are queried in order (thus implementing an implicit priority hierarchy between them): the `stop_machine` scheduling class, which does not implement any real scheduling policy but is used for kernel facilities; the `SCHED_DEADLINE` scheduling policy, extensively discussed in the following; two variants of the fixed-priority policy (`SCHED_FIFO` and `SCHED_RR`); and the general purpose policy `SCHED_OTHER`, based on a proportional share scheduling algorithm.

As previously discussed, under `SCHED_DEADLINE` each vCPU of a VP leverages a *CBS reservation server* that is characterized by its time budget of Q time units and a period P. Each reservation server can be scheduled by either a partitioned or global scheduling approach. The former approach statically assigns each vCPU to a specific physical core, forbidding migrations; the latter allows each vCPU to be scheduled on any core, also allowing runtime migrations. Both approaches have advantages and disadvantages, which have been intensely studied in the literature [10], [11]. The Q and P parameters have a direct mathematical relation with two other parameters that are easier to relate to the application *key performance indicators*: the amount of *bandwidth* α of the vCPU and its worst-case *maximum service latency* Δ. More precisely, α expresses the fraction of processing capacity that the reservation server receives in the long run; Δ denotes the maximum delay that a task can achieve in a vCPU from when it becomes ready to execute to when it starts being serviced.

The α and Δ parameters of a vCPU implemented by a `SCHED_DEADLINE` reservation server under partitioned scheduling is expressed as [6], [12], [13]

$$\alpha = P/Q,\ \Delta = 2 \cdot (P - Q) \tag{1}$$

Different relations and approaches to obtain a bandwidth and latency requirement from a budget and period pair in other settings are available in the literature [6], [12]–[17], depending on the scheduling paradigm used to schedule `SCHED_DEADLINE` vCPUs of the VP on the physical cores of the processing platform and tasks by vCPUs (typically, global or partitioned scheduling).

III. System Model

The considered system is composed of a distributed network of edge nodes. Each edge node is a (Linux-based) computing platform with homogeneous physical cores.

On each edge node serves a set $\mathcal{V}$ of VPs, denoted as v_j, which includes m_j vCPUs $c_{j,1}, \dots c_{j,m_j}$. Each v_j is characterized by a tuple $(\overline{Q}_j, \overline{P}_j)$: $\overline{Q}_j = (Q_{j,1}, \dots Q_{j,m_j})$ and $\overline{P}_j = (P_{j,1}, \dots P_{j,m_j})$ are the vectors of budgets and periods of each individual vCPU $c_{j,x} \in v_j$.

The virtual platforms in set $\mathcal{V}$ are scheduled by the Linux operating system according to the EDF algorithm.

The workload running inside each vCPU is scheduled with one of the other Linux schedulers (e.g., the fixed-priority scheduler). Each v_j serves a workload composed of a set of tasks Γ_j. Each task $\tau_i \in \Gamma_j$ is characterized by a maximum-observed execution time C_i and an activation period T_i, meaning that the task is considered releasing a (potentially, infinite) sequence of instances (called jobs), each one spaced by T_i time units.

IV. Task Period Estimation using PeriodWiz

To properly set the vCPUs scheduling parameters, it is paramount to estimate the tasks' activation patterns accurately. In particular, it is essential to identify tasks that can be modeled through periodic activation patterns and to estimate their activation periods. This can be performed by identifying tasks' activation events and performing a frequency-domain analysis on them [8], [18].

The Linux kernel's *function tracer* (`ftrace`[1]) is used to extract the tasks' "wakeup" events, indicating that a process or thread becomes selectable by the kernel CPU scheduler, moving from a blocked state to the ready state. The sequence of wakeups for each relevant task is registered by modeling the j^{th} wakeup of task τ_i, occurring at time $r_{i,j}$, as a Dirac delta $\delta(t - r_{i,j})$ centered at time $r_{i,j}$. After collecting N of these events, a function $a_i(t) = \sum_j \delta(t - r_{i,j})$ describing the activations of task τ_i is built and is transformed to the frequency domain:

$$\begin{aligned}\mathcal{F}(a_i(t)) &= \int_{-\infty}^{\infty} a_i(t) e^{-j2\pi ft} dt = \\ &= \int_{-\infty}^{\infty} \sum_{j=1}^{N} \delta(t - r_{i,j}) e^{-j2\pi ft} dt = \sum_{j=1}^{N} e^{-j2\pi f r_{i,j}}\end{aligned} \tag{2}$$

The energy of this Fourier transform is then computed as

$$S_i(f) = \sum_{j=1}^{N} \sqrt{\cos^2(2\pi r_{i,j} f) + \sin^2(2\pi r_{i,j} f)} \tag{3}$$

Identifying peaks in this energy function can then estimate the task's periodicity [8].

The program originally used to detect periodic tasks with a top-like interface [8] has been modified turning it into **PeriodWiz** (the Period Wizard), a daemon that:

[1] See https://www.kernel.org/doc/html/latest/trace/ftrace.html.

- sets up `ftrace` for tracing the wakeup events of the monitored real-time tasks;
- stores the functions $a_i(t)$ describing the activation patterns of such tasks;
- periodically (with a configurable period T) computes the energy $S_i(f)$ for each monitored task τ_i, looking at the peaks in $S_i(f)$ and identifying the periodic tasks with their periods.

PeriodWiz exports an RPC interface that allows clients to add new tasks to the set of real-time tasks to be monitored (by registering a new process ID) and to query for the period of a monitored task.

Clearly, it is essential to compute the Fourier transform of $a_i()$ after collecting an appropriate number N of events: if a too-small number is selected, not enough samples are registered in $a_i(t)$ and the period estimation risks to be based on noisy data; on the other hand, if N is too large, we risk to detect periodic tasks with a too long delay.

Some experiments about this will be reported in Section VI. To address this issue, the PeriodWiz starts by computing $S_i(f)$ based on a small number of samples and increases N if the task is not identified as periodic. When a maximum N^{max} is reached without identifying a period, the task is marked as "not periodic" and N is not increased further.

Another parameter is the period T of PeriodWiz. It has no influence on the sampling frequency of wakeup events, which are registered by the tracing facilities. However, it is important to tune it properly: if T is too long, a period estimate can be available after too much time after enough samples have been collected, while if T is too short, PeriodWiz can spuriously wake up (causing overhead at the operating system level) without that enough samples being collected.

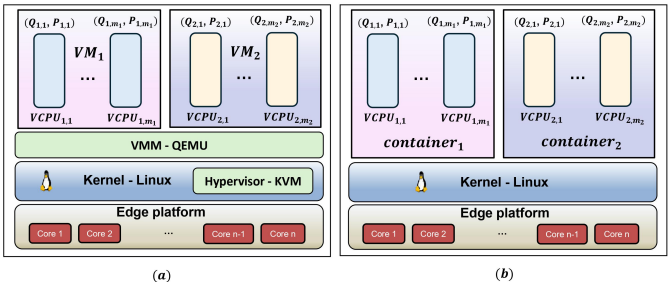
V. From Task Periods to Reservation Periods

Classical real-time systems consider the parameters of each thread to be known and a static workload (no new thread joins at runtime), enabling the design of the vCPU parameters offline, at design time. The considered edge computing context is instead much different and provides a dynamic workload of VM or containers, which need to correspond to a VP v_j. However, deciding the parameters $Q_{j,x}$ and $P_{j,x}$ is a hard task when no prior information about the workload is available.

The tool PeriodWiz presented in this paper allows to detect the periodicity of a task running in a Linux system.

However, once the periods T_i of the tasks $\tau_i \in \Gamma_j$ assigned to a VP v_j have been obtained, they must be used to configure the VP itself, i.e., the budgets $(Q_{j,1}, \dots Q_{j,m_j})$ and periods $(P_{j,1}, \dots P_{j,m_j})$, and possibly also the number of vCPUs m_j.

Clearly, this requires also knowing the execution time C_i of each task: this parameter can be estimated by extending PeriodWiz to trace context switch events too, or can be measured with state-of-the-art tracing tools such as perf [19]. As an alternative, feedback scheduling techniques [20] can be used to adapt the runtimes $Q_{j,i}$ to the tasks' execution times. Different design alternatives are possible, depending on how the VPs are implemented. Figure 1 shows two possible options: inset (a) refers to the case in which VPs are implemented by KVM-like virtual machines, and inset (b) considers the case in which the VP are implemented by containers.



(a) (b)

Fig. 1. Two different ways of instantiating a virtual platform: inset (a) shows the case in which the VP is a virtual machine managed by KVM/QEMU; inset (b) considers the case in which the VP is a container.

A. The one-vCPU-per-task approach

If the VP is implemented by a container, the host operating system has complete visibility of all the tasks running in the container. Therefore, a simple - yet effective - option could be to assign a `SCHED_DEADLINE` vCPU to each task, setting its budget to $Q_{j,i} \geq C_i$ and $P_{j,i} \leq T_i$.

This parameter assignment guarantees that each job of τ_i always receives at least the C_i time units required to complete before the next activation, which occurs with period T_i [5]. Clearly, this can only be guaranteed if the physical platform is not overloaded. For example, if vCPUs are assigned to physical cores following a partitioned scheduling approach, the physical core in which $c_{j,i}$ is allocated must not be overloaded: this must be verified by checking that the sum of the ratios of the budgets and periods of all the vCPUs allocated to a physical core is less than or equal to one [21].

When using this approach, the number of vCPUs m_j is equal to the cardinality of set Γ_j ($|\Gamma_j|$).

When VPs are implemented by a KVM-like virtual machine, this approach is not possible. Indeed, the host operating system has no visibility about the tasks running inside the VM but only visibility about the Linux processes that implement the virtual CPUs of the virtual machine. Also, scheduling the individual tasks with the `SCHED_DEADLINE` policy of the guest kernel would not lead to the intended temporal behavior since the VM is subject to the scheduling effects occurring at the host operating system level. Since the guest kernel sees the host's "real time", every time the VM is preempted, the tasks running in the guest would be accounted for the wrong runtimes (including the time for which the VM did not run).

To overcome this issue, the *m-vCPU approach* can be used.

B. The m-vCPU approach

The m-vCPU approach considers a fixed number m_j of vCPUs to implement the VP v_j, allowing vCPUs to manage multiple tasks. As previously discussed, this approach is more natural for using `SCHED_DEADLINE` reservations for the processes implementing the vCPUs of a VM under KVM-like virtualization.

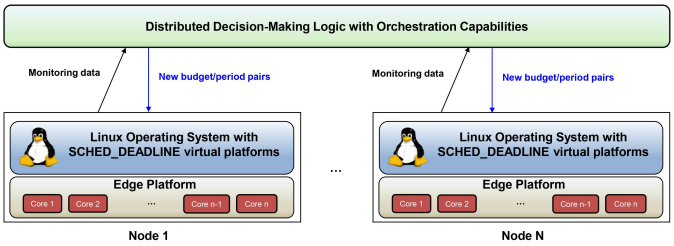


Fig. 2. SCHED_DEADLINE in an Edge-based decision-making architecture

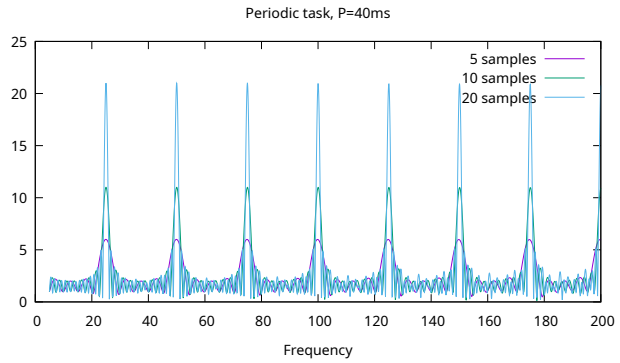


Fig. 3. Energy of the Fourier transform of the activations for a periodic task with period $T = 40ms$, for different numbers of samples.

Furthermore, this approach can also be used when using containers to simplify the decision-making problem: for example, if m_j is fixed, the budget and periods of all the vCPUs can be set to the same value ($Q_{j,1} = \ldots = Q_{j,m_j} = Q_j$) and ($P_{j,1} = \ldots = P_{j,m_j} = P_j$), and the set of parameters needed to identify the timing behavior of a VP just consist of the triplet (Q_j, P_j, m_j).

In this case, suitable budget and period parameters can be achieved using methods from the real-time systems literature for the design of the parameters of reservation servers. A vast literature exists on this topic; however, since in an edge architecture these parameters need to be defined online, we refer the interested reader to works [7], [22] that provides heuristics methods for designing the reservation budgets and periods in a few milliseconds.

C. Running PeriodWiz inside a VP

The implementation choice for the virtual platform not only influences the assignment of the reservation parameters from tasks' parameters but also affects how PeriodWiz can be used.

If the virtual platform in which the application is running is based on a Docker-like container, the host kernel has complete visibility of the application's tasks, hence the PeriodWiz daemon can run on the host and can trace the tasks to identify their periods without issues.

If, instead, the virtual platform is based on KVM-like virtualization, then the host kernel only sees the VMM's vCPU threads. Hence, PeriodWiz cannot directly trace the application's tasks to detect their periods. In this second case, there are various possibilities:

- The PeriodWiz daemon can be executed inside the VM; in this case, if the host scheduler does not affect the applications' activation pattern, the daemon can still detect periodic applications and their periods.
- The PeriodWiz daemon can be executed in the host to analyze the activation pattern of the vCPU threads.
- The applications' activation patterns can be detected before starting the applications in the virtual platform by running the application in a container or on a different node, where PeriodWiz can analyze it.

We consider some of these cases later in Section VI.

D. PeriodWiz in an Edge decision-making architecture

Figure 2 shows a reference architecture for edge systems using `SCHED_DEADLINE` reservations. The figure shows that a distributed runtime decision-making logic with orchestration capabilities (e.g., implemented with `SCHED_DEADLINE`-aware versions of Kubernetes [23] or OpenStack [1]).

In this context, the orchestrator receives offloading requests for applications from mobile devices, which require to be allocated in a VP on the available edge nodes. Using `SCHED_DEADLINE`, this involves setting the budget and period parameters. Computing nodes report to the orchestrator monitoring data and receive updated values for the budgets and periods of the vCPUs implementing each VP. When an offloading request is received by the orchestrator, the periodicity of the application's tasks is estimated with PeriodWiz. As previously discussed, PeriodWiz can be either running on the same edge node of the deployed VP or in a remote node (e.g., together with the orchestrator).

If PeriodWiz is deployed on the same edge node, the edge node can sporadically report the periods of the observed tasks to the orchestrator, which uses them to decide refined reservation parameters and communicate them to the edge node in which the VP is deployed.

If PeriodWiz is deployed on a remote node, the node running the VP communicates to the remote node running PeriodWiz which are the tasks to be observed. The remote node sporadically informs the orchestrator about the observed task periods, which are again used to derive new budgets and periods for reservation servers to be communicated to the edge node running the VP.

Finally, it is worth noting that offloading requests can be associated with a maximum admission delay: such a delay directly influences the number of samples N used for making the period estimate (explained in Section IV).

If the observation period of PeriodWiz is too short, the number of samples is too low, and the period estimate is inaccurate. Nevertheless, if the observation period is too long, the maximum admission delay constraint can be violated. To find a trade-off, it is suggested to incrementally increase the number of samples as discussed in Section IV.

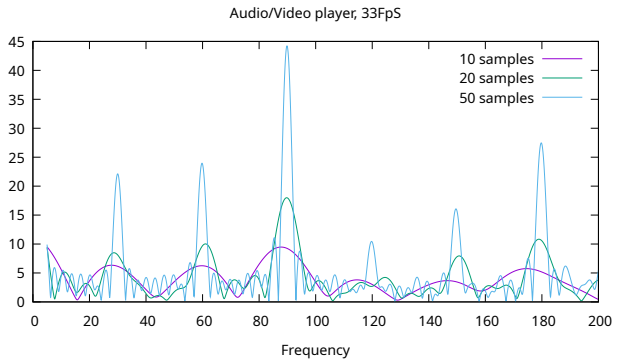


Fig. 4. Energy of the Fourier transform of the activations for an audio/video player, for different numbers of samples.

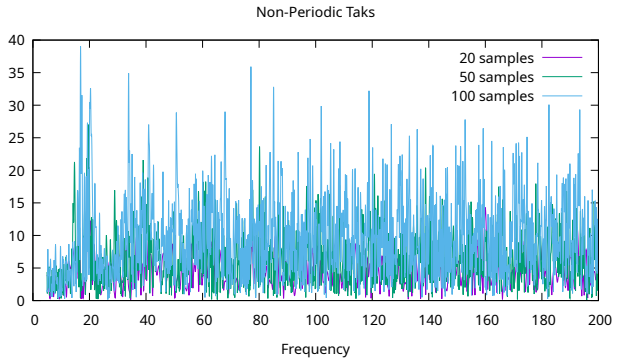


Fig. 5. Energy of the Fourier transform of the activations for a non-periodic task, for different numbers of samples.

VI. Evaluation

We now evaluate the effects of the number N of samples using our frequency-estimation mechanism on 3 different applications: a synthetic real-time application composed of a periodic task with period $P = 40ms$, the `ffplay` video player reproducing a video (with its synchronized audio track) at 33 frames per second (FPS), and a non-periodic application performing some processing on data stored on the disk.

Figure 3 shows the energy $S_i(f)$ of the Fourier transform for the periodic task, with $N \in \{5, 10, 20\}$ samples. The figure shows how increasing the number of samples increases the energy of the frequency peaks, making it easier to detect them. Our tool is able to identify the program as periodic (with the correct period $T = 40ms$) when $N = 10$ or $N = 20$ samples are used; hence, the minimum delay for identifying the task as periodic is $\delta = 40ms \cdot 10 = 400ms$.

Figure 4 shows the energy $S_i(f)$ of the Fourier transform for the audio/video player, computed on $N \in \{10, 20, 50\}$ samples (in this case, $N = 5$ did not provide any useful information). Although this application does not exhibit a clearly periodic activation pattern (it has to display a video frame every $33.3ms$, to periodically decode and play the audio track, to read the compressed data from disk, etc...) the energy $S_i(f)$ allows to identify some peaks. However, such peaks are visible only when enough samples are used; in particular, the application is able to identify them for $N = 50$.

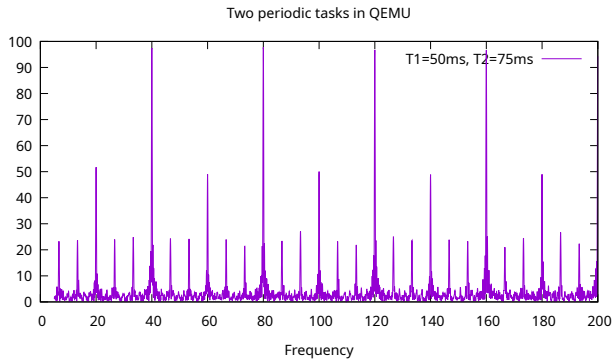


Fig. 6. Energy of the Fourier transform of the activations for the QEMU vCPU thread when two real-time tasks with periods $T_1 = 50ms$ and $T_2 = 75ms$ run inside the VM.

Finally, Figure 5 shows the energy $S_i(f)$ of the Fourier transform for a non-periodic application, computed on $N \in \{20, 50, 100\}$ samples. In this case, it is clearly not possible to identify any peaks in the energy function, and PeriodWiz marks the application as "not periodic".

Moreover, PeriodWiz has been tested to analyze the activation patterns of various periodic tasks, using `cyclictest`, `rt-app`[2] and some synthetic real-time applications, and it was always able to correctly identify such applications as periodic (with the correct period). It has also been tested with some "almost periodic" applications (such as audio/video players), and it was able to detect periodic activation patterns, even if with some surprises. For example, an audio/video player reproducing video at 30FpS was detected as periodic with a period $T = 11.1ms$, probably because audio decoding/reproduction and file parsing introduced some high-frequency components. Nevertheless, $11.1ms$ is a suitable period for an application at 30FpS (hence with a video period of $33.3ms$) since it is a sub-multiple of the video period.

Finally, some experiments have been performed to check how PeriodWiz performs when trying to identify tasks running in a QEMU/KVM VM. To this end, some periodic task sets have been executed inside QEMU/KVM VMs, using the PeriodWiz daemon to analyze the activation pattern of the QEMU's vCPU threads. For example, running two periodic real-time tasks with periods $T_1 = 50ms$ and $T_2 = 75ms$ in a single-CPU VM, PeriodWiz identifies the vCPU thread as periodic with period $T = 25ms$; Figure 6 shows the energy of $S(f)$, which has a peak in $f = 40Hz$ (corresponding to $T = 1000ms/40 = 25ms$) allowing to identify the period. Similarly, Figure 7 shows the energy of the Fourier transform when three tasks with periods $T_1 = 45ms$, $T_2 = 60ms$, and $T_3 = 105ms$ run inside the VM. In this case, PeriodWiz identifies the vCPU thread as periodic with period $15ms$. More experiments revealed that PeriodWiz is generally able

[2] *cyclictest:* https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start, *rt-app:* https://github.com/scheduler-tools/rt-app

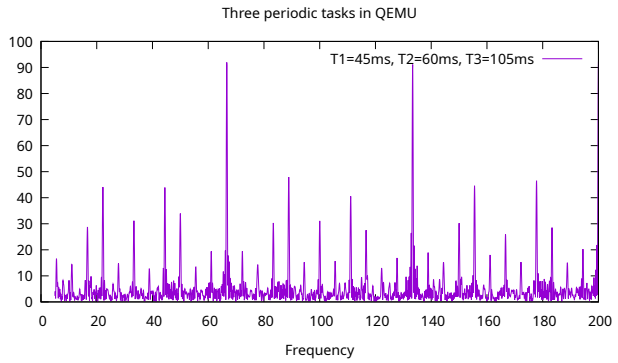


Fig. 7. Energy of the Fourier transform of the activations for the QEMU vCPU thread when three real-time tasks with periods $T_1 = 45ms$, $T_2 = 60ms$, and $T_3 = 105ms$ run inside the VM.

to identify the greatest common divisor of the periods of the tasks running in the VM; this is actually a very good choice for the vCPU's reservation period. Hence, we conclude that this approach is usable for hypervisor-based VMs, too.

VII. Conclusions

This paper presented an approach for scheduling time-sensitive applications in virtual platforms hosted by edge nodes. Since the platforms' virtual CPUs are scheduled through CPU reservations based on the `SCHED_DEADLINE` scheduling policy provided by the Linux kernel, it is essential to identify the applications' periodicity to properly set the budget and period parameters of `SCHED_DEADLINE` reservations, allowing to achieve the desired bandwidth and maximum service latency for each vCPU. To this end, we implemented a tool, named PeriodWiz, which is able to trace the tasks running on the edge node and to identify their activation patterns.

Some preliminary experiments on the PeriodWiz prototype showed that it is able to correctly identify periodic processes and threads and to characterize the activation patterns of "almost periodic" ones. This information can be combined with some kind of runtime estimation (e.g., based on feedback scheduling) to setup the `SCHED_DEADLINE` scheduling parameters; as future work, we plan to implement this integration. Moreover, the usage of the PeriodWiz daemon inside virtual machines still has to be fully investigated (preliminary experiments showed that PeriodWiz running in the host is able to identify periodic patterns in the VM's vCPU threads, but no work on PeriodWiz as a VM guest has been performed yet).

Acknowledgment

This work has been partially supported by the European Union's Horizon Europe Framework Programme project NANCY under the grant agreement No. 101096456, and the project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU.

References

[1] T. Cucinotta, L. Abeni, M. Marinoni, R. Mancini, and C. Vitucci, "Strong temporal isolation among containers in openstack for nfv services," *IEEE Transactions on Cloud Computing*, 2021.

[2] V. Struhár, S. S. Craciunas, M. Ashjaei, M. Behnam, and A. V. Papadopoulos, "Hierarchical resource orchestration framework for real-time containers," *ACM Trans. Embed. Comput. Syst.*, vol. 23, no. 1, jan 2024. [Online]. Available: https://doi.org/10.1145/3592856

[3] E. Quiñones *et al.*, "The AMPERE project: : A model-driven development framework for highly parallel and energy-efficient computation supporting multi-criteria optimization," in *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, 2020.

[4] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli, "Deadline scheduling in the Linux kernel," *Software: Practice and Experience*, vol. 46, no. 6, pp. 821–839, 2016.

[5] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*. IEEE, 1998, pp. 4–13.

[6] A. Mok, X. Feng, and D. Chen, "Resource partition for real-time systems," in *7th IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2001, pp. 75–84.

[7] L. Abeni, A. Balsini, and T. Cucinotta, "Container-based real-time scheduling in the linux kernel," *ACM SIGBED Review*, vol. 16, no. 3, pp. 33–38, 2019.

[8] T. Cucinotta, F. Checconi, L. Abeni, and L. Palopoli, "Adaptive real-time scheduling for legacy multimedia applications," *ACM Trans. Embed. Comput. Syst.*, vol. 11, no. 4, jan 2013. [Online]. Available: https://doi.org/10.1145/2362336.2362353

[9] L. Tung, "Spacex: We've launched 32,000 linux computers into space for starlink internet," 2020.

[10] B. B. Brandenburg and M. Gül, "Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations," in *2016 IEEE Real-Time Systems Symposium (RTSS)*, 2016, pp. 99–110.

[11] J. Lelli, D. Faggioli, T. Cucinotta, and G. Lipari, "An experimental comparison of different real-time schedulers on multicore systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2405–2416, 2012.

[12] X. Feng and A. K. Mok, "A model of hierarchical real-time virtual resources," in *Proc. of 23rd IEEE Real-Time Systems Symposium*, 2002.

[13] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *24th IEEE Real-Time Systems Symposium*, 2003.

[14] L. Almeida and P. Pedreiras, "Scheduling within temporal partitions: response-time analysis and server design," in *Proc. of 4th ACM International Conference on Embedded Software*, Sep. 2004, pp. 95–103.

[15] E. Bini, M. Bertogna, and S. Baruah, "Virtual multiprocessor platforms: Specification and use," in *2009 30th IEEE Real-Time Systems Symposium*, 2009, pp. 437–446.

[16] A. Easwaran, I. Shin, and I. Lee, "Optimal virtual cluster-based multiproc. scheduling," *Real-Time Systems*, vol. 43, no. 1, pp. 25–59, 2009.

[17] H. Leontyev and J. H. Anderson, "A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees," *Real-Time Systems*, vol. 43, no. 1, pp. 60–92, 2009.

[18] T. Cucinotta, F. Checconi, L. Abeni, and L. Palopoli, "Self-tuning schedulers for legacy real-time applications," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 55–68. [Online]. Available: https://doi.org/10.1145/1755913.1755921

[19] A. C. De Melo, "The new linux 'perf' tools," in *Slides from Linux Kongress*, vol. 18, 2010, pp. 1–42.

[20] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao, "Feedback control real-time scheduling: Framework, modeling, and algorithms," *Real-Time Systems*, vol. 23, pp. 85–126, 2002.

[21] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[22] L. Abeni, A. Biondi, and E. Bini, "Partitioning real-time workloads on multi-core virtual machines," *Journal of Systems Architecture*, vol. 131, p. 102733, 2022.

[23] S. Fiori, L. Abeni, and T. Cucinotta, "Rt-kubernetes: containerized real-time cloud computing," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, 2022, pp. 36–39.