Runtime Monitoring for Edge Applications

Daniel Casini[1,2], Luca Abeni[1], Mauro Marinoni[1], and Alessandro Biondi[1,2]
[1]*TeCIP Institute, Scuola Superiore Sant'Anna, Pisa, Italy*
[2]*Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy*

***Abstract*—In edge computing, it is common to lack sufficient computational resources to enable a physical separation between applications, making resource sharing unavoidable. However, resource-sharing techniques must ensure that mutually untrusted applications colocated on the same nodes and core to meet timing constraints. To this end, the Linux `SCHED_DEADLINE` scheduler offers timing isolation and real-time guarantees through resource reservation. However, configuring its parameters is complex, especially in the case of highly variable workloads that are common in dynamic environments, e.g., as in mobility. This paper presents an adaptive monitoring mechanism to dynamically reconfigure `SCHED_DEADLINE` reservations based on runtime workload behavior. A kernel patch exports execution statistics, enabling real-time parameter adjustments: it increases the runtime allocated when a thread exhausts it before completing, and it reduces the runtime when the resource reservation is underutilized. Our prototype is implemented in Rust and supports the monitoring of threads, process groups, containers, and VMs. Evaluation results demonstrate the flexibility of the tool, showing how it is capable to adapt to different scenarios.**

***Index Terms*—edge computing, Linux scheduling, resource reservation, isolation, real-time systems, monitoring.**

I. Introduction

In recent years, the edge computing paradigm has gained increasing interest due to the need to keep computations closer to where data originates, providing advantages in terms of latency, energy consumption, and privacy over traditional cloud-based approaches that require data to be sent to centralized servers for processing [1]. However, resources are more limited at the edge than in the cloud; therefore, efficiently sharing edge computing nodes among various applications from different tenants is crucial to achieving high resource utilization.

A case in point is the mobility sector, where modern and potentially autonomous vehicles must offload computationally intensive tasks to the closest edge server along the movement path. Traditional allocation techniques typically partition computational resources in a coarse-grained manner, assigning applications to cores or nodes in an exclusive way.

While it has the benefit of simplicity, such an approach can lead to significant underutilization of computing platforms, e.g., when lightweight workloads are assigned an entire core due to the need for timeliness and timing isolations from other, untrusted, workloads.

Linux-based edge nodes represent an excellent solution, as they can provide low latency [2], [3] even for virtualized applications [4], are flexible, and are compatible with a broad range of software stacks (e.g., AI frameworks) thanks to a large availability of device drivers and low-level support (e.g., for hardware accelerators) that may be incompatible with specialized real-time operating systems such as VxWorks or FreeRTOS.

Such systems can leverage the `SCHED_DEADLINE` kernel scheduler [5], which allows for fine-grained virtualization of computational resources, enabling flexible and efficient resource sharing while ensuring strict timing guarantees based on theoretical guarantees [6]. The `SCHED_DEADLINE` scheduler implements a *resource reservation* mechanism, encapsulating each application within a virtual platform (VP) composed of multiple virtual CPUs (vCPUs), each receiving a guaranteed fraction of the overall CPU bandwidth and a bounded latency (service delay) [7].

In addition to *resource partitioning*, `SCHED_DEADLINE` also provides *resource enforcement* so that each application receives no more than its allocated CPU bandwidth. Such an enforcement is crucial in multi-tenant environments, where different applications may not trust each other, and potential software bugs or cyber-attacks suffered by an application can affect the timing behavior of another application that may also have a higher criticality.

`SCHED_DEADLINE` allows to provide resource reservation and real-time guarantees in various configurations: from single Linux threads to entire virtual machines (VMs) (e.g., KVM/QEMU [8], [9]), up to containerized workloads. The latter case requires an out-of-tree patch [10], which extends `SCHED_DEADLINE` to allow multiple Linux processes to be grouped into the same reservation using cgroups.

However, efficiently configuring `SCHED_DEADLINE` reservations is challenging and requires tuning the budget parameter (also called runtime, and denoted by Q) and period (P) for each reservation. These parameters directly determine the fraction of CPU bandwidth allocated ($\alpha = Q/P$) and the worst-case latency delay ($\Delta = 2 \cdot (P - Q)$). This relation is due to the theoretical properties of the scheduling algorithm implemented by `SCHED_DEADLINE` [11], [12].

Improper configurations may lead to performance issues: a too-small budget or a too-large period can violate real-time constraints of the application leveraging improperly configured `SCHED_DEADLINE` reservations, while a too-large budget or a too-small period can lead to underutilization of the node.

In traditional safety-critical real-time systems, Q and P are set during standard system design activities, which take place offline, based on the known temporal properties of tasks, such as worst-case execution time and activation period [13].

However, this approach is infeasible in dynamic environments, such as edge computing platforms supporting mobile

applications, where workloads arrive unpredictably, and their timing characteristics are often unknown. Furthermore, in highly dynamic distributed edge applications, the workload assigned to a reservation can vary significantly over time due to external factors such as user mobility and time-of-day fluctuations (e.g., higher computational demand during peak hours and lower demand at night). In such scenarios, static reservation parameters cannot guarantee efficient resource utilization.

Contribution. This paper investigates dynamic runtime monitoring mechanisms for SCHED_DEADLINE-based reservations, aiming to adaptively configure budget parameters based on observed workload characteristics. Specifically, our approach leverages the CPU time accounting mechanisms provided by SCHED_DEADLINE to dynamically monitor the execution time of tasks without requiring code instrumentation. A kernel patch exports execution statistics to user space, allowing for precise runtime tracking. When a task consistently exhausts its allocated runtime before completing, its budget is increased to avoid violating real-time constraints. Conversely, if a task does not fully use its reserved time, the allocation is reduced to minimize wasted resources. To validate our approach, we implement a runtime monitor in Rust, an efficient and memory-safe programming language. The monitor can track individual threads, groups of threads (using Linux cgroups), containerized workloads (e.g., Docker, Kubernetes), and virtual machines (by monitoring their vCPU threads). We perform extensive experiments in different scenarios to show it allows dynamically tuning SCHED_DEADLINE reservations in edge computing platforms to balance the need for temporal isolation and timeliness with optimized resource utilization, even in highly dynamic environments characterized by workload variations.

II. Problem Statement

The SCHED_DEADLINE scheduling policy [5] provided by the Linux kernel implements a resource reservation mechanism to allow encapsulating tasks (threads or processes) into virtual platforms, each with a guaranteed fraction of the core CPU bandwidth and with a bounded CPU service delay [7]. In essence, SCHED_DEADLINE allows guaranteeing that a task executes for Q time units (the so-called budget, or runtime) every period P, enforcing temporal isolation and providing the task with a configurable fraction Q/P of CPU time (named "CPU bandwidth" from now on) with bounded service delay (latency). The resource reservation mechanism implemented by SCHED_DEADLINE is based on the Constant Bandwidth Server algorithm [6] and both provide a resource partitioning and an enforcement mechanism. The latter feature is particularly important in open systems, where different applications may not trust each other. SCHED_DEADLINE ensures that each application receives no more than the allocated CPU bandwidth, thus shielding other applications from possible misbehaviours of malicious (or simply bugged) applications that may otherwise harm their real-time behaviour. For example, consider two co-located applications in which one

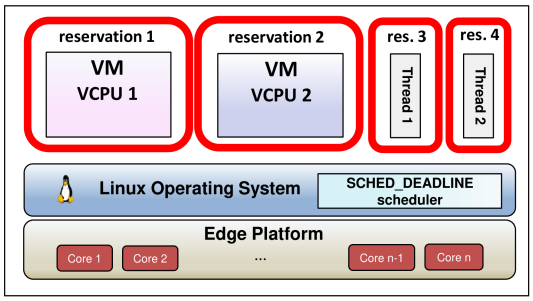

Fig. 1. Scheduling Virtual Machines or threads with SCHED_DEADLINE reservations. The red borders represent the isolation boundaries provided by SCHED_DEADLINE reservations.

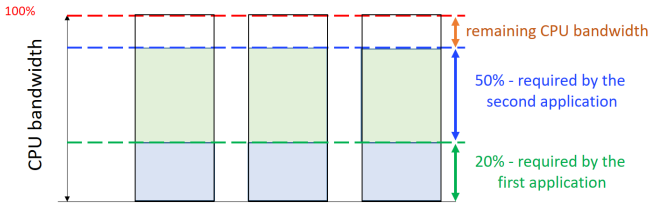

Fig. 2. Example of temporal isolation with SCHED_DEADLINE.

starts to execute continuously due to a cyber-attack, interfering with the other. Without adequate protections, this can cause considerable delays or a denial-of-service. If applications are instead encapsulated within SCHED_DEADLINE reservation, the budgeting mechanisms ensures that the attacked application executes no more than for the assigned CPU fraction, thus protecting the other applications and considerably increasing the security from timing attacks occurring at the CPU level.

The SCHED_DEADLINE policy can be used to serve either Linux threads and processes, virtual machines (e.g., KVM/QEMU), or containers. The first two options are shown in Figure 1. Threads, processes, and VMs are supported by the mainline Linux scheduler, while an out-of-tree kernel patch is required to manage containers [10].

Each SCHED_DEADLINE CPU reservation must be properly configured by assigning an appropriate runtime and period (the two key parameters Q and P).

The theoretical properties of SCHED_DEADLINE [6], [12] make a link between these values and the fraction of CPU bandwidth and the CPU service delay provided to the workload running in the reservation. In particular:

- $\alpha = Q/P$ is the CPU bandwidth provided to the reservation;
- $\Delta = 2 \cdot (P - Q)$ is the CPU service latency.

When reservations are properly configured, SCHED_DEADLINE allows different applications to be co-located in the same platform (e.g., edge node) thanks to the resource isolation mechanism (implemented by budget enforcement). Figure 2 shows the advantage: two (temporally)-untrusted applications requiring both three cores in parallel to perform a parallel operation, one with a requirement of 50% of the core bandwidth and one with the 20% would

be allocated to six cores using classical coarse-grained allocation solutions (i.e., each application is exclusively allocated to an appropriate number of dedicated cores to avoid CPU interference). Differently, SCHED_DEADLINE allows co-locating multiple applications on three cores, also leaving some spare bandwidth for other tasks.

As mentioned, the SCHED_DEADLINE parameters (runtime and period) have to be properly configured to provide the desired properties. Indeed, an inaccurate configuration of a CPU reservation may lead to the following issues:

- If the budget is too small or the period is too large real-time constraints cannot be guaranteed.
- If the budget is too large or the period is too small the edge node can be underutilized.

In traditional (safety-critical) real-time systems, the runtime and period parameters are set based on the temporal properties of the tasks running within the reservation, i.e., the (maximum) execution time and the period, which are known a-priori [7], [13], [14]. However, the situation is much more complex in the context of open systems in which the workloads are dynamic.

Indeed, platform-specific execution time estimates are often unknown or not accurate and applications can even have non-periodic activation patterns. Hence, tools for estimating these parameters are needed. While the problem of estimating an appropriate value for the SCHED_DEADLINE period has been previously addressed [15], [16], assigning an appropriate runtime to an offloaded application is still an open issue for modern systems based on SCHED_DEADLINE.

This paper focuses on the configuration of the runtime parameter, providing an integrated solution to monitor the computational requirements of the virtualized workload running inside a reservation and tuning the reserved CPU time accordingly. The provided solution directly uses the mechanisms provided by SCHED_DEADLINE and is suitable for dynamic workloads because it does not require code instrumentation.

III. Dynamic Detection of Runtimes

When scheduling one or more activities (e.g., single threads, processes, containers, VMs) with SCHED_DEADLINE (possibly using the real-time control group scheduling patchset [10] in the case of containers), the experienced QoS can be controlled if the tasks' periods and runtimes are known. As previously discussed, the QoS of virtualized software workload is controlled by the SCHED_DEADLINE parameters (runtime and period) and the activation period of one or more activities can be estimated through tracing and frequency-based analysis [15].

Nevertheless, even with a suitable period, the problem of monitoring the tasks' runtimes still needed to be addressed. This information is crucial to set the budget for SCHED_DEADLINE reservations.

This scheduling approach–called *dynamic detection of runtimes*–is similar to feedback scheduling [17], [18] (and adaptive reservations [19] in particular). However, adaptive reservations are implemented by reading some observed value (the so-called *scheduling error*) used by a control algorithm to set some actuator (the reserved runtime). An overview of our approach is shown in Fig. 3.

Observing the scheduling error requires instrumenting the code to mark the beginning (and/or the end) of each activation of the real-time task (named "job" in real-time jargon). For example, a periodic task can be implemented as in Figure 4, where the `wait_for_next_activation()` call marks the end of each periodic activity. Unfortunately, it is often not possible to know a-priori the structure of applications offloaded to edge nodes, which could not follow this code structure.

Hence, a different approach based on monitoring the execution time of each task has been adopted. The monitoring mechanism is designed to be applied to generic code, even to non-real-time applications, and takes advantage of the SCHED_DEADLINE features.

As already introduced, this scheduling policy reserves a budget of time units to be executed every period to the scheduled task: these quantities are stored in the `dl_runtime` and `dl_period` variables, respectively.

Then, the CPU time accounting mechanism implemented by the Linux kernel can be used to measure how much time the task actually executed for. As an example, the kernel has been patched to export this information through the `sched_getattr()` system call. Such a system call is generally used to read the scheduling policy (SCHED_DEADLINE, SCHED_FIFO, SCHED_RR, or SCHED_OTHER) and the scheduling parameters (runtime, deadline, period, or priority) of a task and accepts a "`flags`" parameters to allow future extensions. Our patch introduces a new "SCHED_GETATTR_FLAG_RUNTIME" flag that allows reading the amount of time a task has executed. If the amount of time executed by a task in a period of length `dl_period` is smaller than `dl_runtime`, then the reserved runtime can be decreased; otherwise, it must be increased.

The resulting runtime monitor is based on the following design principles:

DP1 SCHED_DEADLINE is used to ensure that a task (or a group of tasks) can execute for at most Q time units (`dl_runtime`) every P time units (`dl_period`);

DP2 A kernel patch is used to periodically monitor the amount of execution time used by each task;

DP3 If a task (or group of tasks) executes for the maximum reserved time, it is assumed that the reserved time is too small. Hence, Q is increased;

DP4 If the initial value of Q is too small (e.g., because no initial estimate of the execution time is available at the beginning), then a too long time can elapsed until the reserved time is enough:

- To compensate this effect, if the tasks consume the whole reserved time for multiple times in a row, then the *"speed"* at which Q is increased is accelerated.
- This feature is implemented by using a variable `l` - called *adaptation rate* - which is doubled every time that executed time increases and is set to 1 when the

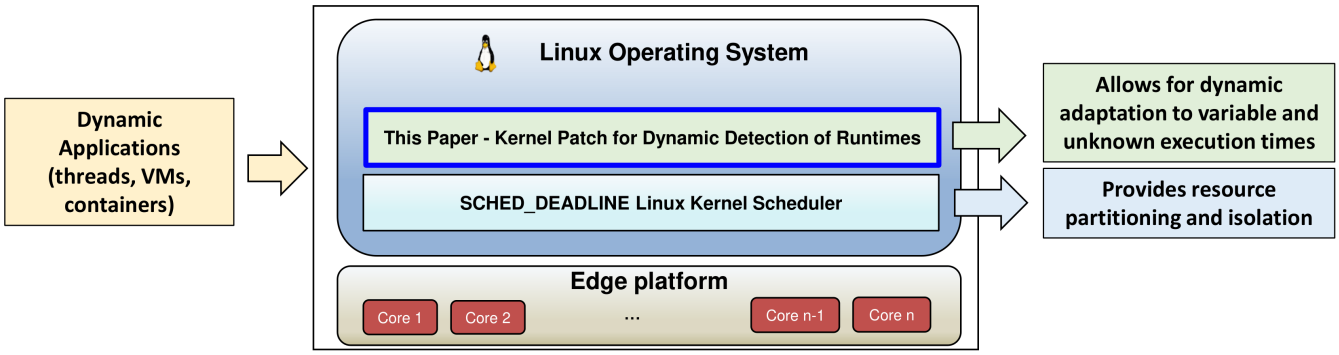

Fig. 3. Overview of the proposed runtime monitoring framework, with the kernel-level patch, dynamic adaptation, and multi-scheduling entity applicability.

```
while (!finished) {
  wait_for_next_activation();

  /* ...do something... */
  /*    (job body)    */
}
```

Fig. 4. Example of periodic task.

executed time does not increase.

Notice that in order for the feedback algorithm to work, the allocated runtime cannot be set exactly to the maximum measured execution time, but it must be set to a slightly larger value (otherwise the tasks would not be able to execute for more than the detected time, and the allocated runtime would not increase).

Hence, if some tasks execute for C time units over P time units, Q is not set as $Q = C$ but as $Q = C*(1+o_{\text{vh}})$, with o_{vh} defined as the *allocation overhead*. Finally, to account for the *adaptation rate* l mentioned above, the equation is updated as $Q = C \cdot (1 + o_{\text{vh}} \cdot l)$.

The pseudocode of the algorithm is reported in Algorithm 1.

In the algorithm, a circular array is used to compute the maximum of the last N samples (line 17). N is the size of the circular array - i.e., the number of previous samples used to compute the maximum and o_{vh} is the overallocation overhead. N and o_{vh} are configurable parameters of the algorithm.

The adaptation rate is initially set to 1 (line 10). The variable `cmax_old`, which stores the last observed maximum execution time in the reservation, is initially set to -1, in such a way that the adaptation process is performed only if at least two execution time values have been observed (line 18, for the startup phase).

The algorithm doubles the adaptation rate if the maximum execution time observed in the sliding window of N instances (`cmax`) is greater than the maximum execution time observed in the previous activation `cmax_old`, meaning that the allocated budget is not enough (line 19). Otherwise, the adaptation rate is set to 1 (line 21).

Finally, the new budget (`dl_runtime`) is set (line 25).

Please note that, for simplicity in the presentation, the monitor in Algorithm 1 considers a single thread in a reservation. Nonetheless, our algorithm and implementation can be used to monitor multiple reservations simultaneously, regardless of the encapsulated scheduling entity (threads, VMs, containers).

Algorithm 1 Runtime Monitoring and Update

```
1:  function MONITORANDUPDATE(P, Q, ID)
2:      Input: P (task period)
3:      Input: Q (initial runtime estimation)
4:      Input: ID (task identifier)
5:      Parameter: N (size of the circular array)
6:      Parameter: o_vh (overallocation overhead)
7:                                  ▷ Set sched. params for res. ID
8:      dl_runtime ← Q; dl_period ← P
9:                                  ▷ Set algorithm parameters
10:     l ← 1;                      ▷ Adaptation rate
11:     cmax_old ← -1;              ▷ Last max. exec. time
12:     circular_array ← circular array of size N
13:     while true do               ▷ Start periodic monitoring
14:         wait for T time units
15:         c ← GET_EXECUTED_TIME(ID)
16:         INSERT(circular_array, c)
17:         cmax ← GET_MAX(circular_array)
18:         if cmax_old ≠ −1 then
19:             if cmax > cmax_old then
20:                 l ← l · 2
21:             else
22:                 l ← 1
23:         cmax_old ← cmax
24:                                 ▷ Budget adaptation
25:         dl_runtime ← dl_runtime ·(1 + o_vh · l)
```

IV. Implementation and Usage

The proposed runtime monitoring and adaptation algorithm has been implemented in a runtime monitor which is able to monitor single threads (as described above) or groups of threads (using the Linux cgroups feature). Docker/Podman (or Kubernetes) containers are handled through their cgroups, while KVM-based VMs can be handled by monitoring their virtual CPU threads (running a monitor as a daemon inside the

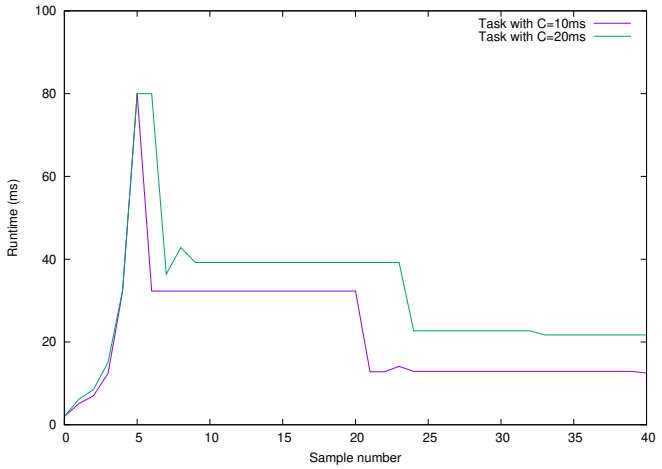

Fig. 5. Estimated runtimes for two periodic real-time tasks, starting from unrealistically low initial estimations.

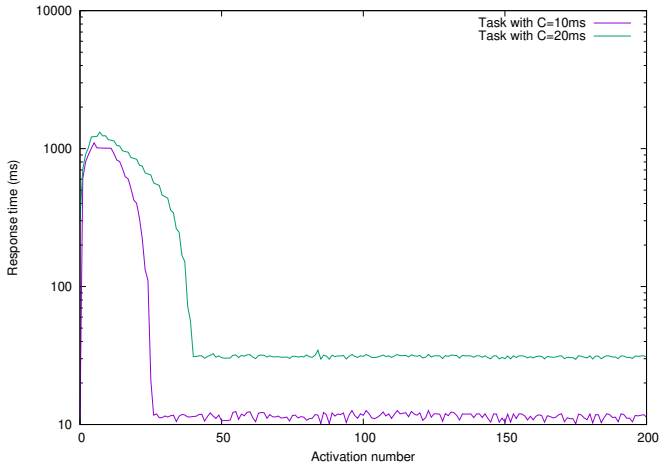

Fig. 6. Response times for the two periodic real-time tasks of Figure 5.

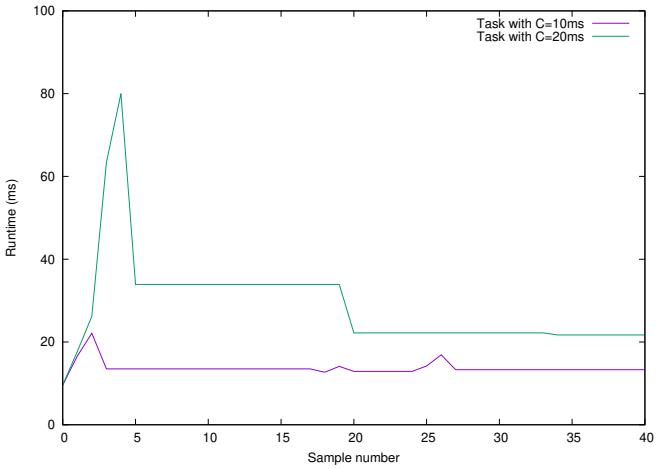

Fig. 7. Estimated runtimes for two periodic real-time tasks, starting from realistic initial estimations.

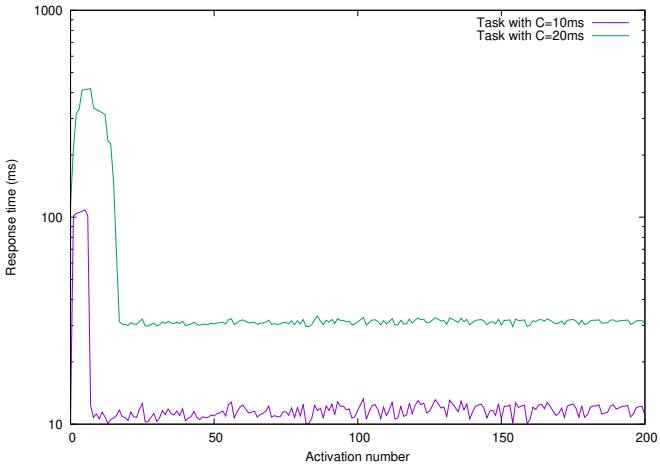

Fig. 8. Response times for the two periodic real-time tasks of Figure 7.

VM is also possible). The monitor needs to be both efficient (introducing a small overhead) and safe (avoiding bugs due to wrong memory accesses), and the Rust language looked like a good compromise between efficiency and safety. Hence, the monitor has been implemented in Rust.

Usage. The monitoring program can be used in different ways:

- As a wrapper that is able to start the monitored programs and monitor/manage them.
- As a standalone program that can monitor one or more existing threads (also setting the reserved runtime).
- As a daemon providing a REST API, which can receive the IDs of the threads to be monitored.

V. Experimental Results

To evaluate the effectiveness and the performance of the presented runtime monitor, experiments have been performed using a node based on an AMD Ryzen 7 5700U CPU running at 1.8 GHz. The node runs Ubuntu 24.04.2 with a custom 6.10 Linux kernel patched as described in Section III.

Robustness to initial runtime misestimation. In a first experiment, we tested the ability of the monitor to infer correct runtimes even when the initial runtime estimations are unrealistically low. Figure 5 shows the evolution of the runtime allocated to two periodic threads with execution time 10ms and 20ms (the two tasks have period 100ms). To stress the robustness of the algorithm, the original runtime estimations have been set to two completely underestimated values (2ms). As it is possible to see from the figure, the allocated runtime rapidly increases to a maximum value (set to 80% of the period), thanks to the multiplicative effect of the adaptation rate `l`; this allows recovering from the delay accumulated by the tasks in the initial periods, when the allocated runtime was not large enough. After this delay is recovered, the allocated runtime decreases to a value slightly larger than the thread's execution time (this small overallocation is due to the overallocation overhead o_{vh}). Figure 6 shows the effect of these runtime allocations on the tasks' response times: the response times initially increase to very large values, but after few

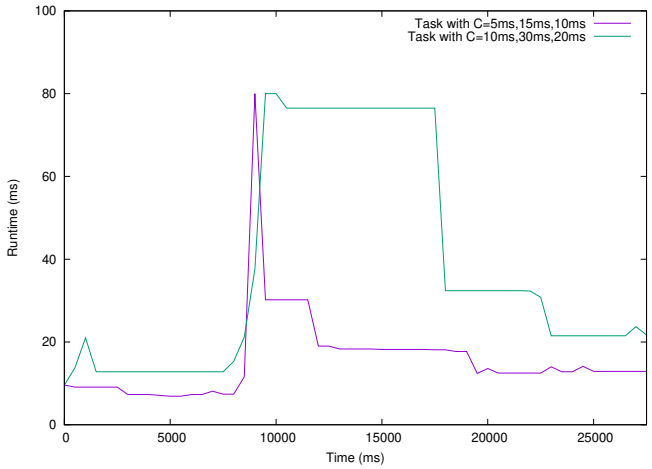

Fig. 9. Estimated runtimes for two periodic real-time tasks with varying execution times.

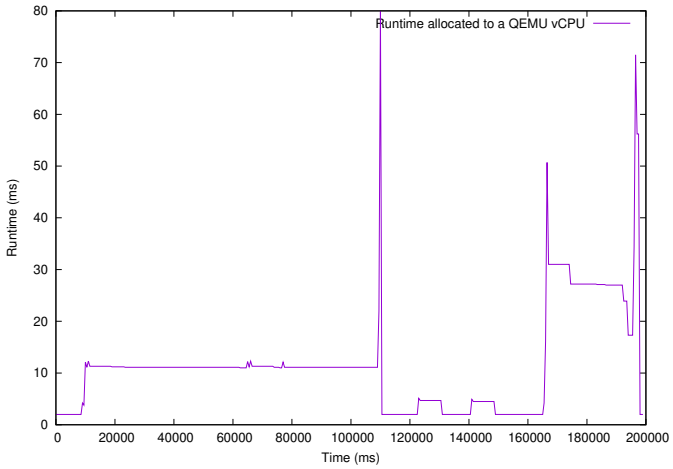

Fig. 10. Estimated runtime for a VM's virtual CPU thread, when executing various real-time applications.

monitoring cycles they become equal to the threads' execution times, showing that the monitoring and feedback algorithms worked correctly.

Experiments with a better initial runtime estimation. After verifying that the monitor can tolerate initial underestimations of the runtimes, the effect of more correct estimations has been tested. Figure 7 and Figure 8 show the results of a different experiment in which the initial runtime estimation was more accurate ($Q = 8ms$). As it is possible to see from the figure, in this case the runtime converges more quickly to the correct value and the response times are much more controlled.

Experiment with a mode switch. To check how the runtime monitor can cope with variations in the runtime, the experiment has been repeated with two tasks having variable execution times: the first one starts with an execution time equal to $5ms$, after some time switches to a different mode of execution, in which it has an execution time equal to $15ms$, and then switches to $10ms$, while the second one has three modes: it starts with an execution time equal to $10ms$, then switches to $30ms$, and then to $20ms$. The evolution of the estimated runtime is displayed in Figure 9, showing that the monitor is able to correctly cope with the changes in the tasks' execution time, quickly adapting to the variations in the tasks' behaviour.

Monitoring a QEMU VM. After verifying that the runtime monitor is able to correctly estimate the runtime needed by tasks executing on the host machine, some experiments have been performed to test its behaviour with VMs.

In particular, a QEMU/KVM VM (with one single virtual CPU) has been started, using the runtime monitor to estimate the runtime of the QEMU's virtual CPU thread (QEMU creates a Linux thread for each virtual CPU). Inside the VM, two real-time applications have been sequentially executed: the first one is composed of one single periodic thread (named "`thread1`" and having period $100ms$ and execution time $10ms$), and the second one is composed of two periodic threads ("`thread2`",

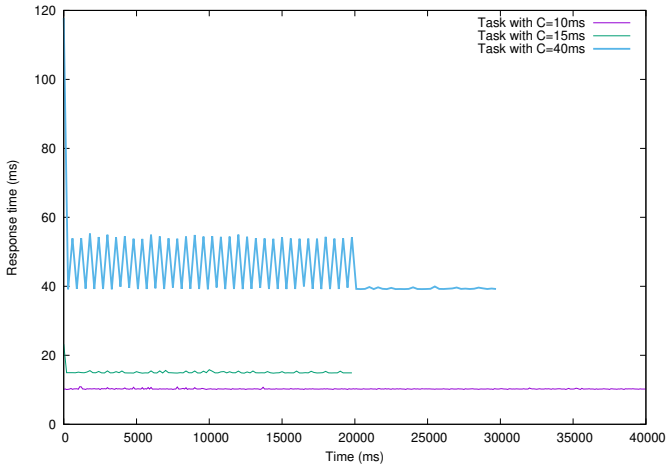

Fig. 11. Response times for the threads of the real-time applications running inside the VM of Figure 10.

with period $200ms$ and execution time about $15ms$, and "`thread3`", with period $300ms$ and execution time about $40ms$).

The estimated runtime is displayed in Figure 10. From the figure it is possible to see that the runtime estimation is originally very low (around $2ms$) because the VM is idle; when the first real-time application (composed of `thread1`, with $(C = 10ms, P = 100ms)$) starts, the runtime is increased to the correct value and stays almost constant during the execution of the application. When the application terminates, the runtime estimation experiences a spike to the maximum value, probably because the guest OS kernel needs some time to save the application's results and to terminate it. Then, the runtime estimation returns around $2ms$ (idle VM), with some slight increases probably due to some activity on the VM console. After some time, a second real-time application (composed of `thread2` with $(C = 15ms, P = 200ms)$ and `thread3` with $(C = 40ms, P = 300ms)$) starts (around

time $17s$) and the estimated runtime increases, stabilizing around the correct value after a small spike. When `thread2` terminates and `thread3` remains the only active real-time thread, the estimated runtime decreases to the new correct value; finally, when the application finishes the estimated runtime experiences a spike (this is similar to what happened when the first application terminated) and then returns to $2ms$ (idle VM).

Figure 11 reports the response times of the three threads, showing that the monitor is able to keep them under control (the only thread experiencing a high response time is `thread3`, at the beginning of its execution when the runtime estimation is not accurate yet). As a side note, it is interesting to notice that when `thread2` finishes the response time of `thread3` stabilizes (because it stops suffering interference from `thread2` execution).

VI. Related Work

Co-locating different applications on physical computational resources has been an open problem for the last two decades [7], [17]. The complexity arises from the heterogeneous constraints of the various application scenarios, the evolution of hardware platforms, the addressed workloads, and the integration with software stacks. In critical applications with limited resources, accurately monitoring resource utilization and assigning the computational platform's share becomes paramount [20], [21]. The problem is characterized by several challenges: accurate workload estimation, proper allocation of shared resources, and the capability to enforce this allocation with robust mechanisms.

Several works have been presented in the literature to exploit different virtualization mechanisms. Abeni et al. [8] proposed a solution to enforce isolation among KVM-based virtual machines exploiting the standard `SCHED_DEADLINE`. They also proposed an analysis whose model is coherent with the implementation details.

Methods for leveraging `SCHED_DEADLINE` for executing deep networks have also been proposed [22].

Containers are becoming the most used and studied in the edge context due to their lightweight approach and limited overhead. Struhár et al. [23] have proposed an overview of the topic. In particular, Telschig et al. [24] presented a real-time container architecture for dependable distributed embedded applications with different criticalities and an implementation based on lxc. Abeni et al. [10] proposed a preliminary extension of the `SCHED_DEADLINE` scheduling class to enforce container isolation. Cucinotta et al. [25] exploited such an isolation mechanism to improve the colocation of containers in edge infrastructures. Barletta et al. [26] studied issues related to the use of container-based solutions in the field of industrial edge nodes. Fiori et al. [27] proposed an extension to the Kubernetes platform called RT-Kubernetes to better exploit the advantages of `SCHED_DEADLINE` to isolate containers in modern orchestration infrastructures. Other works targeted WebAssembly virtualization [28]–[30].

Most of these works focus on implementing isolation mechanisms without considering the problem of adequately tuning the allocated resources and adapting such parameters in the case of dynamic workloads [31].

Their runtime allocation is crucial to handling variable workloads. Initially, authors focused on simple scenarios with single real-time tasks. Lu et al. [18] proposed a scheduling algorithm applying feedback control techniques to handle runtime variations of task execution times. Cervin and Eker [32] presented an approach to schedule control tasks characterized by high computational variability, such as hybrid controllers. Abeni et al. [19] integrated feedback control with bandwidth reservation to guarantee the effective availability of the allocated resources, showing the results of the proposed solution to manage multimedia applications.

Another work [16] proposed a solution for multimedia applications that combines a monitor to observe task activations and a feedback mechanism to adapt scheduling parameters. The solution was based on the ad-hoc API of the AQuoSA project and cannot be seamlessly integrated into modern solutions based on containers and VMs.

Struhar proposed the REACT framework [33] to orchestrate containers with real-time constraints, which was further extended [1].

Barletta et al. [34] proposed k4.0s, an extension based on Kubernetes focusing on the real-time constraints for the industrial application field. Similarly, Lumpp et al. [35] presented RT-Kube, which is an extension of Kubernetes able to monitor deadline violations and adapt container parameters for autonomous robots. In the context of more general performance-sensitive applications, methods to distribute the workload in Amazon AWS have been proposed [36].

VII. Conclusions and Future Work

This paper presented a dynamic monitoring approach for `SCHED_DEADLINE` reservations to dynamically detect reservation budgets at runtime based on observed workload measurements.

The approach and its implementation in Linux enables efficient resource in dynamic real-time edge environments without requiring application instrumentation and leverages Rust to be both efficient and safe.

Future work will focus on integrating runtime monitoring with dynamic period estimation [15] and orchestration [27]. Furthermore, another possible research direction consists of layering a (possibly machine-learning-based) lightweight predictor to forecast workload spikes and pre-adjust the budget proactively.

Acknowledgment

This work has been supported by the European Union's Horizon Europe Framework Programme project NANCY under the grant agreement No. 101096456.

REFERENCES

[1] 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

[2] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole, "A measurement-based analysis of the real-time performance of linux," in *Proc. of 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, Sep. 2002, pp. 133–142.

[3] D. B. de Oliveira, D. Casini, and T. Cucinotta, "Operating system noise in the linux kernel," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 196–207, 2023.

[4] L. Abeni, "Virtualized real-time workloads in containers and virtual machines," *Journal of Systems Architecture*, vol. 154, p. 103238, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762124001759

[5] 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.

[6] 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.

[7] 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.

[8] L. Abeni, A. Biondi, and E. Bini, "Hierarchical scheduling of real-time tasks over linux-based virtual machines," *Journal of Systems and Software*, vol. 149, pp. 234–249, 2019.

[9] ——, "Partitioning real-time workloads on multi-core virtual machines," *Journal of Systems Architecture*, vol. 131, p. 102733, 2022.

[10] 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.

[11] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS 1998)*, Madrid, Spain, December 2-4 1998.

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

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

[14] 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.

[15] L. Abeni, T. Cucinotta, and D. Casini, "Period estimation for linux-based edge computing virtualization with strong temporal isolation," in *2024 IEEE 3rd Real-Time and Intelligent Edge Computing Workshop (RAGE)*. IEEE, 2024, pp. 1–6.

[16] 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

[17] J. A. Stankovic, C. Lu, S. H. Son, and G. Tao, "The case for feedback control real-time scheduling," in *Proceedings of 11th Euromicro Conference on Real-Time Systems*. IEEE, 1999, pp. 11–20.

[18] 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.

[19] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole, "Analysis of a reservation-based feedback scheduler," in *Proc. of the Real-Time Systems Symposium*, Austin, Texas, November 2002.

[20] D. Casini, P. Pazzaglia, and M. Becker, "Managing real-time constraints through monitoring and analysis-driven edge orchestration," *Journal of Systems Architecture*, vol. 163, p. 103403, 2025.

[21] M. Barletta, M. Cinque, L. De Simone, R. D. Corte, G. Farina, and D. Ottaviano, "Partitioned containers: Towards safe clouds for industrial applications," in *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)*, 2023, pp. 84–88.

[22] D. Casini, A. Biondi, and G. Buttazzo, "Timing isolation and improved scheduling of deep neural networks for real-time systems," *Software: Practice and Experience*, vol. 50, no. 9, pp. 1760–1777, 2020.

[23] V. Struhár, M. Behnam, M. Ashjaei, and A. V. Papadopoulos, "Real-Time Containers: A Survey," in *2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)*, ser. Open Access Series in Informatics (OASIcs), vol. 80. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020, pp. 7:1–7:9.

[24] K. Telschig, A. Schönberger, and A. Knapp, "A real-time container architecture for dependable distributed embedded applications," in *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*, 2018, pp. 1367–1374.

[25] T. Cucinotta, L. Abeni, M. Marinoni, A. Balsini, and C. Vitucci, "Reducing temporal interference in private clouds through real-time containers," in *2019 IEEE International Conference on Edge Computing (EDGE)*, 2019, pp. 124–131.

[26] M. Barletta, M. Cinque, L. De Simone, and R. Della Corte, "Achieving Isolation in Mixed-Criticality Industrial Edge Systems with Real-Time Containers," in *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 231, 2022, pp. 15:1–15:23.

[27] 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.

[28] A. Ramesh, E. Ruppel, A. Rowe, D. Dasari, B. Pourmohseni, F. Smirnov, T. Huang, N. Pereira, M. Giani, A. Hamann, D. Ziegenbein, C. Shelton, and P. Pazzaglia, "A framework for managing edge-cloud distributed embedded systems," in *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Irvine, USA, May 2025.

[29] R. Vaño, I. Lacalle, P. Sowiński, R. S-Julián, and C. E. Palau, "Cloud-native workload orchestration at the edge: A deployment review and future directions," *Sensors*, vol. 23, no. 4, p. 2215, 2023.

[30] A. Ramesh, T. Huang, B. L. Titzer, and A. Rowe, "Empowering webassembly with thin kernel interfaces," in *Proceedings of the Twentieth European Conference on Computer Systems*, 2025, pp. 1–20.

[31] D. Casini, A. Biondi, and G. Buttazzo, "Task splitting and load balancing of dynamic real-time workloads for semi-partitioned edf," *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2168–2181, 2020.

[32] A. Cervin and J. Eker, "Feedback scheduling of control tasks," in *Proceedings of the 39th IEEE Conference on Decision and Control (Cat. No. 00CH37187)*, vol. 5. IEEE, 2000, pp. 4871–4876.

[33] V. Struhár, S. S. Craciunas, M. Ashjaei, M. Behnam, and A. V. Papadopoulos, "React: Enabling real-time container orchestration," in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2021, pp. 1–8.

[34] M. Barletta, M. Cinque, L. D. Simone, and R. D. Corte, "Criticality-aware monitoring and orchestration for containerized industry 4.0 environments," *ACM Transactions on Embedded Computing Systems*, 2024.

[35] F. Lumpp, F. Fummi, H. D. Patel, and N. Bombieri, "Enabling kubernetes orchestration of mixed-criticality software for autonomous mobile robots," *IEEE Transactions on Robotics*, vol. 40, pp. 540–553, 2024.

[36] G. Serra, P. Fara, and D. Casini, "Enhancing the availability of web services in the iot-to-edge-to-cloud compute continuum: A wordpress case study," in *2023 26th Euromicro Conference on Digital System Design (DSD)*. IEEE, 2023, pp. 602–609.