Achieving Isolation in Mixed-Criticality Industrial Edge Systems with Real-Time Containers

Marco Barletta
Università degli Studi di Napoli Federico II, Italy

Marcello Cinque
Università degli Studi di Napoli Federico II, Italy

Luigi De Simone
Università degli Studi di Napoli Federico II, Italy

Raffaele Della Corte
Università degli Studi di Napoli Federico II, Italy

Abstract

Real-time containers are a promising solution to reduce latencies in time-sensitive cloud systems. Recent efforts are emerging to extend their usage in industrial edge systems with mixed-criticality constraints. In these contexts, isolation becomes a major concern: a disturbance (such as timing faults or unexpected overloads) affecting a container must not impact the behavior of other containers deployed on the same hardware. In this paper, we propose a novel architectural solution to achieve isolation in real-time containers, based on real-time co-kernels, hierarchical scheduling, and time-division networking. The architecture has been implemented on Linux patched with the Xenomai co-kernel, extended with a new hierarchical scheduling policy, named `SCHED_DS`, and integrating the RTNet stack. Experimental results are promising in terms of overhead and latency compared to other Linux-based solutions. More importantly, the isolation of containers is guaranteed even in presence of severe co-located disturbances, such as faulty tasks (elapsing more time than declared) or high CPU, network, or I/O stress on the same machine.

2012 ACM Subject Classification Software and its engineering → Real-time systems software

Keywords and phrases Real-time, Mixed-criticality, Containers, Edge computing

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.15

Supplementary Material *Software (ECRTS 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.1.1`

Funding This work has been partially supported by the project COSMIC of UNINA DIETI and by the R&D project "REINForce: REsearch to INspire the Future" (CDS000609), funded by the Italian Ministry for Economic Development (MISE).

1 Introduction

Nowadays, we are witnessing the spread of cloud (including fog and edge) technologies in industrial domain and cyber-physical systems, such as Industrial Internet of Things (IIoT) [31], Industry 4.0, automotive [2], enabling the remote housing of critical software components on the edge of infrastructure, according to the *software defined everything* trend [51]. In these scenarios, edge computing systems can be seen as mixed-criticality systems (MCSs) [5], consolidating real-time tasks with different levels of criticality along non–real-time tasks, all of them on a reduced number of computing nodes in order to reduce the size, weight, power, and cost of hardware. An example is represented by the predictive maintenance of a wind turbine based on artificial intelligence, where the real-time data acquisition/prepossessing is run on the same edge node of a deep neural network [41]. Another example is real-time

control in Industry 4.0 applications, where control activities are designed as tasks with different levels of criticality that perform the control functions, visualization, and interaction with other devices on the same node [47].

Given this scenario, timeliness and isolation become major concerns, since a *deadline miss* could result in a severe failure or hazard. For example, if a critical virtualized ECU on a car is late at handling a brake pedal command, due to a co-located disturbance caused by a non-critical component, this could result in a crash [44]. Similarly, late sensor feedback in a smart factory could bring to the blocking of a production line or even cause injuries.

Recently, container-based virtualization has been gaining the limelight in cloud environments for several reasons: low overhead, simplified migration/orchestration of software appliances, and increased scalability if compared to traditional hypervisor-based approach. Containers are thus considered today a key technology to achieve high workload consolidation and to better exploit hardware resources. For the same reasons, they are starting to attract interest for the realization of real-time systems [7, 48]. However, the adoption of containers in industrial edge systems requires facing compelling challenges due to isolation guarantees needed between different co-located workloads. First, strict resource reservation (such as CPU and network bandwidth) must be guaranteed with acceptable overheads (e.g., low scheduling latency), to grant predictable timing behavior to networked tasks running within critical containers. Second, disturbances must not affect the correct execution of such tasks, other containers (e.g., timing faults in tasks elapsing longer than declared), or the system in general (e.g., unexpected high CPU, I/O, or network load on the same machine).

Several approaches implementing real-time containers have been proposed in literature. Among these, we focus on Linux co-kernel–based solutions due to their wide use in industrial settings, e.g., for robot control [18, 58] and industrial networking support [30], and to their known benefits in terms of latency [13,26] and isolation, as they make Linux fully preemptable in favor of real-time tasks. Nevertheless, co-kernel–based container solutions available to date enforce isolation through active monitoring, which introduces overhead and represents a single point of failure. In addition, existing real-time containers solutions focus on CPU reservation, neglecting network isolation issues, which are fundamental for networked real-time tasks in industrial edge systems.

We aim to overcome existing issues, proposing a novel architectural solution to achieve container isolation in mixed-criticality edge systems. Specifically, our contributions are:

- a *hierarchical scheduling solution*, named `SCHED_DS`, for real-time containers managed by a real-time co-kernel, which results in promising scheduling runtimes (about hundreds of *ns* on our setting) with no need of active monitoring but proactively isolating CPU;
- a *schedulability test*, adapted from early results [16], to verify the feasibility of deployment of a newly created container on existing infrastructure, useful for orchestration purposes;
- a *POSIX-compliant API* to foster the transparent adoption of the solution by practitioners;
- the integration of the solution with a *real-time networking stack* (specifically, RTnet [33]), to achieve network bandwidth guarantees with time-division medium access.

The proposed architecture has been implemented on the Xenomai co-kernel [32], by modifying its `SCHED_QUOTA` to fit a hierarchical deferrable server model (allowing theoretical treatment for schedulability analysis). The resulting policy, `SCHED_DS`, is transparently adopted by real-time tasks running within containers through our API, which also enables containers to use the real-time network. Experimental results show the benefits of the proposed solution in our settings, which exhibit a scheduling runtime in the order of hundreds of nanoseconds, and an overhead lower than 0.1%. More importantly, they highlight the isolation properties of our solution, as tasks run within critical containers are not impacted by faulty tasks running in other containers or by the presence of high CPU, network, or I/O load on the same hardware. Finally, the proposed solution outperforms, in terms of task activation

latency, recent state-of-the-art implementations based on low-latency Linux containers and hierarchical scheduling, in respect to which our solution can be seen as complementary. For instance, compared to PREEMPT_RT, we assessed a relative improvement of at least 30%.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the proposed architecture (see Subsection 3.1), the system model and schedulability analysis (see Subsection 3.2), and the proposed hierarchical scheduling solution (see Subsection 3.3). Section 4 gives implementation details of the architecture on top of the Xenomai co-kernel. Section 5 provides a thorough experimental analysis for proving the applicability of the proposed architecture. Section 6 concludes the paper.

2 Related Work

Currently, mixed-criticality systems development, particularly in edge computing scenarios, brings several stringent requirements like tiny memory footprints (a few MBs or even KBs), fast instantiation times (tens of milliseconds or less), high network throughput (10-40 Gb/s), high consolidation (e.g., being able to run thousands of instances on a single commodity server), and a reduced attack surface for certification purposes. For this reason, lightweight virtualization approaches, like containers, are becoming attractive and starting to be explored in industrial domains.

PREEMPT_RT. Recent studies have started to explore the possibility to use containers to run real-time tasks. For example, [36] presents an empirical study on the problem of minimizing computational and networking latencies for Radio Access Networks through lightweight containers. The study analyzes the performance of Docker containers running on the top of a Linux kernel patched with the well-known `PREEMPT_RT` [17, 45]. This modifies the Linux kernel in order to provide real-time guarantees (e.g., predictability, low latencies) still using a single-kernel approach. The results highlight that using `PREEMPT_RT` improves latencies on Docker containers when compared to a generic kernel. Similarly, authors in [38] propose a sandboxed environment, still based on Docker containers on a `PREEMPT_RT`-based Linux kernel, in the context of an automotive scenario. In [29] a container-based architecture for real-time automation controllers is proposed, using Docker and LXC containers on top of a Linux patched with the `PREEMPT_RT` real-time kernel patch. Experiments have been done using both Docker and LXC containers running on top of a Linux OS patched with the `PREEMPT_RT` real-time kernel patch. Tests emphasize that the use of containers for control applications can meet the requirements of the target systems.

RT-cgroups. Other solutions in literature are based on the control groups (*cgroups*) [40]. Currently, the Linux kernel offers the *real-time group scheduling* (`rt-cgroups`) to host real-time containers [54]. However, the implemented algorithm is somehow unclear [1] and even worse the `PREEMPT_RT` patch has been in conflict with this kind of group scheduling, allowing only a low latency configuration (until v5.15.34) [25] and preventing the fully preemptable one. In [1], the authors propose to modify the `rt-cgroups` implementation by using a real-time deadline-based scheduling policy. The offered solution extends the `SCHED_DEADLINE` policy to schedule Linux control groups. It implements a two-level hierarchical scheduling framework [19] [42] in Linux, allowing user threads to be scheduled with fixed priorities inside the control group scheduled by `SCHED_DEADLINE`.

Co-kernel approach. Besides *cgroups*, alternative approaches include using real-time Linux co-kernels, such as RTAI [20] and Xenomai [32]. The main idea behind the co-kernel approach is to have a small real-time core running at higher priority, which can intercept interrupts and defer them if needed. This choice allows keeping the benefits of the Linux ecosystem (e.g., fully re-using container engines and control group features), while making Linux fully-preemptable in favor of real-time tasks and including the support for real-time networking in the co-kernel, making them a good fit for industrial settings [18, 30, 58]. Generally, RTOS and co-kernel approaches outperform `PREEMPT_RT` in terms of latencies and task-switch times, while Linux is better on average performance, as expected by its GPOS nature [6, 13, 14, 26, 27, 45]. On this line, *Tasci et al.* [50] leveraged both real-time patch and co-kernel approaches to run real-time control applications within Docker containers. In our past work ([8] and [9]) we adopted RTAI to schedule hard real-time tasks within containers, using either fixed priorities in the former or Earliest Deadline First (EDF) [35] in the latter. The idea is to fully inherit the advantages of the co-kernel, in terms of real-time performance and functionalities, while letting tasks within containers keep using the same application programming interface. However, both solutions require active monitoring of tasks run within containers, to tolerate timing failures, which introduces overhead and a single point of failure to the architecture. Further, our solution in [8] requires a global fixed-priority assignment that may limit the adoption of built-in orchestrators, as priorities might be re-assigned if a container needs to be moved from a host to another during its lifespan.

Container orchestration. Container-based virtualization is an attractive choice for time-sensitive edge systems also for the orchestration capabilities provided, such as migration, balancing, and high-availability mechanisms. For example, in [28] and [4] the need for a Kubernetes-compatible [53] edge container orchestrator is highlighted since edge devices are gaining enough power to run containerized services while remaining small and low-powered. Different comparisons and analyses to understand the advantages and disadvantages of the existing orchestrators for edge cloud are made in the two studies. In [24, 49] containers based on rt-cgroups are integrated into Kubernetes in order to orchestrate real-time containers for low latency applications, while in [12] the same containers have been integrated into OpenStack in order to build a framework for NFV for next generation networks. In [59] a *latency-aware* edge computing platform based on Storm is introduced to run computer vision applications with a real-time response time exploiting edge resources. In the context of the LF Edge foundation [52], Xilinx is currently developing a Xen-based lightweight orchestration solution called *RunX* [57]. RunX allows running containers as VMs, either with the provided custom-built Linux-based kernel with a Busybox-based ramdisk, or with a container-specific kernel/ramdisk. Despite some efforts from both industry and academia, there is the need for further analysis in the context of mixed-criticality edge computing systems.

Hypervisor-based solutions. A consolidated trend for time and space partitioning in mixed-criticality systems is to use hypervisor-based solutions, with the aim to completely separate real-time kernels from non–real-time ones. *RT-Xen* [56] is a real-time hypervisor scheduling framework, which extends Xen to support VMs with real-time performance requirements as well as to enable both global and partitioned VM schedulers. *XTRATUM* [39] is a type 1 hypervisor designed for real-time embedded system. XTRATUM can be used to build partitioned systems, and provides both temporal and spatial separation. The Wind River *VxWorks RTOS* [55] features a Virtualization Profile that integrates a real-time embedded, type-1 hypervisor into VxWorks. The hypervisor is able to slow down general-purpose

operating systems to ensure that real-time ones can execute without performance impact. *Jailhouse* [46] is a Linux-based partitioning hypervisor for mixed-critical applications; it enables asymmetric multiprocessing to run both bare-metal applications or guest operating systems (including RTOS). Jailhouse splits physical resources into isolated compartments named cells, with a root cell dedicated to run the Linux kernel and the hypervisor itself, and non-root cells assigned to one guest OS. *Bao* [37] is an open-source lightweight embedded hypervisor for mixed-criticality systems. Bao aims at providing isolation for fault-containment and real-time guarantees, through a static partitioning hypervisor architecture: resources are statically partitioned and assigned at VM instantiation time.

Our proposal advances the state-of-the-art in terms of readiness for industrial settings, since we propose a complementary solution to Linux real-time containers that fully exploits the advantages of co-kernels, while overcoming the current limitations of co-kernel–based container solutions. In particular, we avoid global fixed-priority assignment and active monitoring, using hierarchical scheduling and enabling proactive temporal protection at the co-kernel level, to provide guaranteed CPU bandwidths (i.e., utilization) to different tasks grouped in different containers. Further, the proposed architecture is designed to easily support container orchestration tools in the context of edge computing scenarios, and it integrates a real-time networking stack accessible from containers, an aspect of crucial importance for mixed-criticality edge and neglected by the current literature on real-time containers. Finally, concerning hypervisor-based solutions, our container-based approach allows running multiple isolated Linux systems on a single host with minimal space and performance overhead, avoiding the need of running full VMs (each one replicating the entire OS stack). Other tiny hypervisor-based solutions are based on the concept of *static partitioning* [37, 39, 46]. This makes the environment less prone to changes since this kind of hypervisor includes a one-to-one mapping between virtual CPUs and physical CPUs, and devices are mapped directly into the guest memory areas. Despite the lightness of these approaches, in our context, the use of container-based solutions paves the way to service orchestration and migration, automatic and easy deployment of tasks, which is fundamental with changing environments at the edge level.

3 Proposal

3.1 Architecture

Figure 1 depicts our proposed architecture for achieving isolation in real-time containers. Similarly to earlier proposals [8, 9], we envision the concept of real-time containers (*rt-containers*) hosting real-time tasks (*rt-tasks*) and run on the top of an operating system with real-time scheduling capabilities. The idea is letting rt-containers run along with traditional containers (running non–real-time tasks) on the same machine, with temporal isolation guarantees for rt-tasks. This enables a container-based mixed-criticality environment.

The proposal encompasses an engine layer based on the Docker *container engine* (although other container engines should work too) and includes a *feasibility checker*, which executes a schedulability test to verify (in terms of both CPU and network) if a new rt-container can be created on the computing node, without affecting the containers already hosted. This is particularly useful for orchestration purposes. Moreover, compared to solutions in [1, 54], our proposal is based on *Linux kernel* coupled with a *real-time co-kernel*.

Differently from existing co-kernel-based solutions [8,9], which enforce temporal protection of rt-tasks through a dedicated real-time monitoring component, our proposal leverages a *hierarchical group scheduling* approach, called `SCHED_DS`. The idea is to map the rt-tasks

running within an rt-container in a *task group* at OS co-kernel level, where each group has its own bandwidth (i.e., slices of CPU time) limitation. In this way, the CPU consumption of rt-tasks is limited, by design, over a globally defined period. The rt-tasks with common requirements are pooled in groups; each group receives a share of the global period. Therefore, our proposal does not require active monitoring for temporal protection of rt-tasks, which represents a single point of failure as well as a source of overhead [8,9]. Moreover, differently from the solution in [8], the priority assignment is not globally fixed: within an rt-container, the developer is free to choose task priority levels, as the group CPU bandwidth will be independently guaranteed by the hierarchical scheduler.

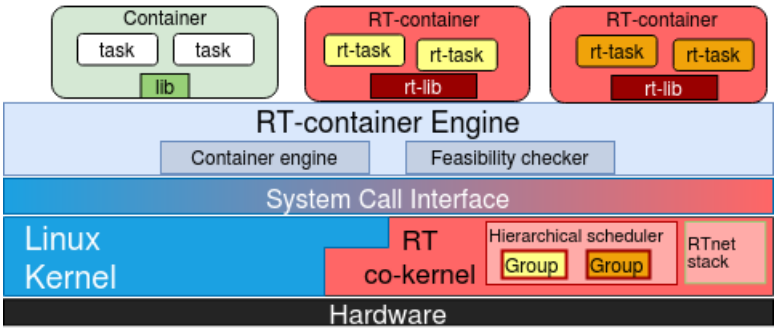


Figure 1 The proposed rt-container architecture.

Our proposal includes also the *rt-lib* layer, which provides a transparent mapping of rt-tasks onto the underlying real-time core. *rt-lib* exposes a POSIX compliant API towards rt-tasks by hiding the information exchanged between the container level and the OS layer, including the one to leverage the co-kernel scheduling policies. It is worth noting that, once a task within a rt-container is mapped to a real-time group (through the rt-lib API) and starts to run, it is scheduled directly by the real-time co-kernel. Hence, it fully preempts Linux and all indirection levels above have no impact on real-time path of the tasks and their scheduling latency.

Finally, we want to remark that our solution integrates also a real-time networking stack for real-time containers, in contrast with past studies. This could be useful in a container-based cloud/edge environment, where several containers, with both non–real-time and real-time requirements, can communicate over the network [11,43]. We rely on the *RTnet stack* [33] usually shipped with real-time co-kernels (such as Xenomai and RTAI) to provide a real-time networking stack. RTnet works with a pluggable real-time medium access control (MAC) policy on standard Ethernet hardware. It offers, by default, a MAC TDMA (Time Division Multiple Access) policy, which consists in allocating a time slot to each physical node of the network for sending/receiving traffic.

Real-time networking guarantees for rt-containers are enforced by assigning a slot to each rt-container, as if they were physical nodes on the network, taking advantage of the RTnet multi-slot configuration. Therefore, the *rt-lib* is in charge of mapping a socket opened by an rt-container with a RTnet socket that has its own slot for transmitting packets. Figure 2 shows an example of TDMA scheduling in the context of rt-containers, in which each slot is identified on the node by an ID, and each node has a default slot for sending traffic (ID 0).

The network relies on a master node providing sync frames, which delimit the TDMA cycles. Since slots cannot overlap, an entity such as an orchestration system should store all the information needed (e.g. slot times and offsets) to assign slots to containers and to

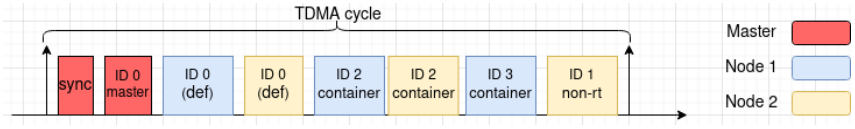


Figure 2 Example of TDMA scheduling.

change dynamically the network configuration. Indeed, RTnet allows adding, removing, or modifying time slots dynamically. For these reasons, the container admission request should contain the network requirements (time slot needed) as well, other than CPU bandwidth. For non–real-time networking instead, well-known methods can be used such as non–real-time buddy tasks, which use the standard Linux networking stack. We highlight that non–real-time traffic is allowed in RTnet in a tunneled way, if necessary.

Figure 3 depicts a typical industrial edge networking scenario in which our proposal can be successfully adopted. The scenario includes a real-time RTnet network interconnecting two edge *RT-Servers* (acting as slaves), a master node, sensors, and actuators. The RT-Servers run several containers and carry out part of the workload with critical timing requirements; however, they can also run non–real-time workloads. The master node represents a high precision clock node supporting RTnet TDMA, which is kept separated from RT-Servers to improve reliability and security. The RT-Servers are connected to both the industrial RTnet and the Internet with two distinct Network Interface Controllers (NICs). In this scenario the rt-containers architecture in Figure 1 can be deployed on RT-Servers in order to manage the scheduling and orchestration of both real-time and non–real-time workloads while ensuring isolation of rt-containers, which may contain critical tasks to retrieve data from sensors and send commands to actuators through the industrial network.

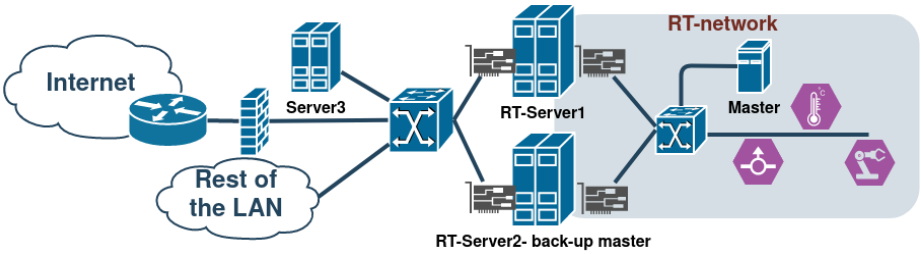


Figure 3 Potential scenario: deployment of rt-containers on real-time edge servers.

3.2 System Model

We assume a system composed of M rt-containers, each of them with an assigned CPU bandwidth (i.e., utilization) U_j^C, a priority level P_j, possibly a network timeslot TS_j and a criticality level CL_j, where $j \in [0, M]$. The criticality represents the severity of consequences in case of failure and it is a notion different from priority, that could be assigned by the schedulability algorithm.

Each rt-container hosts N_j sporadic hard real-time tasks $\tau_i^j \ : \ i = 0...N_j - 1$, characterized by a WCET C_i^j, a minimum inter-arrival separation (or period) T_i^j and a priority level $P_i^j \ : \ i = 0...N_j - 1$. We assume T_i^j to be coincident with the relative deadline D_i^j of the task. Priorities within a container can be freely assigned by the application designer. We assume that tasks can suffer from timing failures, i.e., they can run for a time greater than their declared WCET, because of a wrong timing analysis or a bug inside the code, like an

endless loop. Overall, the system is composed of a set Γ of N tasks, each of them assigned to an rt-container with a given bandwidth. With $\Gamma(j)$ we indicate the subset of tasks within the j-th rt-container. Thus: $\Gamma = \Gamma(0) \cup ... \cup \Gamma(M-1)$ and $\Gamma(k) \cap \Gamma(h) = \emptyset$, where $k \neq h$. The minimum CPU bandwidth of each rt-container is defined as the sum of its task CPU utilization, defined as $U_i^j = C_i^j / T_i^j$. We define the overall system bandwidth as: $U_{TOT} = \sum_{j=0}^{M-1} \sum_{i=0}^{N_j-1} U_i^j$.

We assume non–real-time tasks, internal or external to containers, running with a best-effort policy, receiving the bandwidth unused by the real-time tasks. The model must provide isolation from potential timing failures due to both excessive non–real-time workload on the same machine and faulty tasks, exceeding their declared WCET, run within rt-containers. In order to provide isolation, we rely on bandwidth reservation, assuming a two-level scheduler with a deferrable server[1] at the root level: each rt-container is thus mapped on a server. At the leaf level, the policy is a fixed priority preemptive scheduler, serving tasks within containers. The admission check is performed by the RT-container engine via the *feasibility checker* (implementation details in Section 4), or by an *orchestrator*. We can take advantage of the schedulability analysis presented in [15, 16], where an exact response time analysis improving the already existing analyzes for hierarchical fixed priority schedulers is presented. We recap briefly the test. The response time analysis is described by equations (1) and (2).

$$L_i(w_i) = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j \qquad with\ w_i^0 = C_i + \left(\left\lceil \frac{C_i}{C_S} \right\rceil - 1 \right) (T_S - C_S) \tag{1}$$

$$w_i^{n+1} = L(w_i^n) + \left(\left\lceil \frac{L(w_i^n)}{C_s} \right\rceil - 1 \right) (T_s - C_s) + \sum_{\substack{\forall X \in hp(S) \\ servers}} \left\lceil \frac{max\left(0, w_i^n - \left(\left\lceil \frac{L(w_i^n)}{C_s} \right\rceil - 1 \right) T_s \right) + J_X}{T_X} \right\rceil C_X \tag{2}$$

Equation (1) represents the task load at priority level i and higher, ready to be executed in the *busy period* w_i. $hp(i)$ is the set of tasks that have priorities higher than task τ_i (the task under analysis) within the same server (i.e., a container in our study), and J_j is the release jitter of the task, which is 0 for a bound task and $T_S - C_S$ for unbound tasks, due to the functioning of the server. Equation (2) is a recursive relation for w_i, namely the interval between the time a task is released and its completion time, where $hp(S)$ are the higher priority servers (i.e., containers) for the server S of the task under analysis, J_X is the release jitter of the higher priority server X, which is $T_X - C_X$ for a deferrable server.

The equations model the critical instant for a task scheduled belonging to the server. The recurrence starts with the value in Equation (1) and ends either when $w_i^{n+1} = w_i^n$, in this case $w_i^n + J_i$ indicates the task response time, or when $w_i^{n+1} > D_i - J_i$, thus the task is not schedulable. It is worth nothing that for the schedulability of a task only the budget of higher priority servers is needed. Therefore, even if tasks belonging to other servers (i.e., other rt-containers) are faulty, the schedulability analysis does not need to be modified.

In order to ease the implementation, we hypothesize that the refill timer, i.e., the timer driving the replenishment of the deferrable server budget, is shared between every task group, thus server periods elapse in a lockstep fashion and there is no chance to suffer from back to back hits, which is typical in deferrable servers. This simplifies the third term in Equation (2), which can be rewritten as:

[1] A deferrable server [35] is the simplest of bandwidth-preserving servers, and it provides a good tradeoff between implementation complexity and low response times. It has an execution budget C_s that is replenished each period T_s. A deferrable server preserves its budget until the end of the period.

$$\sum_{\substack{\forall X \in hp(S) \\ servers}} C_X \tag{3}$$

This is possible since having common periods for all servers, a server can run for no more than its runtime budget in each period. This simplification can be demonstrated analytically and we provided the demonstration in an appendix [2]. This also simplifies the server schedulability analysis, since we just have to keep the sum of budgets beneath the period value. Therefore, the server scheduling test becomes: $\sum_{\forall S} C_S \leq T$, where S represents a server, and C_S represents its budget in a period, which is obtained as $C_S = U_S * T$, where U_S is the server bandwidth and T is the common period.

Since the period is fixed, we can use algorithms presented in [16] for optimal schedulability, i.e., the optimal priority assignment or the optimal server capacity allocation. Even if theoretically both algorithms could be used, the optimal priority assignment one requires to assign in advance a bandwidth to containers, which makes it hard to obtain a schedulable set. It is more practical to know container criticality that can be associated with a suitable priority and then compute the needed bandwidth. Therefore, we decide to use the optimal server capacity allocation algorithm described in [16].

Similarly to server scheduling, the test for the network is: $\sum_{\forall N} \sum_{\forall S_{net} \in N} TS_S \leq T_{cycle}$, where N is an RTnet node, S_{net} is a container requiring RTnet, T_{cycle} is the length of the TDMA cycle and TS_S are timeslots not overlapping in time. It is important to notice that despite the network timeslot being independent of the CPU time, it must be taken into account for task dependencies.

3.3 The `SCHED_DS` policy

In this section, we describe the proposed hierarchical scheduling solution, named `SCHED_DS`, for real-time containers managed by a real-time co-kernel. We assume two levels of runqueues: one for the groups and one for tasks belonging to each group. The policy works as follows. It picks the highest priority group with a ready thread, checks its budget and if expired it is moved to an expired list and another group is picked. If the group has budget, the policy picks the highest priority ready thread within the group. The thread is removed from the runqueue, and if the group becomes empty, it is removed from the group runnable queue. The thread starts running and a timer is armed to expire at the end of the budget. Periodically, a refill timer moves all expired groups back to the runqueue and replenishes budgets. When a thread becomes ready it is enqueued in the runnable queue of its group and if it is the only thread in the queue the group is enqueued in the group runnable queue. Algorithm 1 presents the pick function of `SCHED_DS`.

4 Implementation

4.1 A Xenomai-based implementation

For implementing the `SCHED_DS` policy, we chose Xenomai as a co-kernel due to its flexibility, maintainability, and extensibility, and its POSIX-compliant library. Above all, Xenomai recently introduced `SCHED_QUOTA` and `SCHED_TP` policies as partitioned hierarchical scheduling solutions, useful to overcome `SCHED_FIFO` limits. The `SCHED_QUOTA` policy enforces a

[2] Appendix is available at `http://www.fedoa.unina.it/13352/`

Algorithm 1 `sched_ds_pick` pseudocode.
tg = thread group, rq = runqueue, T_{old} is the scheduled out thread, otg = group of T_{old}.

```
1: procedure SCHED_DS_PICK
2:     now ← current time
3:     if tg of T_old is null then
4:         goto pick (6)
5:     subtract (now - start_time) from otg budget
6:     pick highest priority ready tg from rq
7:     if tg is null then
8:         stop limit timer
9:         return null
10:    if tg is empty then
11:        dequeue tg
12:        goto pick (6)
13:    if tg runtime budget is 0 then
14:        enqueue tg in expired queue
15:        goto pick (6)
16:    start_time of tg ← now
17:    pick highest prio ready thread from tg rq
18:    if otg == tg AND limit timer is running
       AND budget_refilled == false then
19:        goto out (22)
20:    budget_refilled ← false
21:    arm limit timer to go off at now+tg budget
22:    decrease active threads in tg
23:    return selected thread
```

limitation on the CPU consumption of threads over a globally defined period, known as the *quota interval*. This is done by pooling threads with common requirements into groups, and giving each group a share of the global period. On the other hand, the `SCHED_TP` policy is a temporal partition scheduler that allows installing a schedule made up of consecutive time windows in a fixed-cyclic style, leaving slots for non–real-time tasks. Even though a prototype based on `SCHED_TP` was built and tested, we preferred leveraging on `SCHED_QUOTA` because of its flexibility (e.g., it allows adding and removing single groups at runtime), which is required in cloud/edge contexts. We remark that Xenomai maintainers have partially already addressed the lack of the PID namespace support, which anyway does not undermine the contribution of this study.

4.2 The `SCHED_QUOTA` Limitations

The `SCHED_QUOTA` policy acts in this way: it picks the highest priority ready thread, and the budget of its group is checked. If it is expired, the thread is moved to an expired list and another thread is picked, otherwise, it starts running with a timer armed to expire at the end of the budget. Periodically, the refill timer moves all expired threads back to the runqueue and replenishes budgets. This implementation revealed three major limitations: (**L1**) the vanilla policy is not actually hierarchical since scheduling is at thread-level; further, the notion of *quota group* only regards the budget evaluation, which is done for each picked thread, causing an O(N) complexity; (**L2**) there is no notion of group priority, since `SCHED_QUOTA` relies on the unique fixed-priority runqueue, thus giving no assurance about isolation between threads of different groups; (**L3**) the scheduler entails a credit mechanism, along with other subtle problems, that prevent from modeling a *quota group* as a deferrable server.

Starting from (L3), we highlight a devious problem: as soon as the refill timer expires, the budgets are refilled, a rescheduling is forced, and the consumed CPU time is subtracted from the budget of the outgoing quota group. This could result in a group starting the new period with a reduced budget if it kept the highest priority before and after the refill. A related problem is that when a group starts executing, the limit timer is armed to expire when the budget is exhausted, without taking care if this moment in time precedes or follows a budget refill. Figure 4 shows an example of the described problem. We consider two groups, *Group 0* (G0) and *Group 1* (G1), where G0 has a capacity equals to 3 over a quota period equal to 5, and a priority higher than G1. A *G1* thread (τ_0 in G_1) starts a request and will be served according to remaining budget for *G1*. Further, at the end of the first quota period (i.e., at t_5), a *Group 0* thread (τ_1 in G_0) starts to execute, with the limit timer armed to expire at

time t_A (i.e., current time, t_5, plus the remaining capacity). After t_5, a refill occurs. Due to the implementation, and assuming that the thread τ_1 in G_0 consumed 1 unit of capacity, the budget of *Group 0* passes from 3 to 2, but the limit timer is left untouched. At t_A, the timer expires and *Group 0* is scheduled out. Thus, in the second quota period, *Group 0* runs for a total of 2 units of time, even if its capacity is 3. We expect the *Group 0* to run for 3 units every 5 to resemble a deferrable server, and the timer armed to expire at t_B.

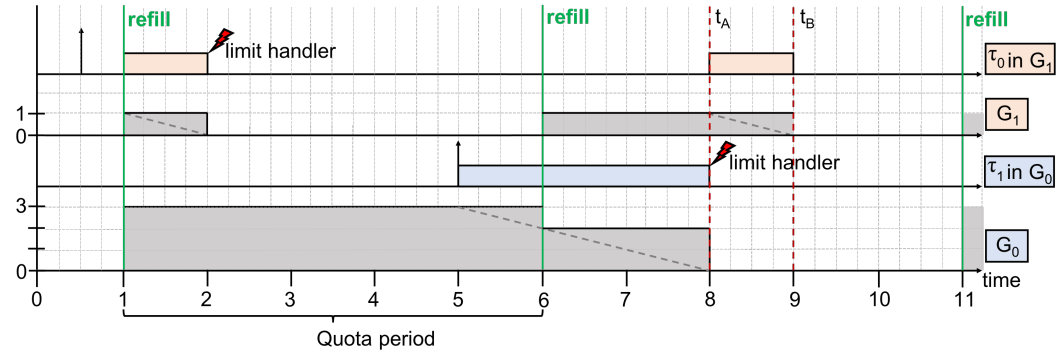


Figure 4 `SCHED_QUOTA` problem. Dashed lines for group budget rensemble the ideal behavior of a deferrable server, while solid lines are the real behavior as implemented in the proposed policy.

Moreover, according to limitation (L2) (see above), two threads belonging to different groups or other scheduling classes share the same range of priority levels within the shared queue. This would mix in time the execution of `SCHED_QUOTA` threads and other threads, nonetheless mixing up threads belonging to different groups with no clue of group priority, with no assurance about temporal isolation of different groups. This behavior of `SCHED_QUOTA` is inadequate to achieve a proper hierarchical scheduler. A solution that uses a user-level priority remapping would result in a restricted priority range, along with another problem (see limitation (L1) above): when a budget expires, the scheduler tries to pick every ready thread of that group in priority order, evaluating exhausted budget for each thread; since the number of threads in a group is not bounded, this operation has an $O(N)$ complexity, where N is the number of consecutive exhausted threads. Similar analysis keeps for refill function that moves expired threads to runqueue.

4.3 Implementation details

We implemented the proposed `SCHED_DS` policy by modifying the existing `SCHED_QUOTA` policy in Xenomai in order to obtain a truly hierarchical scheduler that solves the limitations described above. The Xenomai-based implementation is provided in [3]. The main modifications regard the functions `refill_handler`, which replenishes budgets every quota period, `xnsched_quota_enqueue/requeue/dequeue`, which handle the queue moving, and `xnsched_quota_pick`, which selects the thread to be executed. We created an additional level of scheduling with a runqueue for each group, in order to avoid the single common runqueue, and extending the group structure to encompass a priority level needed to create an order relation between groups to be queued. Furthermore, we created a fixed-priority FIFO runqueue that hosts runnable groups along with a list holding expired groups. We addressed the limitations mentioned in Subsection 4.2 as described in the following:

- **L1.** In the pathological situation formerly considered, the `SCHED_DS` policy will evaluate the budget of the whole group instead of evaluating for every single thread, moving only group structures if needed. These operations are O(1) because the number of groups is upper-bounded by a kernel parameter (by default 32).

- **L2.** `SCHED_DS` policy isolates threads of different groups within the queue, since we do not use anymore the shared FIFO runqueue, allowing usage of the entire priority range without mixing threads of different groups.
- **L3.** `SCHED_DS` policy adds a boolean that is set when the refill timer expires and evaluated during `xnsched_quota_pick` execution, when a true value prevents the reduction o the budget of the formerly running quota group.

Compared to the `xnsched_quota_pick` implementation, we removed some unnecessary checks by improving performance and reducing overhead. Indeed, we allowed empty groups in the group runqueue, dynamically checking for this condition and removing them. This modification has the aim to relieve the following recurring pattern: a group with only one thread is scheduled, next the thread is removed from the group and after that, the group is removed from the queue. When rescheduling is triggered, the currently running thread is requeued as well as its group, causing a heavily ordered insertion, that could be potentially useless if the currently running thread is still the highest priority thread. According to the example shown in Figure 4, the boolean check of *budget_refilled* implies no budget reduction at the scheduling after the refill, the group then starts the new quota period with a full budget. Indeed, the check at line 18 of Algorithm 1 will fail, and the scheduler is obliged to arm once again the limit timer, which will now expire in t_B, in fulfillment of our requirement about the deferrable server behavior.

4.4 Feasibility Checker and `rt-lib`

We implemented a *feasibility checker* that parses the input and computes the tests introduced in Subsection 3.2. This consists of 600 LOC of Python code, which calls two C programs that use Xenomai services to bring up and tear down groups.

The *rt-lib* layer is implemented as a header file that relies on the *–wrap* linker flag to override the POSIX wrapper functions defined by Xenomai. Overridden functions belong to two distinct groups: one for hiding the complexity of thread management and one for the RTnet socket management. The first group of functions uses the original data structures to find and operate on the related extended ones. This exposes to the user a standard POSIX API that under the hood replaces real-time scheduling policies with `SCHED_DS`. On the other hand, networking functions ensure that a newly created socket calls an ioctl to bind to the allotted RTnet timeslot.

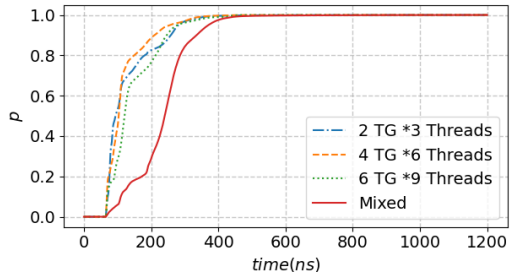
5 Experimental Results

In this section, we report the experimentation we performed on the implemented architecture. The experimentation aims to: *i)* measure scheduling policy runtimes; *ii)* estimate the overall throughput overhead as seen by the tasks to understand if the proposed algorithm can be practically used with negligible risk and to derive a suitable server period; *iii)* assess temporal isolation under several disturbances conditions, evaluating the solution in terms of number of failures, specifically, the *deadline misses* of the real-time tasks under non–real-time stress and faulty real-time container; *iv)* estimate the RTnet error deviation for sending packets through measurements achieved on a real network deployment; *v)* estimate the activation latency of the scheduler, comparing results with state-of-art solutions for real-time containers. In Table 1, there is a summary of experimental parameters used in the following tests.

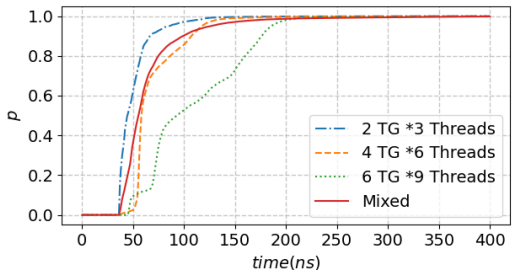
The host system is equipped with an Intel Core I5-6500, 16 GB DDR4 RAM, Samsung 970EVO SSD, running Linux Mint 20.1, kernel v5.4.77 patched with Xenomai v3.1, and Docker v20.10.7. To reduce hardware sources of non-determinism, we disabled power optimizations, frequency scaling, Intel SpeedStep, TurboBoost, and C-States.

Table 1 Values and rationale behind parameters used in the experimentation.

Parameter	Ref. Sec.	Value	Rationale
Groups/Threads	5.1	2*3 up to 6*9	Understand how execution times vary with numbers of threads spread along an order of magnitude.
Runs	5.1	5	Sufficient to average out transient behaviors.
Repetitions	5.2	50	Computed by the sample size formula, to have 90% confidence.
Quota period	5.2	1 ms to 10 ms	Reasonable for the hardware used and DS runtimes obtained.
Target quota	5.2	40%	Sufficient runtime to show overhead effects.
Yielding period	5.2	0.1, 1, 10 ms	Different orders of magnitude, in the range below the quota period.
Duration	5.2	10 s	Enough to verify the overhead, on the base of the period.
U_{TOT} limit	5.3	85%	From [9], utilization > 90 % brings system to instability.
WCET buffer	5.3	30 μs	Max activation latency measured through the Xenomai official guide plus hardware unpredictability measured.
Additional band	5.3	1 %	Compensate scheduling overhead under stress.
Repetitions	5.3	30	Tradeoff between statistical significance and time needed.
Containers/tasks	5.3	2*2 up to 2*6	Reasonable numbers to find schedulable tasksets with small periods.
Task periods	5.3	1 ms up to 2 s	Reasonable real-time periods for a general purpose hardware.
Duration	5.3	60 s	Reasonable with regard to task periods.
TDMA cycle	5.4	6 ms	Reasonable time based on the roundtrip time measured.
Repetitions	5.5	60	Sample size formula for 90% confidence.
Sampling period	5.5	100 μs	Default period of the test.
Duration	5.5	60 s	Enough to experiment outliers due to kernels. Several orders of magnitude above the period.
Task WCET	5.6	1.8 ms	Avoid budget limitation of the policies to simplify analysis.
Task Period	5.6	10 ms	Reasonable for the WCET to have as many sampling as possible.



(a) CDF of runtimes of `xnsched_pick_next`.



(b) CDF of runtimes of `refill_handler`.

Figure 5 CDF runtimes of `SCHED_DS` (TG = Thread Group).

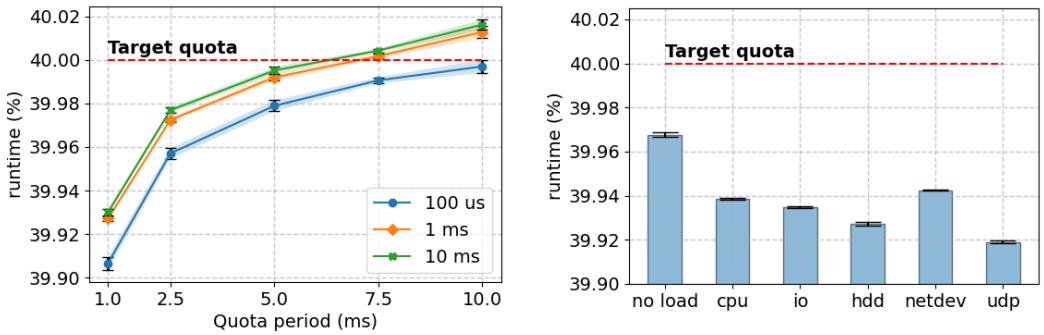
5.1 `SCHED_DS` Runtimes

We estimate the runtime of the main functions of the scheduler for a first quality evaluation. Both the `xnsched_pick_next` and the `refill_handler` runtimes were plotted, in order to understand which quota period could have been suitable for the system. The `xnsched_pick_next` checks the current thread status and, if needed, calls a requeue and a sched_pick for each scheduling policy in priority order, until a thread is found. The `refill_handler` was described previously in Section 3.3 and represents a periodic fixed overhead. Therefore, we can have an upper bound of enqueue, dequeue, and requeue functions, and an upper bound of the `xnsched_ds_pick` function, i.e., the one described in Algorithm 1. Runtimes are taken with a couple of clock monotonic timestamps, at the beginning and end of the functions under exam, under different workloads: 2 thread groups with 3 threads

each; 4 groups with 6 threads each; 6 groups with 9 threads; a mixed workload created as described in Section 5.3, with a random group number between 2 and 6, and a random thread number between 2 and 7 for each group. For the first three loads, two tasks for the group are permanently running in order to exhaust the budget at every period. The selected number of groups/threads have the only aim to understand how execution times vary with an increasing number of threads during nominal execution. Results in terms of CDF (averaged over 5 runs) are shown in Figures 5a and 5b. The workloads run on a single CPU and thread/group numbers have the only purpose to exercise the system with an increasing stress, to have an estimation of the overhead. A replenishment takes in the worst case 630 ns (Figure 5b), with a much lower average time, while the pick function has a worst case of 1,343 ns (Figure 5a), but it generally presents much lower runtimes: around 450 ns the CDF value is almost one.

5.2 `SCHED_DS` Overhead

In order to estimate the overhead induced by the proposed scheduler, we adapted the test for `SCHED_QUOTA` of the Xenomai suite. The test is an application that creates FIFO threads, which increment a counter and then yield in favor of another thread after a specified amount of time, called *yielding period* hereinafter. The obtained counter will be then used as a reference value. Next, a number of `SCHED_DS` threads, in the same amount as the FIFO threads, are created. The `SCHED_DS` threads belong to a group created with a determined quota, and have the same body as the FIFO ones. At the end of the application (after a timeout of 10 seconds), the `SCHED_DS` counters are compared to the FIFO reference. In this way, the effective percentage of the runtime of `SCHED_DS` threads can be computed. It should be noted that the results depend on the length of the quota period. The smaller period is used, the more overhead is induced and the smaller effective runtime percentage is computed at the end of the test. The `SCHED_DS` threads within the test are run with $1ms$, $2.5ms$, $5ms$, $7.5ms$, $10ms$ *quota periods*, assigning a target quota equal to 40%. Moreover, we experiment 100 μs, $1ms$, and $10ms$ as yielding periods for both `SCHED_FIFO` and `SCHED_DS` threads. For each test, we run 12 threads (FIFO threads to compute the reference values and `SCHED_DS` threads to compute the overhead), with a random activation phase with regard to the server period to reduce anomalies that depend the first and last period execution. Each test is repeated 50 times for statistical significance purposes. Afterward, the average of the runtimes percentage, fixing the same quota period and yielding period, are computed and plotted with 90% of confidence.



(a) Varying quota and yielding periods. **(b)** Under stress scenarios.

Figure 6 Runtime percentages of running `SCHED_DS` threads.

Figure 6a shows the obtained runtime results by varying both quota and yielding periods, along with the quota target percentage, i.e., 40% indicated by the dotted red line. It can be noted that when the quota period is $1ms$ the induced overhead is approximately $0.05 - 0.1\%$, while with a quota period of 2.5 ms the estimated overhead is lower, i.e., between 0.02 and 0.05%. Differently, with a quota period equal to 10 ms the results show a runtime percentage greater than 40%. This is probably due to both the thread phasing with regard to the beginning of the first period and the deferrable server behavior: indeed, threads can benefit from the entire runtime budget over a reduced quota period because of late arrival. For lower yielding periods, the overhead is greater, as expected. Furthermore, we highlight that the runtime percentage difference between the yielding period of $100\,\mu s$ compared to both $1ms$ and $10ms$ is clear since at $1ms$ and $10ms$ the scheduling policy preemption dominates the number of yielding preemptions. According to the results obtained for the overhead, we set the quota period for the subsequent test equal to $2.5ms$ since it represents a good choice for the schedulability compared to the other periods tested.

We compute also the overhead of `SCHED_DS` under various stress conditions. The disturbances are generated via the `stress-ng` tool [22]. We impose hard pinning for the CPU core that executes the stress load by using the `-taskset` flag. We test different stress load scenarios, i.e., *no load* (tests executed with no load), *cpu* (a heavy arithmetic computation to fully use one CPU core), *io* (an IO load using one worker spinning on `sync()` command to force writes data buffered in memory out to disk), *hdd* (a disk load that starts one worker generating various operations towards the disk), *netdev* (a network load where a worker exercises various netdevice ioctl commands for all available network interface controllers) and *udp* (a network load starting a pair of client and server workers that transmits data using UDP on localhost). Regarding stress tests, we still perform 50 repetitions for statistical significance purposes. The quota period is set equal to $2.5ms$ and the yielding period equal to $1ms$. Figure 6b shows the obtained results, which highlight that scheduling efficiency slightly decreases under stress conditions. The results under stress conditions compared to the *no load* ones are approximately $0.03 - 0.04\%$ lower, with *udp* and *hdd* as worst loads because of the high volume of interrupts generated.

5.3 Failure Isolation Test

In this section, we perform tests for the proposed scheduling policy for revealing potential temporal isolation issues. The tests include, first, the scheduler correctness (whether it behaves as expected). Further, we want to obtain real performances in the field of `SCHED_DS` under synthetic workloads, against several stress conditions. We demonstrate that the provided proactive temporal isolation successfully prevents failures (deadline misses) propagation within the system, crossing criticality levels. Specifically, the test assumes two criticality levels, thus including two real-time containers: one container with a high priority and the other one with a low priority.

The total bandwidth U_{TOT} is spread over the two containers using the *RandFixedSum* algorithm [21] (we used the Python implementation [23]), such that $U_{TOT} = U_{LOW} + U_{HI}$, where U_{LOW} and U_{HI} are the bandwidths of low-priority and high-priority rt-containers, respectively. For both containers a group of tasks is created through the *RandFixedSum* algorithm as well, such that the tasks bandwidth for each container is $U_j = \sum_{i=0}^{N_j - 1} U_i^j$, where $j \in \{LOW, HI\}$ and N_j is the number of tasks of the container j. The periods used as input for the algorithm range in likely times for real-time tasks (ms up to s). Priorities are assigned to the two task groups according to the order of creation, while the algorithm in [16] and Equations 1, 2, 3 are used to determine the minimum bandwidth needed for the containers to have a schedulable task set.

Tasks within a task group are ordered following Rate Monotonic (RM) rules [34]. If the sum of required bandwidth needed to have a schedulable task set is greater than 85%, the current task set is discarded, and a new one is generated. The required bandwidth of each container is increased by 1% to compensate for the scheduling overhead. In the algorithm, a little margin of 30 μs is added to WCET of each task to consider maximum Xenomai scheduling latency plus a margin due to hardware unpredictability. Moreover, for each task, there is a lower threshold WCET of 30 μs that allows avoiding borderline situations, e.g., workloads too short to be properly emulated on x86 architectures. For the same reason, each group must have a total utilization greater than 0.1% quota.

Once created a task set with the given constraints and computed the needed bandwidth for each container, quota groups are created, and synthetic load is created through the *rt-app* [10], exploiting the *CPU property* to pin tasks on a CPU core, the *run property* to create a workload of a determined time, the scheduler policy and absolute timer properties to select the `SCHED_DS` policy and generate a periodic timer. The *rt-app* has been modified and recompiled for our purposes, linking real-time libraries. Each thread logs some information for each iteration in a pre-allocated memory area. The slack time, defined as next activation time minus current finish time, can be used to compute deadline miss occurrences; indeed, a negative slack means that the task has finished after the start of the next period.

Tests are run with varying total bandwidth U_{TOT}: 50, 55, 60, 65, 70 percent of the CPU usage. In each container, there is a random number of threads, chosen between 2 and 6 (excluded). 30 repetitions are done for each combination of total bandwidth and stress condition. Stress conditions considered in this test include a scenario, namely *low-hi*, in which the low-priority container behaves as declared, while the high-priority container is faulty, i.e., it executes tasks with a runtime of 1.8 times than declared. This scenario aims to detect if quota groups are correctly isolated in time: the high-priority container is faulty since has an actual load almost twice the declared. The desired behavior is that the lower priority container will be isolated from the faulty container. Thus, we expect deadline misses in the high-priority container but no deadline miss in the low-priority container, which could be potentially more critical than the high-priority one. The other tested stress scenarios are the same as the ones described in Section 5.2. These scenarios aim to understand the behavior of the system under various non–real-time workloads. We expect to have no deadline miss at all. Each experiment lasts 60 seconds. Task periods are sampled with a *loguniform* distribution, with a granularity of 1 ms, with a minimum period of 1 ms, and a maximum period of 2 s. As expected, no failures came out from experiments under non–real-time stress-ng load. It is worth mentioning that, although the server period is $2,500$ μs, the schedulability test can guarantee shorter period tasks as well the time to execute, and despite the needed bandwidth is quite high, several tasks with a period of 1 ms have been scheduled in the low-priority container. We tested the 1 ms quota period and their schedulability is a lot easier. The results meet our expectations under the *low-hi* scenario as well. The high-priority container presents a high number of failures (in terms of deadline misses), while the low-priority container achieves the correct execution.

Figure 7 shows the boxplots of the deadline misses in the *low-hi* scenario. Each point represents the sum of the number of deadlines misses from all tasks in a container, while on the x-axis there is the total bandwidth used to generate the task set. It is worth noting that some tests run with no deadline miss: this is likely due to the extra bandwidth, or the pessimistic behavior of the schedulability test. The test seems to have quite encouraging results, as `SCHED_DS` can resist several high-stress conditions and temporal isolation is assured.

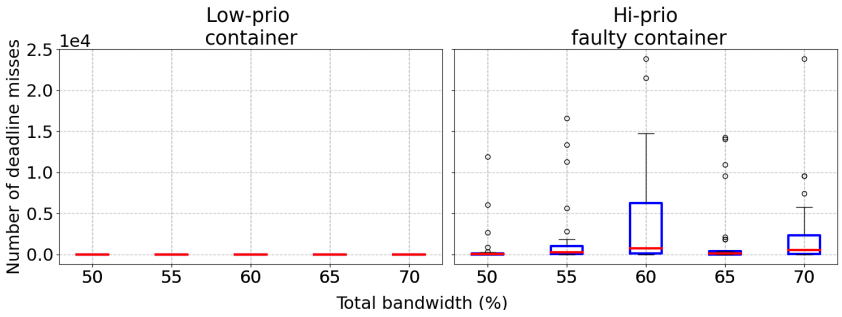


Figure 7 Boxplots of deadline misses for the *low-hi* scenario across different usage percentage of total bandwidth U_{TOT}.

5.4 RTnet

In this section, we perform tests to show the real-time capabilities of the proposed RTnet wrapper used within containers and in presence of non-real-time traffic disturbances. The aim is to highlight that the RTnet stack is characterized by a low average and standard deviation of the sending delay (defined as the difference between the expected sending time and the actual one) even under non–real-time tunneled traffic. Tests were run on two HP z230 workstations, equipped with an Intel Core i7-4790, 16 GB RAM, and an i217-LM network controller with `rt_e1000e` driver. NICs were linked to a 100 MBit/s switch through cat6 Ethernet cables. The machines hosted Ubuntu Server 21.04, with Linux kernel v5.4.77 patched with Xenomai v3.1.1.

The TDMA cycle was set to 6 ms, with slots of 1 ms. The transmission delay of the *sync frame* from the setup was estimated at around 112 μs (computed as described in [33]). We start a couple of containers on the slave node, i.e., the *RT-Server1* in Figure 3, in which they spawn one periodic thread each, with coprime periods to force coincidence for network transmission within the same TDMA cycle. These tasks send a UDP packet towards the master, i.e., the *Master* node in Figure 3, for each period. Further, at the host level on the *RT-Server1*, we used the *socat* tool to send, continuously, a file in UDP broadcast, in order to generate non–real-time traffic. These UDP packets are automatically divided into fragments by the RTnet stack to fit the slot reserved for non–real-time traffic. We recorded the actual sending timestamp and scheduled sending timestamp for the *sync frames*, and sending timestamps for slave packets. The slave properly used the estimated transmission delay to adjust the slot starting times [33]: the sending offset of the outgoing packets with regard to the arrival time of *sync frames* were not integer multiples of slot durations, and 112 μs was the offset that minimized the sum of squared errors, defined as the difference of time to the nearest multiple of slot duration from sync frame. The mean, minimum, maximum, and standard deviation of the difference between scheduled sending timestamp and actual sending timestamp for *sync frames* were respectively 385.8, 257, 1,499, and 103.2 ns. On the other hand, keeping into account the correction of 112 μs, the slave errors for sending frames were characterized by an average of 211.8 ns and a standard deviation of 1,006.7 ns, showing good predictability with regard to a non–real-time RTnet stack.

5.5 Task Activation Latencies

In this section, we aim to estimate the tasks activation latencies provided by our solution and use them to compare the proposal with the following alternative solutions for real-time containers based on hierarchical scheduling: *i)* Linux low-latency vanilla rt-cgroups (LL

hereinafter) and *ii)* rt-cgroups patched as described in [1] (HCBS hereinafter). Kernels in LL and HCBS are in low-latency configuration due to reasons explained in Section 2. Despite we set our solution as complementary (i.e., designed for usage in different use cases), we compare our implementation with LL and HCBS since they are the only ones to date based on group scheduling. The vanilla rt-cgroups allow dividing CPU time, specifying how much time can be spent running in a given period. The *runtime* is allocated to each real-time group (*rt-group*), and other groups will not be allowed to use it. The time not allocated to a *rt-group* is used to run `SCHED_OTHER` tasks. Currently, this feature still lacks an EDF scheduler to use non-uniform periods. The low-latency kernel version is the same as Xenomai (v5.4.77), while HCBS supports an older kernel version (i.e., v5.2.8). Kernel settings are almost the same, the only differences between the configurations are constrained by patches. The latency test provided by Xenomai test suite is modified to run with `SCHED_FIFO` for Linux, making a lift and shift of data structures used, creating an independent program. We further adapted this latency test to support `SCHED_DS` threads.

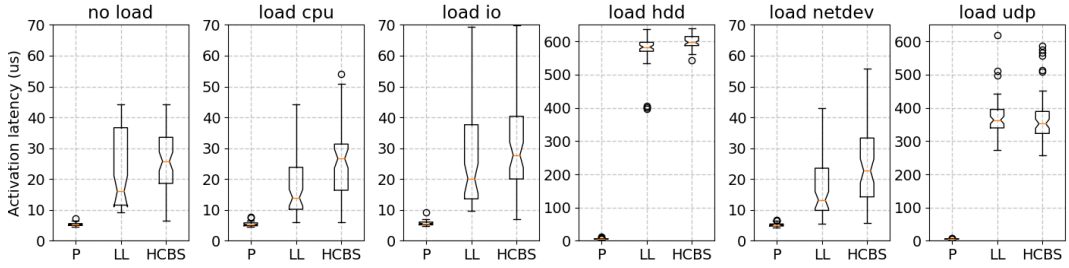


Figure 8 Boxplots of task activation latencies. In *no load* scenario, for HCBS, we removed two outliers at 114 and 118 μs for readability. P is the proposed solution.

We run the latency test for 60 seconds for each targeted solutions (i.e., the proposed one, *LL*, and *HCBS*) according to stress load scenarios described in Section 5.2. For each test, we saved the maximum latency and the overruns. We repeated each test 60 times for statistical significance purposes. The task period is left as the default value used in the original Xenomai latency test, i.e., 100 μs.

Figure 8 shows boxplots of the latency experienced by the analyzed solutions under different disturbances, while Table 2 provides the mean and standard deviation of the obtained maximum latencies. It can be noted that *No stress*, *cpu*, *io*, and *netdev* loads provide similar results. The proposal seems to be robust against all the disturbances, giving persistently lower latencies ($\sim$ 10 μs). Under *udp* and *hdd* loads, both the LL and HCBS provide higher latencies, with a mean of the maximum latencies about 60 times worse than the proposed solution and a non-negligible standard deviation. We expected this behavior since both *udp* and *hdd* loads raise a high volume of interrupts, which heavily affect kernel performance despite the low-latency configuration of the Linux kernel. Our co-kernel–based implementation is not affected significantly by such high interrupts load, making it a good fit for critical high-frequency tasks for industrial control. Finally, we tried to switch the SSD disk with HDD one. We noticed a non-negligible difference under the *hdd* load, concluding that latencies are mostly caused by drivers, which probably present non-preemptible sections. For the sake of brevity, we do not report this experiment. The difference between *LL* and *HCBS* is not significant, thus the main reason behind high latencies is not the hierarchical scheduler itself, but the preemption model of Linux. On the other hand, Xenomai makes Linux fully preemptible, thus Xenomai outperforms other kernels.

Table 2 Maximum task activation latency.

	Proposed		LL		HCBS	
Load	Mean [μs]	Std	Mean [μs]	Std	Mean [μs]	Std
no load	5.332	0.509	22.334	12.075	28.396	18.464
cpu	5.367	0.665	18.477	10.743	25.795	11.813
io	5.695	0.726	25.016	13.083	30.771	14.328
hdd	5.983	1.061	572.969	50.047	599.308	19.923
netdev	5.231	0.600	17.718	10.643	25.082	14.313
udp	5.879	0.253	369.252	58.109	368.706	73.646

5.6 Comparison with PREEMPT_RT

Since the low-latency configuration is not able to always guarantee timeliness, in this section we directly compare Xenomai against PREEMPT_RT. The PREEMPT_RT patch has recently solved the conflict with rt-cgroups, from kernel v5.15.34. However, we highlight that without the HCBS patch, the rt-cgroup scheduler does not fit any theoretical model for schedulability. We ran *rt-app* in a container for 60 seconds, sampling the worst task activation latency and slack time along the minute for each of the stress loads considered in the previous test. The slack time is defined as $d - a - C$, where d is the task deadline, a is the arrival time, and C its WCET. We repeated each test 60 times for statistical significance purposes. The *rt-app* generates a single thread with a period of 10 ms and a runtime of 1.8 ms, to avoid budget limitation of the scheduling policies. Obtained results are shown in Figure 9.

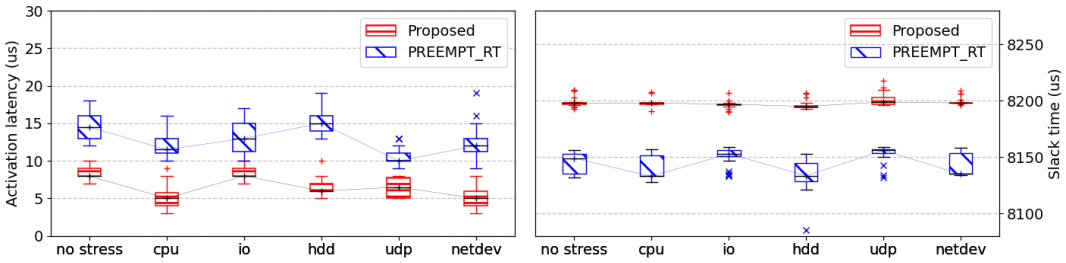


Figure 9 Boxplots of both worst task activation latencies (lower is better) and slack times (higher is better) under different stress loads.

Activation latencies, although comparable, are always lower for Xenomai under each stress, with an average improvement of at least 30% in the case of the *udp* load, and above 50% for the *hdd* load, which is still particularly troublesome for Linux. The lower latencies with regard the low-latency configuration confirm our statement that high latencies in the previous experiments are due to non-preemptible sections. Even the slack time for the proposed system is closer to the expected (i.e., 8200μs). Moreover, the slack time of our solution presents a lower standard deviation with regard to PREEMPT_RT, probably due to a lower predictability and higher the complexity of Linux, along its difficulty to handle workloads. Once again, under *hdd* load, Linux presents the worst outlier.

6 Conclusion

In this work, we introduced a novel architecture for real-time containers by leveraging a hierarchical deferrable server scheduler in a real-time co-kernel. We integrated the solution with a real-time networking stack (i.e., RTnet) for communication purposes, and provided the schedulability test and a user-level APIs to deploy the containers. The proposed architecture has been implemented extending the Xenomai co-kernel; the source code has

been made publicly available. Extensive experimental tests have been performed, showing that the proposed solution is promising in terms of latency, overhead, and, most importantly, isolation against disturbances. Specifically, a low-priority container can resist against severe misbehavior of a high-priority container, with no impact in terms of timing failures (i.e., no deadline miss). Further, comparing the proposed architecture with solutions in the state of the art, we obtain benefits in terms of task activation latencies. Future works aim at supporting rt-containers in the context of container orchestration platforms (e.g., Kubernetes), in order to create a fully-automated mixed-criticality industrial edge/cloud solution.

References

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

2 Amazon Inc. Getting started with cloud-native automotive software development. URL: `https://catalog.us-east-1.prod.workshops.aws/v2/workshops/12f31c93-5926-4477-996c-d47f4524905d/en-US`. Accessed 17th June 2022.

3 Marco Barletta, Marcello Cinque, Luigi De Simone, and Raffaele Della Corte. Xeno-containers, GitLab repo. `https://dessert.unina.it:8088/marcobarlo/xeno-containers`.

4 Sebastian Böhm and Guido Wirtz. Profiling lightweight container platforms: Microk8s and k3s in comparison to kubernetes. In *Proc. ZEUS*, pages 65–73, 2021.

5 A. Burns and R. I. Davis. Mixed Criticality Systems – a review. *Tech Rep of the University of York*, 2018. URL: `https://www-users.cs.york.ac.uk/burns/review.pdf`.

6 Felipe Cerqueira and Björn Brandenburg. A comparison of scheduling latency in linux, preempt-rt, and litmus rt. In *9th Annual workshop on operating systems platforms for embedded real-time applications*, pages 19–29. SYSGO AG, 2013.

7 Marcello Cinque, Domenico Cotroneo, Luigi De Simone, and Stefano Rosiello. Virtualizing mixed-criticality systems: A survey on industrial trends and issues. *Elsevier Future Generation Computer Systems*, 2021.

8 Marcello Cinque, Raffaele Della Corte, Antonio Eliso, and Antonio Pecchia. Rt-cases: Container-based virtualization for temporally separated mixed-criticality task sets. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

9 Marcello Cinque, Raffaele Della Corte, and Roberto Ruggiero. Preventing timing failures in mixed-criticality clouds with dynamic real-time containers. In *2021 17th European Dependable Computing Conference (EDCC)*, pages 17–24. IEEE, 2021.

10 Multiple contributors. Scheduler tools /rt-app. `https://github.com/scheduler-tools/rt-app`. Accessed 17th June 2022.

11 Breno Costa, Joao Bachiega Jr, Leonardo Rebouças de Carvalho, and Aleteia PF Araujo. Orchestration in fog computing: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 55(2):1–34, 2022.

12 Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Riccardo Mancini, and Carlo Vitucci. Strong temporal isolation among containers in openstack for nfv services. *IEEE Transactions on Cloud Computing*, pages 1–1, 2021.

13 N. T. Dantam et al. The ach library: A new framework for real-time communication. *IEEE Robotics Automation Magazine*, 22(1):76–85, 2015.

14 Neil T Dantam, Daniel M Lofaro, Ayonga Hereid, Paul Y Oh, Aaron D Ames, and Mike Stilman. The ach library: A new framework for real-time communication. *IEEE Robotics & Automation Magazine*, 22(1):76–85, 2015.

15 R.I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 10 pp.–398, 2005. `doi:10.1109/RTSS.2005.25`.

16 Rob Davis and Alan Burns. An investigation into server parameter selection for hierarchical fixed priority pre-emptive systems. In *16th International Conference on Real-Time and Network Systems (RTNS 2008)*, 2008.

17 Daniel Bristot de Oliveira, Daniel Casini, Rômulo Silva de Oliveira, and Tommaso Cucinotta. Demystifying the real-time linux scheduling latency. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

18 Raimarius Delgado, Bum-Jae You, and Byoung Wook Choi. Real-time control architecture based on xenomai using ros packages for a service robot. *Journal of Systems and Software*, 151:8–19, 2019.

19 Zhong Deng and JW-S Liu. Scheduling real-time applications in an open environment. In *Proceedings Real-Time Systems Symposium*, pages 308–319. IEEE, 1997.

20 Dipartimento di Ingegneria Aerospaziale del Politecnico di Milano (DIAPM). RTAI - the RealTime Application Interface for Linux. `https://www.rtai.org/`. Accessed 17th June 2022.

21 P. Emberson, R. Stafford, and R.I. Davis. Techniques for the synthesis of multiprocessor tasksets. *WATERS'10*, January 2010.

22 Colin Ian King et al. Stress-ng GitHub repository. `https://github.com/ColinIanKing/stress-ng`. Accessed 17th June 2022.

23 Cucinotta et al. Taskset generator. `https://gitlab.retis.santannapisa.it/t.cucinotta/rtsim/-/blob/master/src/taskset_generator/taskgen.py`. Accessed 17th June 2022.

24 Stefano Fiori, Luca Abeni, and Tommaso Cucinotta. Rt-kubernetes—containerized real-time cloud computing. In *37th ACM/SIGAPP Symposium on Applied Computing (SAC '22)*, 2022.

25 The Linux Foundation. Know Limitations of PREEMPT_RT patch. `https://wiki.linuxfoundation.org/realtime/documentation/known_limitations`. Accessed 17th June 2022.

26 C. Garre, D. Mundo, M. Gubitosa, and A. Toso. Real-time and real-fast performance of general-purpose and real-time operating systems in multithreaded physical simulation of complex mechanical systems. *Mathematical Problems in Engineering*, 2014, 2014.

27 Carlos Garre, Domenico Mundo, Marco Gubitosa, and Alessandro Toso. Performance comparison of real-time and general-purpose operating systems in parallel physical simulation with high computational cost. Technical report, SAE Technical Paper, 2014.

28 Tom Goethals, Filip De Turck, and Bruno Volckaert. Fledge: Kubernetes compatible container orchestration on low-resource edge devices. In *International Conference on Internet of Vehicles*, pages 174–189. Springer, 2019.

29 T. Goldschmidt, S. Hauck-Stattelmann, S. Malakuti, and S. Grüner. Container-based architecture for flexible industrial control applications. *Journal of Systems Architecture*, 84:28–36, 2018.

30 Thakor Bhishmapalsinh Jitendrasinh and Shripad Deshpande. Implementation of can bus protocol on xenomai rtos on arm platform for industrial automation. In *2016 International Conference on Computation of Power, Energy Information and Commuincation (ICCPEIC)*, pages 165–169. IEEE, 2016.

31 Kuljeet Kaur, Sahil Garg, Gagangeet Singh Aujla, Neeraj Kumar, Joel JPC Rodrigues, and Mohsen Guizani. Edge computing in the industrial internet of things environment: Software-defined-networks-based edge-cloud interplay. *IEEE communications magazine*, 56(2):44–51, 2018.

32 Jan Kiszka. Xenomai Homepage. URL: `https://source.denx.de/Xenomai/xenomai/-/wikis/home`. Accessed 17th June 2022.

33 Jan Kiszka and Bernardo Wagner. Rtnet-a flexible hard real-time networking framework. In *2005 IEEE Conference on Emerging Technologies and Factory Automation*, volume 1, pages 8–pp. IEEE, 2005.

34 John Lehoczky, Lui Sha, and Yuqin Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *RTSS*, volume 89, pages 166–171, 1989.

35 Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, Upper Saddle River, NJ, 2000.

36 C.-N. Mao et al. Minimizing latency of real-time container cloud for software radio access networks. In *IEEE 7th International Conference on Cloud Computing Technology and Science*, pages 611–616, 2015.

37 José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems. In *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

38 Philip Masek, Magnus Thulin, Hugo Sica de Andrade, Christian Berger, and Ola Benderius. Systematic evaluation of sandboxed software deployment for real-time software on the example of a self-driving heavy vehicle. *CoRR*, abs/1608.06759, 2016. `arXiv:1608.06759`.

39 Miguel Masmano, Ismael Ripoll, Alfons Crespo, and J Metge. Xtratum: a hypervisor for safety critical embedded systems. In *11th Real-Time Linux Workshop*, pages 263–272. Citeseer, 2009.

40 Paul Menage. Cgroups. `https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`. Accessed 17th June 2022.

41 Hassan Ghasemzadeh Mohammadi, Rahil Arshad, Sneha Rautmare, Suraj Manjunatha, Maurice Kuschel, Felix Paul Jentzsch, Marco Platzner, Alexander Boschmann, and Dirk Schollbach. DeepWind: An Accurate Wind Turbine Condition Monitoring Framework via Deep Learning on Embedded Platforms. In *Proc. ETFA*, volume 1, pages 1431–1434, 2020.

42 M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos. Mixed-Criticality Real-Time Scheduling for Multicore Systems. *10th IEEE International Conference on Computer and Information Technology, Bradford, pp. 1864-1871*, 2010.

43 Harald Mueller, Spyridon V. Gogouvitis, Andreas Seitz, and Bernd Bruegge. Seamless computing for industrial systems spanning cloud and edge. In *2017 International Conference on High Performance Computing Simulation (HPCS)*, pages 209–216, 2017.

44 Thorsten Piper, Stefan Winter, Oliver Schwahn, Suman Bidarahalli, and Neeraj Suri. Mitigating timing error propagation in mixed-criticality automotive systems. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, pages 102–109. IEEE, 2015.

45 Federico Reghenzani, Giuseppe Massari, and William Fornaciari. The real-time linux kernel: A survey on preempt_rt. *ACM Computing Surveys (CSUR)*, 52(1):1–36, 2019.

46 Siemens AG. Jailhouse hypervisor source code. URL: `https://github.com/siemens/jailhouse`.

47 José Simó, Patricia Balbastre, Juan Francisco Blanes, José-Luis Poza-Luján, and Ana Guasque. The role of mixed criticality technology in industry 4.0. *Electronics*, 10(3):226, 2021.

48 V. Struhár et al. Real-Time Containers: A Survey. In *2nd Workshop on Fog Computing and the IoT*, volume 80 of *OpenAccess Series in Informatics*, pages 7:1–7:9, Dagstuhl, Germany, 2020.

49 Václav Struhár, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, and Alessandro V. Papadopoulos. React: Enabling real-time container orchestration. In *26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2021.

50 T. Tasci, J. Melcher, and A. Verl. A container-based architecture for real-time control applications. In *2018 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC)*, pages 1–9, 2018.

51 Lane Thames and Dirk Schaefer. Software-defined cloud manufacturing for industry 4.0. *Procedia cirp*, 52:12–17, 2016.

52 The Linux Foundation. Homepage of LF Edge Foundation. `https://elisa.tech/`.

53 The Linux Foundation. Kubernetes Home Page. `https://kubernetes.io/`.

54 The Linux Foundation. Real-time group scheduling. `https://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt`. Accessed 17th June 2022.

55 Wind River Systems, Inc. WindRiver VxWorks Virtualization Profile. `http://www.windriver.com/products/vxworks/technology-profiles/#virtualization`. Accessed 17th June 2022.

56 Sisu Xi, Meng Xu, Chenyang Lu, Linh TX Phan, Christopher Gill, Oleg Sokolsky, and Insup Lee. Real-time multi-core virtual machine scheduling in xen. In *2014 International Conference on Embedded Software (EMSOFT)*, pages 1–10. IEEE, 2014.

57 Xilinx. RunX GitHub repository. `https://github.com/Xilinx/runx`.

58 Chengjing Yu, Xudong Ma, Fang Fang, Kun Qian, Shun Yao, and Yanping Zou. Design of controller system for industrial robot based on rtos xenomai. In *2017 12th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, pages 221–226. IEEE, 2017.

59 Wuyang Zhang, Sugang Li, Luyang Liu, Zhenhua Jia, Yanyong Zhang, and Dipankar Raychaudhuri. Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1270–1278. IEEE, 2019.