RT-CASEs: Container-Based Virtualization for Temporally Separated Mixed-Criticality Task Sets

Marcello Cinque
Federico II University of Naples, Italy
wpage.unina.it/macinque
macinque@unina.it

Raffaele Della Corte
Federico II University of Naples, Italy
raffaele.dellacorte2@unina.it

Antonio Eliso
Federico II University of Naples, Italy
antonio.eliso@studenti.unina.it

Antonio Pecchia
Federico II University of Naples, Italy
antonio.pecchia@unina.it

Abstract

This paper presents the notion of real-time containers, or rt-cases, conceived as the convergence of container-based virtualization technologies, such as Docker, and hard real-time operating systems. The idea is to allow critical containers, characterized by stringent timeliness and reliability requirements, to cohabit with traditional non real-time containers on the same hardware. The approach allows to keep the advantages of real-time virtualization, largely adopted in the industry, while reducing its inherent scalability limitation when to be applied to large-scale mixed-criticality systems or severely constrained hardware environments. The paper provides a reference architecture scheme for implementing the real-time container concept on top of a Linux kernel patched with a hard real-time co-kernel, and it discusses a possible solution, based on execution time monitoring, to achieve temporal separation of fixed-priority hard real-time periodic tasks running within containers with different criticality levels. The solution has been implemented using Docker over a Linux kernel patched with RTAI. Experimental results on real machinery show how the implemented solution is able to achieve temporal separation on a variety of random task sets, despite the presence of faulty tasks within a container that systematically exceed their worst case execution time.

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

Keywords and phrases Containers, mixed-criticality, temporal separation, monitoring

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.5

Funding This work is partially supported by the RT-CASE project, funded by the Dept. of Electrical Engineering and Information Technology of the University of Naples Federico II, Italy.

Acknowledgements We are thankful to the anonymous reviewers of the ECRTS program committee for the valuable comments, which allowed us to improve the paper and provided useful guidance to better target our future research efforts on rt-cases.

1 Introduction

A **mixed-criticality system** (MCS) can be defined as a real-time and embedded system integrating software components with different levels of criticality onto a common hardware platform [4]. The trend in the development of MCSs was initially intertwined with the migration from single-core to many-core architectures, which paved the way to the opportunity of having components with different degrees of criticality (e.g., in terms of timeliness and fault tolerance) running on the same hardware. An increasing number of vendors, such as automotive and avionics industries, are considering MCSs to meet stringent non-functional requirements. A typical example is to run hard-real time tasks to control the breaking system of a car on the same board that runs non-critical monitoring tasks for diagnostics, in order to reduce costs, space, and power consumption. Noteworthy, the concept of mixed-criticality is now recognized and supported by the main software standards in automotive (e.g., AUTOSAR – `www.autosar.org`) and avionics (e.g., ARINC – `www.arinc.com`).

A fundamental research challenge for MCSs is to assure the correct execution of high critical tasks, with a disciplined (and possibly user-transparent) use of the underlying shared resources. At least **temporal separation**, and fault isolation of tasks must be guaranteed, in order to avoid that a problem or a delay in a low criticality task can affect a high criticality one. A number of theoretical approaches have been defined, especially for task allocation [15] and schedulability analysis [24] in multi-processor systems. However, when it comes to the actual implementation of the software components, the assurance of separation and isolation properties can easily become a burden for developers. For this reason, several frameworks have been proposed, many of them tailored for a particular domain (e.g., [11] in the automotive) or bound to particular platforms (such as [22] for FPGAs).

Many solutions capitalize on **virtualization technologies** to separate real-time kernels from non real-time ones, by means of different virtual machines (VMs) hosted by specific hypervisors. For instance, Wind River has recently introduced a Virtualization Profile for VxWorks [25], integrating a real-time embedded, type 1 hypervisor into the VxWorks real-time operating system, making it possible to consolidate multiple stand-alone hardware platforms onto a single multi-core platform. Other examples are PikeOS [14], a separation microkernel providing a paravirtualization real-time operating system for partitioned multi-core platforms, and ARINC 653, one of the first industrially used hypervisors, insuring temporal and spatial isolation of different RTOSes. The use of virtualization allows developers to work with their preferred environment, and to deal with separation and isolation issues at the hypervisor level; however, running VMs on a single host has a significant overhead. More importantly, creating VMs for every hardware platform to consolidate may lead to both software and OS stretch. In practice, real-time virtualization solutions – such as the ones above – are designed to deal with the (deterministic) performance penalty introduced by a limited number of VMs. We observe that the overhead caused by replicating entire OS environments makes it unfeasible to scale-up to a large number of applications of different criticalities, especially when they need to be consolidated on a limited number of machines.

A concrete example of such a need is represented by the so-called real-time infrastructure currently under development as part of the control and data acquisition system of the ITER[1] tokamak [26]. The ITER international project aims at the construction of the world's largest nuclear fusion experimental facility in Saint-Paul-lès-Durance, south of France. The construction of ITER is a challenge itself, as well as its future operation, which aims at proving the feasibility of energy production by means of nuclear fusion on Earth. The inherent complexity of ITER (such as any other large-scale critical system) requires to consolidate tens of thousands of applications of different size and criticality – spanning from distributed control to monitoring, sensor reading and network communication – on a limited number of interconnected machines. Different real-time frameworks are currently under development within the ITER project, in order to deal with the complexity of the overall project and to

[1] ITER is a Nuclear Facility INB-174. The views and opinions expressed herein do not necessarily reflect those of the ITER Organization.

facilitate integrated testing and commissioning [26]. Up to now, the ITER organization has not focused yet on a solution to manage the integration and orchestration steps in a simple way, considering the different criticality levels, temporal separation, and fault isolation needs of the components to be run on the available hardware.

1.1 Contributions of the paper

To face these issues, typical of any large-scale mixed-criticality system, in this paper we propose the notion and a possible implementation of **real-time containers** here referred to as: **rt-cases**. Containers are today considered a key technology to achieve high consolidation scalability with minimal space and performance overhead. For instance, Docker relies on an operating system-level virtualization method for running multiple isolated Linux systems (containers) on a single host. Technically a container is a set of processes and libraries isolated from the rest of the machine. Unlike a VM, a container does not need to replicate the whole OS environment for every system, hence reducing the overhead and increasing the number of applications with different criticalities that can be run on a single node. The possible advantages of rt-cases are evident: different developers can still work with their preferred environment, which is desirable in large and heterogeneous teams. The resulting rt-cases can be flexibly deployed on the available hardware, as done today with non real-time containers.

With rt-cases we aim to explore the benefits of combining containers and hard real-time co-kernels. We propose a reference architecture for the rt-cases concept, firstly introduced in [7], [6]. In this paper we explore the design space of rt-cases and discuss possible alternatives for their realization on top of a Linux kernel patched with a real-time co-kernel. We present a specific solution to rt-cases, based on fixed-priority scheduling, runtime execution monitoring, and on-line mitigation. This solution allows to achieve temporal separation without requiring modifications to the underlying real-time kernel. The paper presents the implementation details and technical challenges of such solution, based on Docker and RTAI, and reports the results of an experimentation with randomly generated feasible task sets. Results show that, thanks to our runtime monitoring and mitigation strategy, the tasks running within a container are not affected by *faulty* tasks (i.e., tasks that exceeds their declared worst case execution time) running in a different container, regardless of the criticality of the container. We also present the limitations of our current implementation, and discuss prospected improvements and related trade-offs.

The rest of the paper is organized as follows. Section 2 presents related work in the area and positions our work in context of existing contributions. Section 3 provides the reference architecture and a discussion on potential alternatives to support the implementation of rt-cases. Section 4 discusses the system model, while Section 5 describes a specific implementation, technical challenges and prospected developments. Section 6 illustrates the functioning of the approach with a case study and a measurement campaign. Section 7 concludes the work.

2 Related work

A consolidated trend in the literature and in the industry for time and space partitioning in mixed-criticality systems is to make use of virtualization technologies. These solutions range from type 1 hypervisors to full-featured operating systems, with the aim to completely separate real-time kernels from non real-time ones.

RT-Xen [29] is a real-time hypervisor scheduling framework for Xen [3], a widely used open-source type 1 hypervisor. The RT-Xen project extends Xen to support virtual machines with real-time performance requirements. RT-Xen features a compositional real-time scheduling framework that bridges the gap between virtualization technology and real-time scheduling theory for predictable computing on virtualized platforms. In [28] the second version of RT-Xen is presented, which implements both global and partitioned VM schedulers, with each scheduler being configurable to support dynamic or static priorities and to run VMs as periodic or deferrable servers. RT-OpenStack [27] is a cloud CPU resource management system for co-hosting real-time and regular VMs. RT-OpenStack integrates the real-time hypervisor RT-Xen in the OpenStack cloud management system through a real-time resource interface. PikeOS [14] is a separation microkernel targeted to real-time embedded systems, which aims to provide a partitioned environment for multiple operating systems with different design goals to coexist in a single multi-core platform. It provides full separation in both time and space for multiple software applications running on different criticality levels. XTRATUM [20] is a type 1 hypervisor specially designed for real-time embedded system, which employs para-virtualization techniques. XTRATUM can be used to build partitioned systems and provides both strong temporal separation, through a fixed cyclic scheduler, and strong spatial isolation, since all partitions are executed in processor user mode and do not share memory. The Wind River VxWorks RTOS [25] features a Virtualization Profile that integrates a real-time embedded, type 1 hypervisor into the core of VxWorks, which is able to slow down general purpose operating systems to ensure that real-time operating systems can execute without performance impact.

An alternative to virtualized solutions, which require running a number of virtual machines, is represented by the use of container-based environments. Differently from virtual machines, where the operating system stack is entirely replicated for each virtual machine, containers allow running multiple isolated Linux systems on a single host with minimal space and performance overhead. We believe that containers can be a useful alternative or complement to virtualized partitioned systems with a lightweight solution to sandbox multiple real-time applications into isolated software environments on the same hardware or VM.

Recently, few studies have started to explore the possibility to use containers to run real-time tasks. For example, [18] presents an empirical study on the problem of minimizing computational and networking latencies for Radio Access Networks (RAN) through lightweight containers. The study analyzes the performance of Docker containers running on the top of a Linux kernel patched with Ingo Molnar's preemption patch (PREEMPT-RT). The obtained results highlight that the use of PREEMPT-RT improves latencies on Docker containers when compared to a generic kernel. The paper in [19] proposes a sand-boxed environment, based on Docker containers on a Linux kernel patched with PREEMPT-RT, to deploy the software in automotive industry. Experimental results highlight that the use of containers does not affect the performance of the software when compared with the native environment.

It should be noted that both the solutions make use of PREEMPT-RT in order to meet the real-time requirements of the considered applications. Despite the good results obtained, it is recognized that real-time co-kernels, such as RTAI (www.rtai.org) or Xenomai (www.xenomai.org), outperform PREEMT-RT in terms of latencies and task switch times [8][12], since they make Linux fully preemptable in favor of real-time tasks. Co-kernels also add core real-time support to user level real-time tasks, such as fixed-priority or dynamic scheduling, resource management with priority inheritance, and inter-task communication. An alternative would be to guarantee a fixed CPU bandwidth to containers using server-based scheduling, such as the Sporadic Server (SS) [23], periodic or deferrable servers as done in RT-XEN [28], or the Constant Bandwidth Server (CBS) [1]. For instance, the SCHED_DEADLINE scheduling policy [16], available in the Linux kernel since version 3.14, is an implementation of the Earliest Deadline First (EDF) scheduling algorithm, augmented with a CBS, that makes it possible to isolate the behavior of task groups. However, the use of containers on top of co-kernels with server-based schedulers has not been explored yet.

Starting from these considerations, and from the opportunity, already emerged in the literature, to use containers in real-time environments, in this paper we aim to propose a reference architecture scheme where hard real-time tasks within containers are scheduled by a co-kernel, such as RTAI or Xenomai. Our aim is to fully inherit the advantages of the co-kernel in terms of real-time performance and functionalities, while letting tasks within containers to keep using the same application programming interface, as if they were running on the native system patched with the co-kernel.

3 High-level architecture

Figure 1 depicts the reference architecture underlying the proposed rt-case approach. The idea is to host real-time (rt-) tasks within containers marked with different Criticality Levels (CL, e.g. CL:0 and CL:1 in figure) on top of a patched real-time Linux kernel. The CL is used to establish the relative importance of rt-cases.

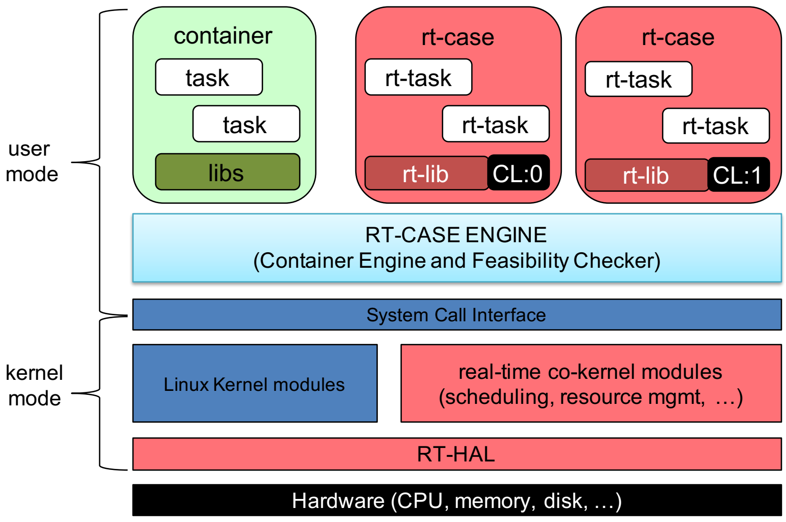


Figure 1 High level scheme of the rt-case architecture.

The RT-CASE engine includes a container engine, e.g., Docker (www.docker.com) or Linux Containers (www.linuxcontainers.org), and a *feasibility checker*, to verify if a new rt-case can be admitted on a running computing node, without affecting the rt-cases already hosted on it. At kernel level, we imagine to have a vanilla Linux kernel patched with a real-time co-kernel, such as RTAI or Xenomai, in order to make the Linux kernel and all the non real-time environment (including traditional containers) fully preemtable by rt-tasks run within rt-cases.

The rt-lib is a key component of the architecture: its objective is to provide a transparent mapping of rt-tasks on the underlying real-time core, depending on the CL of the container, possibly exposing standard primitives to rt-tasks. With this approach, the code of rt-tasks does not need to be modified to run in a container. Hence, the same rt-case can be moved over time on the different machines of a large-scale computing environment, and with a different CL, depending on temporal needs, hardware constraints, and presence of other rt-cases, as regularly done in container-based environments.

3.1 Design alternatives

A simple way to implement the rt-cases architecture is to map whole containers on real-time tasks (see Figure 2a). In this case, a cooperative, non-preemptive user-level scheduler can be implemented in the rt-lib to schedule real-time tasks running within an rt-case. Even if conceptually simple, this solution does not allow to define precise individual deadlines for tasks within containers, which are hence forced to share a container deadline. This also requires to introduce specific primitives for cooperative scheduling in the rt-lib (such as yield primitives) that contradicts our idea to adopt standard primitives for real-time tasks.

Preemptive scheduling for individual tasks within containers overcomes these limitations, but, to achieve temporal separation, tasks belonging to different containers have not to interfere each other. To this aim, a possible solution is to adopt hierarchical scheduling [9][21], e.g., by using Earliest Deadline First (EDF) on task groups (one group for rt-case), each with a fixed bandwidth U_i guaranteed by a server-based mechanism, such as, the Constant Bandwidth Server (CBS) [1] (see Figure 2b), or by using the Hierarchical CBS scheme [17]. This is similar to what done in RT-Xen to assign virtual machines on different virtual CPUs in a multicore environment, with different bandwidths [28]. In this case, to admit a $(n+1)th$ rt-case to an existing computing node with n rt-cases already running, it is sufficient to check that: $\sum_{i=1}^{n} U_i \ + \ U_{n+1} \leq 1$. Even if simple in principle, this solution implies a significant implementation effort in our architectural proposal, since it requires to modify the real-time co-kernel (e.g., RTAI or Xenomai core modules) to implement the hierarchical scheduling solution.

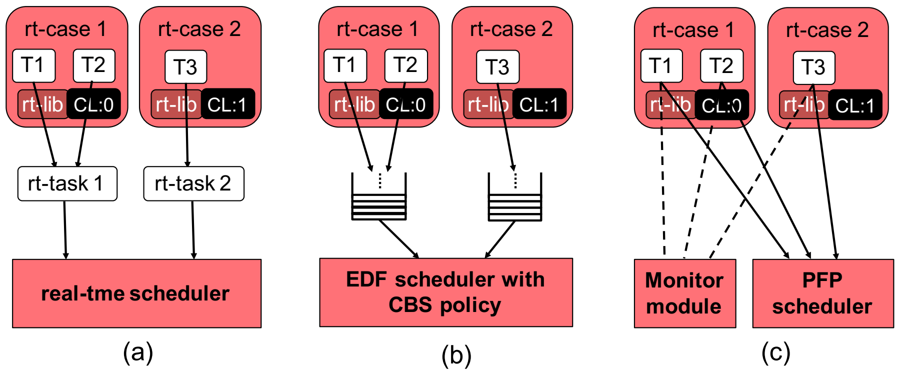


Figure 2 Design alternatives to implement the rt-case model: (a) mapping of whole containers on single rt-tasks; (b) use of task groups and hierarchical scheduling (for instance, EDF with CBS); (c) use of preemptive fixed-priority scheduling, static priority assignment, and temporal protection with on-line execution time monitoring.

A possible compromise to achieve temporal separation without requiring modification to the real-time kernel support is to perform a one-to-one mapping of tasks within containers with real-time tasks at kernel level, using a preemptive fixed-priority (PFP) scheduler and a static priority mapping (see Figure 2c). Using a proper priority assignment, we can assure that rt-tasks running within a container with a high CL (corresponding to a low CL value, e.g., 0 in the figure) cannot be preempted neither by any rt-task running in a container with a lower CL (e.g., 1 in the figure) nor by any non-real time task (running either in other containers or on the host OS). A priority assignment algorithm is needed in this case, in order to assure both the feasibility of individual tasks and the feasibility of whole rt-cases to be admitted on a computing node. The advantage of this solution is simplicity: in principle, it only requires a PFP scheduler, such as rate monotonic (RM), largely available in existing real-time operating systems. In practice, the solution requires a temporal protection mechanism (with on-line task execution time monitoring), in order to assure that faulty tasks, exceeding their worst case execution time (WCET), do not interfere with tasks belonging to different containers.

In this paper we focus on the implementation of the rt-cases model following the last alternative, i.e., fixed priority scheduling with monitoring, as in Figure 2c. Future work will be devoted to the implementation of the hierarchical scheduling alternative (Figure 2b) and to the comparison between the two solutions.

4 System model

We assume a system composed of M rt-cases, each of them with a criticality level $CL_j\ :\ j = 0...M-1$. A criticality level CL_j can assume an integer value in the interval $[0, CL^{max}]$, with 0 being the highest CL, and CL^{max} the lowest CL. Each rt-case hosts one or more periodic hard real-time tasks τ_i, characterized by a worst case execution time C_i and a period T_i. We assume T_i to be coincident with the relative deadline of the task. Overall, the system is composed by a set Γ of N tasks, each of them assigned to an rt-case with a given CL. With $\Gamma(CL)$ we indicate the subset of tasks with criticality level CL. By construction: $\Gamma = \Gamma(CL_0) \cup ... \cup \Gamma(CL_{M-1})$ and $\Gamma(CL_k) \cap \Gamma(CL_h) = \emptyset$, where $CL_k \neq CL_j$.

For example, considering the system depicted in Figure 3, with 4 rt-cases, 8 tasks, and 3 criticality levels, we have: $M = 4$; $N = 8$ $CL^{max} = 2$; $\Gamma = \{\tau_1\ ...\ \tau_8\}$, $\Gamma(0) = \{\tau_1, \tau_2\}$, $\Gamma(1) = \{\tau_3, \tau_4, \tau_5, \tau_6\}$, and $\Gamma(2) = \{\tau_7, \tau_8\}$.

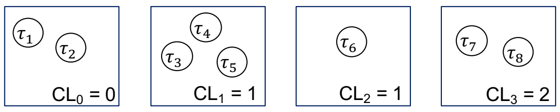


Figure 3 An example of decomposition of tasks within containers: circles represent tasks and squares represent rt-cases with different CLs; rt-cases 1 and 2 have the same CL.

Each task τ_i is characterized by a static priority p_i. Priority values range in the integer interval $[0, N-1]$, where N is the total number of tasks, assuming 0 to be the highest and $N-1$ the lowest priority values. Priorities have to be assigned to tasks according to the CL of the container they belong to. In particular, to avoid that tasks in a high criticality rt-case are preempted by tasks in a lower criticality rt-case, we assume that:

$$CL_k < CL_H \ \wedge\ \tau_i \in \Gamma(CL_k)\ \wedge\ \tau_j \in \Gamma(CL_h) \iff p_i < p_j \tag{1}$$

In other terms, tasks belonging to high criticality containers must receive high priorities, and viceversa.

Once priorities are assigned, tasks can be scheduled by a PFP scheduler. Having assumed a PFP scheduler, a simple method to assign priorities to tasks, while checking the feasibility of the system, is to adopt the Audsley's priority assignment algorithm [2], with the classical Joseph-Pandya's response time analysis (RTA) schedulability test [13]. RTA consists in computing the response time R_i of a task τ_i as: $R_i = C_i + \sum_{p_k<p_i} \lceil R_i/T_k \rceil \cdot C_k$. R_i takes into account the interference that a task can suffer due to preemptions by higher priority tasks. The task set is schedulable if and only if $R_i \leq T_i\ \forall i$.

The Audsley's algorithm is based on the following lemmas:

- *Lemma 1*: the worst case response time of a task τ_i can be determined with a test (such as RTA), by knowing the tasks that have priorities greater than τ_i, but without knowing the specific assignments of such priorities;
- *Lemma 2*: if a task is schedulable considering a given priority assignment, then it remains schedulable if it receives a higher priority.

We adapted the algorithm to our case, as shown in Alg. 1. In particular, to be compliant with our assumption in (1), we force the assignment of priorities to proceed from tasks belonging to the lowest criticality rt-case (with $CL = CL^{max}$) to the highest one (with $CL = 0$, see line 2 in the algorithm), starting from the lowest priority ($N - 1$) to the highest one (0). A priority is assigned to a task τ_i if $R_i < T_i$ (see lines 5-6). If we are not able to assign the priorities to all the tasks of a given CL, then the task set is unschedulable (see lines 10-11). If all priorities are assigned, the task set is schedulable.

Algorithm 1 Priority Assignment Algorithm (Audsley's algorithm adaptation).

```
1:  $p \leftarrow N - 1$
2:  for $CL = CL^{max}$ down to 0 do
3:      while $p \geq 0$ do
4:          for each unassigned task $\tau_i$ in $\Gamma(CL)$ do
5:              if $R_i \leq T_i$, with all unassigned tasks assumed to have priorities $< p$ then
6:                  assign $p$ to $\tau_i$
7:                  $p = p - 1$
8:              end if
9:          end for
10:         if not all tasks in $\Gamma(CL)$ can be assigned then
11:             return UNSCHEDULABLE;
12:         end if
13:         break
14:     end while
15: end for
16: return SCHEDULABLE;
```

Such priority assignment solution assures isolation of high criticality rt-cases from lower ones, since, by construction, a task running in an rt-case with CL_k can never be preempted by a task running in an rt-case with $CL_h > CL_k$, according to (1). However, we must also ensure that faulty tasks in high criticality rt-cases (e.g., a task instance, or job, exceeding its C_i) do not affect tasks in low criticality rt-cases. Hence, we assume the system to be equipped with a temporal protection mechanism implemented by a *monitor*, running on a different CPU than the one(s) running rt-cases. The monitor has to measure the execution time of tasks at runtime, in order to interrupt a job of a periodic task whenever it exceeds its declared C_i.

With reference to the rt-case architecture in Figure 1, the proposed priority assignment algorithm has to be implemented by the feasibility checker within the RT-CASE engine. The algorithm can be run whenever a new rt-case becomes ready, to check if it can be admitted to a CPU hosting other rt-cases without affecting their execution.

It has to be noted that the assignment obtained with Alg. 1 may not reflect the priority assignment originally planned by the application developer for the tasks to be run within his/her container. This might still be fine in the cases where the developer does not care about individual task priorities, as long as the assignment guarantees that task deadlines are met. However, if application dependent constraints on priorities must be met, Alg. 1 cannot be used as it is. In this case, our implementation leaves to the developer the chance to manually assign the priorities to tasks, at his/her own risk. A viable alternative is to force the algorithm (i) to allot disjoint priority intervals to different containers (even for rt-cases with the same CL), and (ii) to assign priorities to tasks within a rt-case by respecting their original relative order, in terms of the priorities assigned by the developer. A different solution is to adopt the rt-case design alternative based on hierarchical scheduling and server-based approaches (see Figure 2b), since it makes priority assignments within rt-cases independent from the feasibility checker. We leave the implementation and evaluation of this solution, along with the comparison with the currently developed scheme, as future work.

5 Implementation details

We present the details on the implementation of our proposal on top of RTAI (Real Time Application Interface), a real-time co-kernel extension for Linux. RTAI is an open-source project born to add real-time capabilities to standard Linux kernels. It is conceived as a patch of the Linux Hardware Abstraction Layer (the RT-HAL) that makes the Linux kernel fully preemptable in favor of real-time tasks, by masking the interrupts handling mechanism and by redirecting interrupts to the Linux kernel only when there is no real-time activity to be performed. The RT-HAL is complemented by a number of RTAI modules, providing a rich set of services for real-time tasks running at the user level, among which real-time task management, real-time inter-process communication with priority inheritance, etc.

We implement our proposal on a vanilla Linux kernel 4.9.80 patched with RTAI 5.1. The proposal encompasses three main components, as depicted in Figure 4: the *RT-CASE engine*, the *RT-lib* and the *RT-CASE monitoring*. The implementation of these components is presented in the following[2].

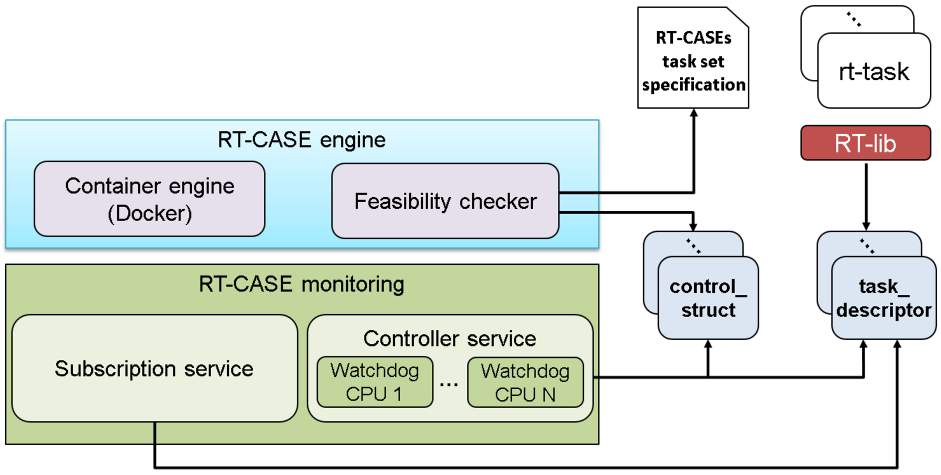


Figure 4 System components.

[2] The source code of the components has been made publicly available at `http://www.dessert.unina.it/RT-CASE.zip`.

5.1 RT-CASE engine

The *RT-CASE engine* represents a macro-component dedicated to the orchestration of the rt-cases and, more important, to the feasibility evaluation of the rt-task set to be run inside the rt-cases. The RT-CASE engine is composed by two components: the *feasibility checker* and the *container engine*.

The **feasibility checker** aims to verify the schedulability of a given task set Γ. It is a Python script that accepts the *specification* of a task set as input. For each task τ_i, the specification contains: (i) id, (ii) name, (iii) the period T_i, (iv) the worst case execution time C_i, (v) the CL and the name of the container hosting the task. Given the task set, the feasibility checker runs the priority assignment algorithm (Alg. 1); if the task set is schedulable, the component returns the priorities to be assigned to tasks. In particular, it fills a data structure, named *control_struct*, containing for each task: its id, its name, the rt-case where it runs, its priority and its C_i. The structure is then used at run-time by the *RT-lib*, as a *contract* for the tasks, as described in Section 5.2.

The **container engine** is used to launch feasible rt-cases. In our implementation, we use Docker as container engine. Each rt-case is a Docker container, including in its image the *RT-lib* and the executables of the rt-tasks to run. Each rt-case is run providing the *PIDC* env variable, which contains the ID of the container; the variable is used by the *RT-lib* as described in Section 5.2. It is important to note that only user-space rt-tasks are allowed to run inside the rt-cases. Kernel-space rt-tasks can be run at host level; however, they are not managed by our proposal. More important, a number of capabilities as well as the access to some host devices have to be granted to rt-cases in order to allow the execution of rt-tasks. For example, if the tasks of a given rt-case needs to access to both the `/dev/rtai_shm` and `/dev/rtc0` device files (representing the RTAI shared memory and real-time clock device files, respectively), the rt-case has to be lunched with the following Docker flags: `-device=/dev/rtai_shm:/dev/rtai_shm -device=/dev/rtc0:/dev/rtc0`. Similarly, since rt-tasks usually need to set the real-time clock, the rt-cases are lunched with the Docker flag `-cap-add=SYS_TIME`.

5.2 RT-lib

The RT-lib exports the user-space APIs provided by RTAI, which are made available for each rt-task running inside an rt-case. More in details, the RT-lib encapsulates a modified version of the RTAI LXRT library, i.e., the library allowing the access to all the services made available by RTAI and its schedulers in user-space; a number of primitives are modified in order to grant *naming isolation* and to assign priorities to rt-tasks running inside rt-cases.

Naming isolation allows avoiding clashes due to the use of the same name for a rt-task or a resource from applications running in different rt-cases. We modify the LXRT *rt_task_init_schmod* primitive, which allows to create and initialize rt-tasks in user-space, to combine the name of the rt-task, generally provided as input parameter, with the ID of the related rt-case, which is defined through the *PIDC* env variable of the rt-case. Similar modifications have to be applied to the other LXRT initialization primitives, such as *rt_typed_sem_init* and *rt_typed_mbx_init*, which allow the initialization of semaphores and mailboxes, respectively.

To perform the priority assignment, we modify the *rt_task_init_schmod* primitive. The modified version of this primitive sets the task priority to the one defined in the *control_struct* (the contract) generated by the *feasibility checker*. If the priority value is not defined in the *control_struct*, i.e., the feasibility check has not been executed for the current task set or the task set is not feasible, the primitive does not change the priority.

Finally, in order to assure temporal separation through the protection mechanism implemented by the monitoring component (see next section), both the *rt_task _init_schmod* and *rt_task_wait_period* are modified. The *rt_task_init_schmod* is modified to create and initialize a data structure, named *task_ descriptor*, used by the monitoring component and containing the information about the current task. The descriptor, created if the monitoring is active, logically extends the Linux *task_struct* (without actually modifying it) with a number of fields required by the monitoring component. The main fields of the the *task_descriptor* are detailed in Table 1. The *rt_task_wait_period*, i.e., the RTAI primitive that suspends the execution of a periodic hard real-time task until the next period, is modified to count the number of cycles of the task. This parameter is reported in the *cycle* field of the *task_descriptor*. Before invoking the syscall allowing the task to wait the next period, the primitive first verifies if the monitoring is active and, in this case, increments the *cycle* field. When the task is resumed at the next cycle, the primitive also saves the current time in the *last_switch_time* field of the *task_descriptor*, which is used by the monitoring component to measure the execution time of the task (see next section).

It should be noted that all the modifications explained above are totally transparent to the developer. In fact, the modifications affect only the body of the primitives, while their signatures are left unchanged. More important, the modifications only affect the RTAI LXRT user-space library, without any change to the RTAI kernel level source code. This avoids the need to re-compile the kernel; only the library source code needs to be compiled and installed in the containers, fostering an easy and quick adoption of the solution. Finally, it is important to note that if the monitoring component or the feasibility checker is not activated, all the modified primitives act as the original ones. This leaves the programmer the freedom to manually assign the priorities at his/her own risk.

Table 1 Main fileds of *task_descriptor*.

Field	**Description**
wcet	Worst Case Execution Time (WCET, or C_i) of the task
exec_time	Execution time of the task in the current cycle
last_switch_time	Time of the last context switch involving the task
last_overrun	Last time the task has been found in overrun
task_alarm	Pointer to the task in charge to manage the task, when it becomes faulty
cycle	Current cycle of the task
last_cycle	Last cycle of the task
overtime	Number of overtimes of the task
overrun	Number of overruns of the task

5.3 RT-CASE monitoring

The *RT-CASE monitoring* component aims to provide temporal protection to rt-cases. The component prevents that a faulty task (i.e., a task exceeding its WCET or having an activation frequency higher than the one declared during the feasibility checking) running within an rt-case may affect the tasks running in rt-cases with lower or equal CLs. To this aim, the component periodically checks the execution time and the activation frequency of the rt-tasks running inside the rt-cases; in addition, it measures the number of *overruns* and *overtimes* of each task, i.e., the number of times a task exceeds its deadline and WCET, respectively. When a faulty task is detected, the RT-CASE monitoring implements one of the following *policies* to guarantee temporal protection:

- `KILL`: the task is killed in a forced way;
- `SUSPEND`: the task is suspended indefinitely;
- `FORCE_PERIOD`: the task is suspended and it will be resumed at the next period activation;
- `SIGNAL`: a notification is sent to a given task, which will be in charge to take action as a consequence of the fault.

The policy to adopt represents one of the parameters of the monitoring component, which also includes the CPU it has to be run on and the monitoring period.

The RT-CASE monitoring component has been developed as a kernel module. The module launches a number of tasks on a dedicated processor in order to avoid that the monitoring operation affects the rt-tasks running inside rt-cases. Therefore, the proposed monitoring approach requires a multiprocessor hardware configuration with at least 2 CPUs, one for monitoring and the other one for the rt-tasks running in rt-cases. The usage of a single processor system is also possible (although we leave it as future work); however, there are two constraints to be considered in this case: (i) the monitoring task has to be taken into account in the feasibility checking and (ii) it has to be the task with the highest priority.

The RT-CASE monitoring component provides two main services: *subscription service* and *controller service*.

The **subscription service** is an aperiodic task that is activated when the `SIGNAL` policy is used. The service allows developers to subscribe a *control task* for a rt-task to monitor. The control task will be notified each time the monitored rt-task is detected as faulty, as defined by the `SIGNAL` policy. When a control task requests the subscription for a given rt-task, the subscription service retrieves the *task_descriptor* of the rt-task and stores the pointer to the control task into the *task_alarm* field. This policy is useful to handle the recovery of tasks in overtime at the user-space, within rt-cases; however developers must be aware of it. The other policies do not require modifications to the source code of tasks.

The **controller service** is the main service of the monitoring component since it aims to detect faulty tasks. The service is conceived as a set of periodic real-time tasks - *watchdogs* hereinafter. A watchdog is created for each CPU running the rt-cases. Each watchdog periodically executes the following steps: (i) gets the task currently running on the CPU it monitors; (ii) if the current task is a periodic, user-space, hard real-time task, it retrieves the pointer to its Linux *task_struct*; (iii) it retrieves the rt-case *task_descriptor* of the task and measures its execution time and activation frequency; (iv) if a deviation is detected, i.e., the execution time exceeds the WCET of the task and/or the activation frequency is higher than the one declared during the feasibility checking, the configured policy will take place for the task (i.e., `SIGNAL`, `KILL`, `SUSPEND` or `FORCE_PERIOD`).

It should be noted that in order to measure the execution time and the number of overruns and overtimes, the watchdogs leverage information contained in the *task_descriptor*, e.g., *cycle*, *last_cycle* and *last_switch_time* fields, as well as RTAI primitives (e.g., *rt_get_time_cpuid* to obtain the current time in internal count units on a given cpu). In addition, the *switch_time[cpu]* provided by RTAI is used, which indicates the time of the last context switch on the indicated *cpu*. Alg. 2 describes the algorithm used by the watchdogs to measure the execution time of rt-tasks.

The algorithm evaluates the execution time in three different cases, depicted in Figure 5, which shows two rt-tasks running in rt-cases with different CLs: (a) the current task is in a cycle different from the one of the last monitoring check (at M_0 and M_4 for Task 2, at M_2 for Task 1); (b) the current task is in the same cycle of the last monitoring check and it has not been preempted (at M_1, M_5 and M_6 for Task 2); (c) the current task is in the same cycle of the last monitoring check and it has been preempted in favor of a task with a higher

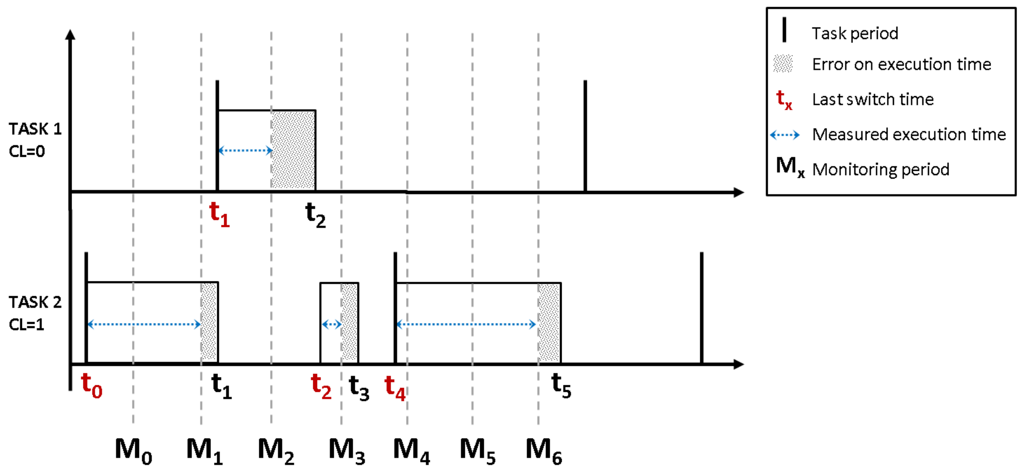


Figure 5 Execution time measurement.

Algorithm 2 Execution time evaluation algorithm.

```
 1: if tsk_desc–>cycle == tsk_desc–>last_cycle then
 2:     if tsk_desc–>last_switch_time == switch_time[cpu] then
 3:         tsk_desc–>exec_time = rt_get_time_cpuid (cpu) - tsk_desc–>last_switch_time;
 4:     else
 5:         tsk_desc–>last_switch_time = switch_time[cpu];
 6:         tsk_desc–>exec_time += rt_get_time_cpuid (cpu) - tsk_desc–>last_switch_time;
 7:     end if
 8: else
 9:     tsk_desc–>last_cycle = tsk_desc–>cycle;
10:     switch_time[cpu] = tsk_desc–>last_switch_time;
11:     tsk_desc–>exec_time = rt_get_time_cpuid (cpu) - tsk_desc–>last_switch_time;
12: end if
```

priority (at M_3 for Task 2). In both cases (a) and (b) the current execution time of the task is evaluated as the difference between the current time, provided by *rt_get_time_cpuid*, and the *last_switch_time* (Alg. 2 - lines 3 and 11), i.e., M_1 - t_0, M_6 - t_4 (Task 1) and M_2 - t_1 (Task 2) in Figure 5; however, in case (a) the algorithm also updates the *last_cycle* value and updates the *switch_time[cpu]* with the *last_switch_time* (lines 9 and 10) since a new cycle is started. It should be noted that the update of the *switch_time[cpu]* is required to avoid an erroneous evaluation of the execution time when a task enters in a new cycle without any context switch; in fact, in this case the *switch_time[cpu]* is not updated by RTAI. In case (c) the algorithm measures the execution time of the task as the sum between the last measured execution time for the task and the difference between the current time and the *last_switch_time* (Alg. 2 - lines 6), i.e., (M_1 - t_0) + (M_3 - t_2) in Figure 5 for Task 2; noteworthy, the *last_switch_time* here is updated to the *switch_time[cpu]* value (Alg. 2 - line 5), i.e., t_2, in order to consider the begin of the new activation of the task.

Alg. 3 describes the algorithm used by watchdogs to detect tasks in overrun and overtime, and rt-tasks activating before the expected time. An overrun is detected by evaluating the difference between the current time, the *periodic_resume_time*, i.e., the time the task has been or will be resumed after a cycle, and the period of the task (Alg. 3 - line 3). Both the *periodic_resume_time* and the task period are provided through the *task struct* of the task (*task* in Alg. 3); in addition, the function *count2nano* used in the algorithm converts a value

Algorithm 3 Detection algorithm.

```
1: //CHECK OVERRUN
2: overrun = count2nano(rt_get_time_cpuid (cpu) - task–>periodic_resume_time - task–>period);
3: if overrun > 0  AND tsk_desc–>last_overrun != tsk_desc–>cycle AND tsk_desc–>cycle > 1  then
4:     tsk_desc–>overrun++;
5:     tsk_desc–>last_overrun = tsk_desc–>cycle;
6: end if
7: //CHECK OVERTIME
8: if count2nano(tsk_desc–>exec_time) > tsk_desc–>wcet AND tsk_desc–>cycle > 0 then
9:     tsk_desc–>overtime ++;
10:     start_policy(tsk_desc);
11: end if
12: //CHECK ACTIVATION
13: if rt_get_time_cpuid (cpu) < task–>periodic_resume_time) then
14:     start_policy(tsk_desc);
15: end if
```

in tick count to nanoseconds. If the obtained value is positive, the rt-task is considered in overrun, since its execution time exceeds its deadline (that we assume to be equal to the period of the task). For example, in Figure 6 it is depicted a task exceeding its deadline. It can be noted that the sum of the *periodic_resume_time* and the task period is lower than the current time; therefore, the difference evaluated by the detection algorithm is a positive value, which allows detecting the overrun at M_0. When a task is found in overrun, the algorithm increments the number of overruns (line 9), i.e., the *overrun* field of the *task_descriptor*, and saves the cycle of the overrun (line 10), i.e., the *last_overrun* field.

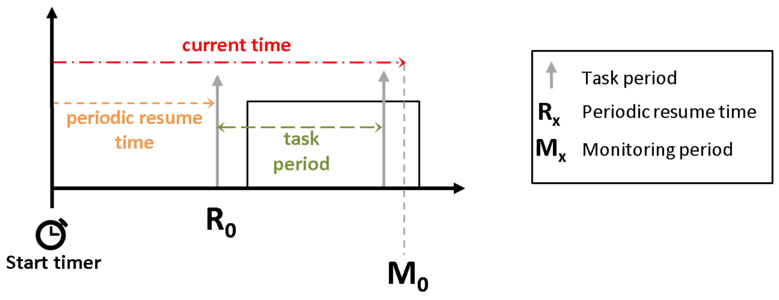


Figure 6 Overrun detection.

Overtimes are evaluated by comparing the evaluated execution time with the WCET of the task (Alg. 3 - line 8); if the execution time is greater than the WCET, the task is considered in overtime and the overtime field of the *task_descriptor* is updated (line 9). Finally, in order to detect task activating before the expected time, the algorithm verifies if the current time is lower than the *periodic_resume_time* (Alg. 3 - line 13); in this case, the task has been activated before expected. If an overtime or a premature task activation occurs, the algorithm calls the *start_policy* function, which applies the policy configured for the temporal protection (lines 10 and 14).

5.4 Discussion

The algorithm currently implemented for RT-CASE monitoring has the benefit of a constant computational complexity, which does not depend on the number of tasks running on the monitored CPUs. At each period, the monitor performs checks on the currently running rt-task, with a relatively small number of lines of code. We measured the typical execution time of an instance of the implemented monitor, e.g., the time it takes to perform a check within a period. The measurements have been conducted in our deployment (composed by a 2.4 GHz dual-core Intel Core i7-5500U machine, equipped with 8 GB DDR3 RAM and a Crucial MX500 SSD) and they show that a watchdog instance is able to run within about 3,500ns in the worst case, regardless of the number of tasks to monitor. This relatively short monitoring time allows very fine grain monitoring periods, in the order of few microseconds, if the monitor is run on a dedicated CPU.

As a drawback, it is important to note that the current implementation of the monitor provides an under-estimation of the execution time of rt-tasks. In fact, each time a task is preempted in favor of a task with a higher priority or terminates its execution in the current activation, the watchdog is not able to estimate the time between the last monitoring activation and the termination of the task execution. For example, in the case depicted in Figure 5 a watchdog is not able to measure the execution time in $[M_2, t_2]$ for Task 1, and in $[M_1, t_1]$, $[M_3, t_3]$ and $[M_6, t_5]$ for Task 2. However, the extent of the estimation error can be easily evaluated. Given a task with a priority p_i and period T_i, the error of the execution time measured by a watchdog in the worst case is given by $T_{mon} \cdot (P + 1)$. T_{mon} is the monitoring period, while P is the number of times the task can be preempted in favor of tasks with a higher priority. Thanks to the adoption of a PFP scheduling policy, the value of P can be easily obtained as: $\sum_{p_k<p_i} \lceil T_k/T_i \rceil$. It should be noted that T_{mon} is multiplied by $P+1$, since we have always an underestimation of the execution time at the task termination in the current period, also if the task is not preempted, as in $[M_6, t_5]$ of Figure 5. Such estimation error can be taken into account in the feasibility analysis (see Section 6.3).

We defined a possible solution for reducing the underestimation obtained with the current implementation. The solution requires that the watchdog stores the current task in a *last monitored task* variable, before terminating the current monitoring activation. This allows the watchdog to update the execution time of the previous monitored task during the next activation; the new value is obtained as the difference between the current *switch_time[cpu]* and *last_switch_time* of the previous monitored task. For example, in Figure 5 the execution time of the Task 2 will be updated at M_2 as t_1 - t_0. In this case, the underestimation error committed by the monitor is T_{mon} at most, when the task ends its execution between two monitoring executions. However, this solution requires the watchdog to analyze two tasks at each cycle. In particular, it has to evaluate the execution time of both the current and the previous task, and it has to execute the overtime and overrun detection for both of them. This may have an impact on the execution time of watchdog instances, consequently requiring to reduce the maximum frequency of monitoring. We leave the implementation of this new version of the monitoring approach, and its evaluation, as future work.

6 Case study

We present a case study of the proposed approach, which consists in a typical setup of *high*, *medium*, and *low* criticality containers, with CL values equal to 0, 1, and 2, respectively. Figure 7 depicts the case study: we hypothesize that the tasks hosted by the *medium* critical rt-case exhibit a **faulty behavior** by exceeding their WCET. We propose a mixture of experiments to gain insights into the following points: (i) thanks to our monitor, the faulty tasks in an rt-case do not impact tasks hosted by the remaining containers, (ii) trade-offs of the monitor and viable workarounds. We deploy the containers on a 2.4 GHz dual-core Intel Core i7-5500U machine, equipped with 8 GB DDR3 RAM and a Crucial MX500 SSD. The installation consists of the following key components: Docker 17.031-ce, Ubuntu 16.04, Linux kernel 4.9.80 and RTAI 5.1. We devote special care to achieve a representative setup although

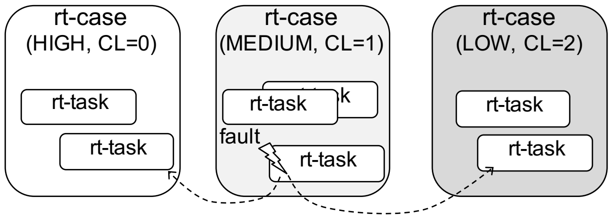


Figure 7 Representation of the case study.

by means of general purpose hardware. As such, the Linux kernel has been configured in order to prevent common sources of non-determinism, such as energy savings, frequency scaling and hyper trading. Monitor and containers are pinned to distinct cores; the *monitoring period* is set to 8,000ns. We would like to point out that this case study is not meant to be exhaustive, but it has been arranged with the aim of eliciting reasonable operating conditions, which illustrate the basic functioning of the approach and its concrete implementation.

6.1 Experiments

We conduct a campaign of experiments[3] to gain insights into the temporal separation across the containers. Each experiment of the campaign consists in generating a **task set** with a total **utilization** U. We run the task set for one minute, and record the following outcomes for each task at the end of the run:

- **overtime**: number of times the task exceeds the WCET;
- **overrun**: number of times the task misses the deadline.

We assess 5 levels of utilization within U={0.45, 0.55, 0.65, 0.75, 0.85}; since the task set is generated randomly – as explained in the following – experiments are replicated 30 times for each level U, thus leading to 150 experiments. Moreover, experiments are done both *without* and *with* our monitor, i.e., total 2×5×30 experiments.

For each experiment, the task set with utilization U is synthesized according to the approach in [10]. As such, if we denote by $U_i = \frac{C_i}{T_i}$ the utilization of a task (with C_i and T_i denoting the WCET and the period, respectively), we obtain $U = \sum_{i=1}^{n} U_i$ where n is the number of tasks. In our case study, n is set to 14 because we note that with a higher number of tasks and at levels of utilization higher than 0.85 (i.e., the maximum level of U assessed) it is becomes hard to find feasible task sets with fixed priority scheduling. Once the task set is generated, we (i) assign the tasks to the containers, i.e., 3, 7, and 4 tasks, to the *high*, *medium*, and *low* criticality container, respectively, and (ii) assign the priorities with the feasibility checker based on Alg. 1 in Section 4. The feasibility checker assigns the priorities only if the task set is schedulable, so to be sure that potential overruns are not merely caused by unfeasible task sets.

The tasks execute a *CPU-bound* workload made of arithmetic operations for a time consistent with the declared WCET; however, in order to emulate *faulty* behaviors, the tasks allotted to the *medium* criticality container deliberately keep the CPU busy up to eight times the WCET. For the experiments with monitoring *on*, we use the `SIGNAL` policy: accordingly, each task is accompanied by the corresponding *control task*, which terminates the task under-monitoring when it receives a notification from the monitoring system.

[3] The code used for experiments has been made public available within the source code of the proposal.

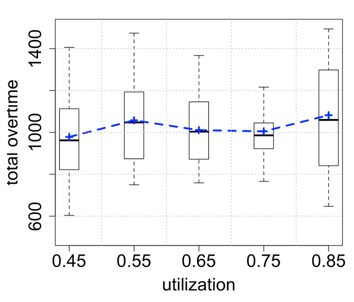
Table 2 shows the outcome of one experiment replication with no monitor and a task set with U=0.85. For each task, we record **overtime** and **overrun** at the end of the experiment by accessing the corresponding `task_descriptor`. In this specific instance, the tasks belonging to the *medium* criticality container cause 1,067 overtimes in total, which reflects into total 92 overruns by the tasks within the *low* criticality container. The *high* criticality container is not affected by the faulty tasks, as expected.

Table 2 Outcome of one experiment replication with no monitor and U=0.85.

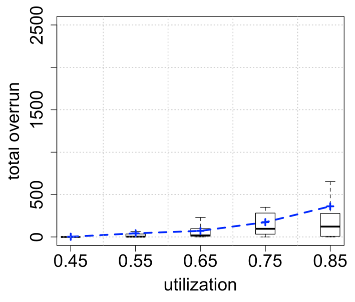
C_i (ns)	T_i (ns)	U_i	criticality	priority	overtime	overrun
978854	7071458	0.138	*high*	1	0	0
621582	7566834	0.082	*high*	2	0	0
1380333	8008509	0.172	*high*	3	0	0
160056	4869494	0.033	*medium*	4	339	0
34866	6432178	0.005	*medium*	5	220	0
103005	6606403	0.016	*medium*	6	169	0
624445	7667583	0.081	*medium*	7	133	0
123770	8385032	0.015	*medium*	8	101	0
95034	8792447	0.011	*medium*	9	70	0
167473	9991428	0.017	*medium*	10	35	1
77299	5288777	0.015	*low*	11	0	21
250561	6660143	0.038	*low*	12	0	4
1438550	7360892	0.195	*low*	13	0	43
283605	8931703	0.032	*low*	14	1	24

6.2 Results with no monitoring

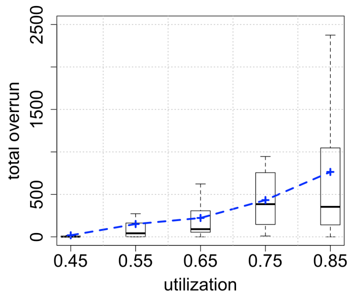
We discuss the results obtained at the termination of the experimental campaign. Figure 8 summarizes the key outcomes obtained *without* the monitoring approach, i.e., the tasks allotted to the *medium* criticality container are free to exceed their WCET with no temporal separation.



(a) Total overtime *medium* criticality container.



(b) Total overrun *medium* criticality container.



(c) Total overrun *low* criticality container.

Figure 8 Total overtime and overrun without our monitoring approach.

Boxplots in Figure 8a summarize the variability of the **total overtime** at increasing utilization U. One observation of the *total overtime* is the sum of the overtime of the tasks allotted to the *medium* criticality container at the end of one experiment: each boxplot is obtained with the data from 30 experiment replications for a given U as stated above.

On average, the tasks produce around 1,027 total overtime *per* experiment across the different levels of utilization U, such as shown by the dotted line in Figure 8a. By means of ANOVA, we observe that there is no statistically significant effect of U on the total overtime with respect to the inherent variability of the overtime across the experiments. This is an expected result that is given by the way we generate the task sets and emulate the faulty tasks.

Similarly to Figure 8a, boxplots in Figure 8b show the **total overrun** – i.e., the sum of the overrun – of the tasks allotted to the *medium* criticality container. Again, each boxplot summarizes the outcomes from 30 experiment replications. Figure 8c shows the total overrun for the *low* criticality container. In both Figure 8b and 8c the dotted line represents the mean of the observations. It can be noted that the total overrun increases as the utilization increases. Noteworthy, this is not *merely* caused by an increase of the total overtime of the tasks in the *medium* criticality container, because we have excluded the existence of a statistically significant trend with respect to U, as stated above. Rather, it can be reasonably stated that at low levels of utilization U (e.g., 0.45-0.55) the remaining amount of free CPU bandwidth provides higher chance to tolerate faults that reflect into exceeding the WCET. This chance is progressively smaller as U increases, which causes higher total overrun.

Regarding the tasks allotted to the *high* criticality container, we observe no overrun. Since this container hosts high-priority tasks – and given that RTAI has a PFP scheduler – even if the tasks in the *medium* criticality exceeds the WCET, they do not affect the *high* criticality container. The effect of priorities can be also noted by comparing Figure 8b and 8c, where the total overrun is smaller for the *medium* criticality container.

6.3 Results and considerations with the proposed monitor

We analyze the outcome of the remaining 150 experiments obtained by monitoring the tasks with the proposed approach, which means that any task exceeding the declared WCET is terminated by its corresponding *control task* upon the receipt of a `SIGNAL`. Differently from the results discussed in Section 6.2, we observe *no* overrun across all the levels of utilization assessed in this case study. Our monitor prevents the overrun affecting the tasks allotted both to the *medium* and *low* criticality container.

Beyond this favorable finding, which pertains the basic functioning of the proposed architecture, we would like to discuss *cons* of our monitor and potential workarounds. At this stage of development, we have pursued a lightweight implementation of the monitor, with a *reactive* protection approach, as it can be noted by the mitigation policies described in Section 5.3. In consequence, it is reasonable to assume the existence of a **latency** of the detection mechanism, i.e., the time between (i) a faulty task exceeding the WCET, and (ii) the effect of the mitigation, e.g., a `KILL`, triggered by our monitor.

Latency may be an issue in some very strict scheduling scenarios. For example, let us hypothesize a scenario – generated with *SimSo* [5] – which consists of six tasks with the parameters shown in Table 3 and total utilization U=1. Figure 9 shows a schedule of the tasks (*CPU* row) and a detailed view for the tasks. It can be noted that there is no timeslot left for tolerating any latency of the detection: for this worst case scenario any minimum delay at terminating the tasks that exceed the WCET will likely cause an overrun.

In principle, it is possible to *tolerate* the latency of the monitor with the following workaround. Let C_{mon} denote a WCET estimation for the *latency* of the detection. Given a task set, for the *sole* purpose of the schedulability test we use the declared WCETs of the tasks *augmented* by C_{mon}, i.e., C_i+C_{mon}. If the task set is admitted, any task can safely exceed its WCET up to C_{mon} since we assure – by construction – enough time for the actions of the monitor to take place. The proposed adjustment of the WCETs has a trade-off, which

Table 3 An example of worst case scenario with U=1.0.

ID	C_i (ms)	T_i (ms)	criticality	priority
Task T1	1.66	10	*high*	1
Task T2	8.33	50	*high*	2
Task T3	8.33	50	*medium*	3
Task T4	16.66	100	*medium*	4
Task T5	25.00	150	*low*	5
Task T6	50.00	300	*low*	6

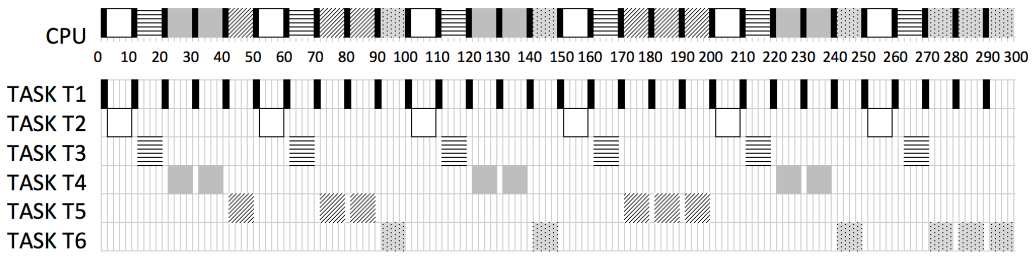


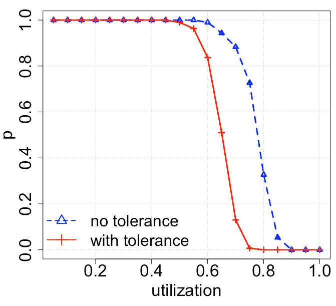
Figure 9 Representation of the worst case scenario (adapted from *SimSo*).

reflects into an inherent increase of the declared utilization of a task set. In the following, we explore this proposition by analyzing the sensitivity of the probability (p) to find schedulable task sets at increasing utilization *with*/*without* tolerance of the latency.

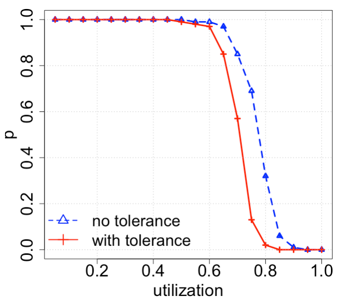
For this analysis we assume the `KILL` policy, i.e., a faulty task is terminated outright by the monitor without the mediation of the corresponding *control task*. In consequence, the duration of the mitigation action – *per se* – is negligible, and the main contribution to C_{mon} consists of the time taken by the monitor to detect a faulty task. As discussed in Section 5.4, at this stage of development C_{mon} depends on both the monitoring period T_{mon} and the number of potential task preemptions, which is a function of the relative priorities and periods of the tasks. In our case study, for simplicity, a reasonable estimate of C_{mon} is around 60,000ns. For example, this value can be obtained by applying the computations described in Section 5.4 to the tasks in Table 2 and taking the mean value across the tasks.

Figure 10a shows the probability (p) to find a schedulable task set with respect to U for 14 tasks and C_{mon}=60,000ns. For each value of utilization U within 0.05 and 1.0 (by step 0.05) we generate 100 task sets according to [10], beforehand. We then run the schedulability test (Alg. 1) with our feasibility checker: p is the ratio between the number of feasible task sets divided by 100. For the dotted time series (i.e., *no tolerance*) WCETs are not modified, while for the solid time series (i.e., *with tolerance*) WCETs are augmented by C_{mon} before assessing the schedulability. As shown in Figure 10a, p decreases as U increases both *with*/*without* tolerance. We observe that the tolerance does not affect p up to U=0.5, while its impact is significant between 0.6 and 0.9. For example, it can be noted that with *no tolerance*, p approaches 0 at U = 0.9, while *with tolerance* is around 0 at U = 0.75. Noteworthy, the distance between *no*/*with tolerance* series depends on n: for example, the series are closer with n=11 – i.e., Figure 10d – at a similar distribution of the tasks across the containers, i.e., 2, 6, and 3 tasks, to the *high*, *medium*, and *low* criticality container, respectively.

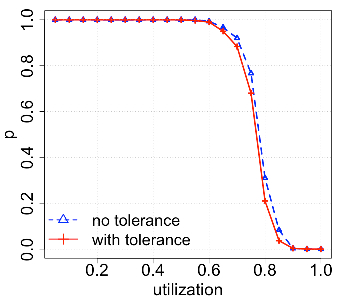
We discuss two workarounds that allow mitigating the loss of utilization. The former, reducing T_{mon}. Although we used 8,000ns in the case study, T_{mon} can be safely set to 4,000ns, which is larger than the WCET of the current algorithm implementation. In



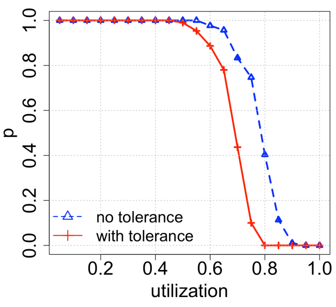
(a) n=14; $C_{mon} = 60,000ns$



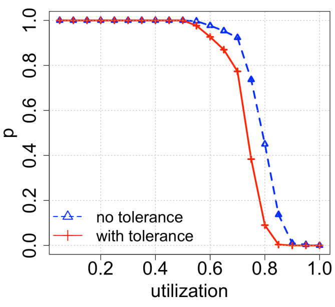
(b) n=14; $C_{mon} = 30,000ns$



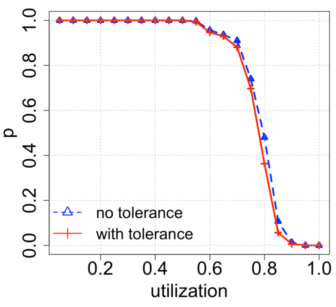
(c) n=14; $C_{mon} = 8,000ns$



(d) n=11; $C_{mon} = 60,000ns$



(e) n=11; $C_{mon} = 30,000ns$



(f) n=11; $C_{mon} = 8,000ns$

Figure 10 Sensitivity of p with respect to the utilization.

this case we obtain C_{mon}=30,000ns. Figure 10b and 10e show how p varies *with/without* tolerance at this lower C_{mon}, and denote a significant improvement with respect to the figures discussed before. The latter, improving the implementation of the monitoring algorithm. According to the discussion in Section 5.4, we are pursuing an implementation that reduces the measurement error on the execution time; at each monitoring activation we plan to leverage the *last_switch_time* of the current rt-task to properly measure the execution time of the previous monitored rt-task. Hence, in the new implementation, C_{mon} would be equal to T_{mon}, which is strongly desirable in practice. In fact, according to the sensitivity analysis presented in Figure 10c and 10f with C_{mon}=8,000ns, *no/with tolerance* series almost overlap. Based on this finding, we are confident that the prospected improvement of the implementation should address the current drawbacks in tolerating the latency.

7 Conclusions

This paper presented the notion and an implementation of real-time containers, or rt-cases, as a possible lightweight solution to let mixed-criticality hard real-time task sets cohabit on the same hardware. Containers are largely adopted in the software industry to modularize application components, to streamline the management of dependencies, and to simplify their deployment and migration in heterogeneous server environments. With rt-cases, we aim to bring these advantages and software management attitudes to real-time mixed criticality systems, allowing for the first time to run hard real-time tasks, from within containers, on low latency Linux systems patched with real-time co-kernels, such as RTAI.

The paper demonstrated the practical feasibility of the rt-case concept in a real context. The main enabling components of the proposed architecture have been implemented in a Linux environment, made publicly available and tested under realistic and feasible task sets. Preemptive fixed priority scheduling has been chosen as underlying task scheduling solution, to avoid modifications to the co-kernel support, thus fostering the early adoption of the solution. As drawbacks, we have observed that the measurement error committed by the monitor can affect significantly the feasibility of the task set when increasing the CPU utilization. In addition, we noted that the priority assignment algorithm currently implemented in the feasibility checker may not respect the relative ordering of task priorities, originally planned by the developer. To mitigate these problems, future work will be devoted to the implementation of the improved version of the monitoring algorithm that reduces the measurement error, as described in section 5.4, and to the development of the design alternative based on the use of task groups and hierarchical scheduling with server-based approaches, which makes priority assignments within rt-cases independent from the feasibility checker. We also plan to test the solution with realistic workloads in the context of the ITER project [26], thanks to an on-going research collaboration.

References

1 L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pages 4–13, December 1998. doi:10.1109/REAL.1998.739726.

2 N.C. Audsley. Optimal Priority Assignment And Feasibility Of Static Priority Tasks With Arbitrary Start Times. *Technical report YCS 164*, 1991.

3 Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM. doi:10.1145/945445.945462.

4 Alan Burns and Robert 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.

5 Maxime Chéramy, Pierre-Emmanuel Hladik, and Anne-Marie Déplanche. SimSo: A Simulation Tool to Evaluate Real-Time Multiprocessor Scheduling Algorithms. In *Proc. of the 5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, WATERS, 2014.

6 M. Cinque and D. Cotroneo. Towards Lightweight Temporal and Fault Isolation in Mixed-Criticality Systems with Real-Time Containers. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 59–60, June 2018. doi:10.1109/DSN-W.2018.00029.

7 M. Cinque and G. De Tommasi. Work-in-Progress: Real-Time Containers for Large-Scale Mixed-Criticality Systems. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 369–371, December 2017. doi:10.1109/RTSS.2017.00046.

8 N. T. Dantam, D. M. Lofaro, A. Hereid, P. Y. Oh, A. D. Ames, and M. Stilman. The Ach Library: A New Framework for Real-Time Communication. *IEEE Robotics Automation Magazine*, 22(1):76–85, March 2015. doi:10.1109/MRA.2014.2356937.

9 Z. Deng and J. W. . Liu. Scheduling real-time applications in an open environment. In *Proceedings Real-Time Systems Symposium*, pages 308–319, December 1997. doi:10.1109/REAL.1997.641292.

10 P. Emberson, R. Stafford, and R.I. Davis. Techniques For The Synthesis Of Multiprocessor Tasksets. In *WATERS workshop at the Euromicro Conference on Real-Time Systems*, pages 6–11, July 2010.

11 G. Farrall, C. Stellwag, J. Diemer, and R. Ernst. Hardware and software support for mixed-criticality multicore systems. In *Proc. of the Conference on Design, Automation and Test in Europe, WICERT, DATE*, 2013.

12 G. 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, Article ID 945850*, 2014. doi:10.1155/2014/945850.

13 M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, January 1986. doi:10.1093/comjnl/29.5.390.

14 R. Kaiser. The PikeOS concept history and design. *Technical Report, SYSGO*, 2007.

15 K. Lakshmanan, D. d. Niz, R. Rajkumar, and G. Moreno. Resource Allocation in Distributed Mixed-Criticality Cyber-Physical Systems. In *2010 IEEE 30th International Conference on Distributed Computing Systems*, pages 169–178, June 2010. doi:10.1109/ICDCS.2010.91.

16 Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. Deadline scheduling in the Linux kernel. *Software: Practice and Experience*, 46(6):821–839, 2016. doi:10.1002/spe.2335.

17 G. Lipari and S. Baruah. A hierarchical extension to the constant bandwidth server framework. In *Proceedings Seventh IEEE Real-Time Technology and Applications Symposium*, pages 26–35, May 2001. doi:10.1109/RTTAS.2001.929863.

18 C. Mao, M. Huang, S. Padhy, S. Wang, W. Chung, Y. Chung, and C. Hsu. Minimizing Latency of Real-Time Container Cloud for Software Radio Access Networks. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 611–616, November 2015. doi:10.1109/CloudCom.2015.67.

19 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.

20 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.

21 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.

22 R. Santos, S. Venkataraman, A. Das, and A. Kumar. Criticality-aware scrubbing mechanism for SRAM-based FPGAs. *Technical report, Nanyang Technological University, Singapore*, 2014.

23 Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for Hard-Real-Time systems. *Real-Time Systems*, 1(1):27–60, June 1989. doi:10.1007/BF02341920.

24 X. Wang, Z. Li, and W. M. Wonham. Optimal Priority-Free Conditionally-Preemptive Real-Time Scheduling of Periodic Tasks Based on DES Supervisory Control. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 47(7):1082–1098, July 2017. doi:10.1109/TSMC.2016.2531681.

25 WindRiver. VxWorks Virtualization Profile. http://www.windriver.com/products/vxworks/technology-profiles/#virtualization. [Online; accessed 15-Jan-2019].

26 A. Winter, P. Makijarvi, S. Simrock, J.A. Snipes, A. Wallander, and L. Zabeo. Towards the conceptual design of the ITER real-time plasma control system. *Fusion Engineering and Design*, 89(3):267–272, 2014. doi:10.1016/j.fusengdes.2014.02.064.

27 S. Xi, C. Li, C. Lu, C. D. Gill, M. Xu, L. T. X. Phan, I. Lee, and O. Sokolsky. RT-Open Stack: CPU Resource Management for Real-Time Cloud Computing. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 179–186, June 2015. doi:10.1109/CLOUD.2015.33.

28 S. Xi, M. Xu, C. Lu, L. T. X. Phan, C. Gill, O. Sokolsky, and I. Lee. Real-time multi-core virtual machine scheduling in Xen. In *2014 International Conference on Embedded Software (EMSOFT)*, pages 1–10, October 2014. doi:10.1145/2656045.2656061.

29 Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. RT-Xen: Towards Real-time Hypervisor Scheduling in Xen. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 39–48, New York, NY, USA, 2011. ACM. doi:10.1145/2038642.2038651.