ReTiF: A Declarative Real-Time Scheduling Framework for POSIX Systems

Gabriele Serra[a,*], Gabriele Ara[a], Pietro Fara[a], Tommaso Cucinotta[a]

[a]*Scuola Superiore Sant'Anna, Via Moruzzi, 1, 56124 Pisa, Italy*

Abstract

This paper proposes a novel framework providing a *declarative interface* to access real-time process scheduling services available in an operating system kernel. The main idea is to let applications declare their temporal requirements or characteristics without knowing exactly which underlying scheduling algorithms are offered by the system. The proposed framework can adequately handle such a set of heterogeneous requirements configuring the platform and partitioning the requests among the available multitude of cores, so to exploit the various scheduling disciplines that are available in the kernel, matching application requirements in the best possible way. The framework is realized with a modular architecture in which different plugins handle independently certain real-time scheduling features. The architecture is designed to make its behavior customization easier and enhance the support for other operating systems by introducing and configuring additional plugins.

Keywords: Real-Time Systems, Scheduling, Programming Model, Linux

1. Introduction

In the past decade, the global interest running real-time applications in distributed or embedded systems rose considerably. Real-time applications however are not limited to specialized systems anymore: multimedia applications like audio/video processing and streaming, gaming, etc. are notable examples of applications with soft or firm real-time constraints that run on General Purpose Operating Systems (GPOSes). To support these applications, modern GPOSes evolved to provide a set of features that allow the coexistence of both real-time and non real-time applications on the same host.

Among these GPOSes, Linux is a common choice for applications that have real-time requirements, thanks to its rich support for multimedia peripherals,

*Corresponding author.
Email addresses: gabriele.serra@santannapisa.it (Gabriele Serra), gabriele.ara@santannapisa.it (Gabriele Ara), pietro.fara@santannapisa.it (Pietro Fara), tommaso.cucinotta@santannapisa.it (Tommaso Cucinotta)

the plethora of libraries and tools readily available for media processing, and the active support of a vast community of open-source developers. In addition, the Android operating system (OS), based on Linux, has become a popular choice for a number of embedded systems for multimedia services, from smartphones and tablets to infotainment systems deployed in modern cars. However, Linux is not the only choice when it comes to systems that provide support for real-time applications; another notable example is FreeBSD, which is also the base for some OSes running on many modern game consoles nowadays.

Similarly to any other GPOS, the Linux kernel development focused on minimizing average OS overheads and optimizing in-kernel operations to maximize the performance of user-space applications, while keeping a good responsiveness for interactive workloads, notably user interactions and multimedia applications. However, Linux has also been consistently improving its support for real-time workloads over the past decade, providing a growing set of features targeting real-time applications [1, 2]: the inclusion of POSIX real-time extensions [3] and the support for real-time mutexes; high-resolution timers with nano-second precision; the removal of the Big Kernel Lock (BKL)[1]; enhancements to the kernel preemptibility options; the introduction of `NO_HZ` for reducing overheads of the periodic bookkeeping timer; the `PREEMPT_RT` [2, 4, 5] variant that reduces worst-case scheduling latencies by running device drivers as kernel threads that can be scheduled and turning most of the spinlocks into mutexes; and the addition of the `SCHED_DEADLINE` process scheduler [6], implementing a global Earliest Deadline First (EDF) algorithm (that can also be configured as partitioned or clustered EDF) that uses a multi-processor variant of the Constant Bandwidth Server (CBS) [7] algorithm to provide temporal isolation among tasks. In addition, a number of frameworks and middlewares have been developed to further enhance the capabilities of Linux as a powerful development platform for real-time applications. These features increased the relevance of Linux as a suitable platform to develop soft real-time applications, even with respect to other modern GPOSes.

In general, GPOSes may be required to host a plethora of different applications, each characterized by their own temporal characteristics and real-time requirements; these may range from interactive applications, to multimedia, to gaming and virtual-reality tools, to real-time control applications for factory automation. These applications may activate periodically or sporadically, they may require access to real-time scheduling priorities, or sometimes their timing requirements are unknown a priori and they should be inferred by comparing their periodicity to other co-located applications. Finally, these systems may need to host simultaneously both real-time and non real-time applications. In a true *component-based approach* for realizing complex real-time systems, it is all but trivial to understand how to let all of these applications coexist on the same system, exploiting the different schedulers that are available, and how to configure them for an optimal use of an underlying multi-core platform.

[1]For more info see `https://kernelnewbies.org/BigKernelLock`

1.1. Contributions

In this work, we provide an overview of the *Real-Time Framework (ReTiF)*[2], aimed at providing *controlled access to real-time CPU scheduling features to unprivileged applications on POSIX-compliant GPOSes.* The goal of this framework is to improve the usability of existing real-time capabilities of various OSes by providing a unified and portable API; this new middleware can be used to declare temporal characteristics of real-time applications, independently of the particular scheduling policy that will be selected to satisfy the declared requirements. This new *declarative* approach allows applications characterized by heterogeneous requirements to coexist on the same host and use the same API independently of the underlying OS, improving their portability across POSIX-compliant GPOSes.

To achieve this goal, *Real-Time Framework (ReTiF)* adopts a modular approach to scheduling, in which a set of plugins are used to translate from the generic attributes declared by each application to the proper configurations of the real-time features exposed by the underlying kernel. To avoid unwanted consequences of unrestricted access to real-time scheduling features of the operating system, we present here for the first time the access control model that we implemented for the framework, providing system administrators a comprehensive mechanism to manage access to real-time resources on target machines. With this new tool, every aspect of the framework can be configured and each request is checked against the policies defined by the system administrator, which maintains total control on the behavior of the managed system. With *ReTiF*, real-time applications can be associated with one of multiple scheduling policies active at the same time on multi-core platforms with ease.

This modularity is the key to offer support to a plethora of scheduling paradigms on the same OS (e.g., rate monotonic is used on some CPUs, while `SCHED_DEADLINE` reservations is used on others), but also to provide portability of real-time applications across multiple OSes, by developing platform-specific implementations of certain plugins.

This work constitutes an extended version of the paper already appeared in [8], where: 1) we provide a more comprehensive description of *ReTiF* in Section 3, including its internals and the interactions among its main components; 2) we present for the first time the security model that we recently added to *ReTiF* in Section 4, which is of utmost importance when running unprivileged real-time applications on shared systems; 3) we present a more detailed and comprehensive discussion of the related research in Section 2; 4) we discuss current limitations of the proposed framework and how we plan to address them in future revisions in Section 7; and 5) we show a novel extensive evaluation of the overheads introduced by the framework with respect to directly accessing real-time features of the operating system in Section 6.

[2]The software is freely available on GitHub, under a GPLv3 license, at: `https://github.com/gabriserra/retif`.

1.2. Paper organization

This paper is organized as follows. After discussing related research in Section 2, an overview of *ReTiF* is presented in Section 3, focusing on the main decisions driving its design, followed by the description of its access control model in Section 4. Some implementation details are provided in Section 5, while experimental data highlighting its performance and overheads are presented in Section 6. Finally, limitations of this approach are thoroughly addressed in Section 7 and conclusions are discussed in Section 8, along with possible directions for future research on the topic.

2. Related Work

This section briefly reviews related works appeared in the scientific literature about mechanisms and middleware layers supporting real-time applications on GPOSes, with a particular focus on operative systems based on the Linux kernel and its extensions.

Support for real-time applications in a GPOS like Linux was first introduced by running the system on top of a micro-kernel layer placed between the hardware and the kernel itself, acting like a hypervisor. In this approach, real-time and "normal" tasks were treated very differently, with the former being handled by the real-time micro-kernel layer and the latter scheduled at lower priority by the Linux scheduler. The most important implementations of this paradigm have been RT-Linux, proposed by Yodaiken et al. [9], and RTAI, proposed by Mantegazza et al. [10]. The latter has also been later forked by Gerum et al. into another project called Xenomai [11]. Both solutions require applications to use custom APIs and heavy modifications to the Linux kernel. Therefore, these solutions are not suitable for certain applications that should run in user-space context as unprivileged processes, as in the case of audio/video processing applications with soft real-time requirements. The ARINC-653 specification [12] uses a similar approach, implementing a kernel-level partitioning mechanism for Linux. As required by the avionic specifications, this implementation provides a high level of isolation among applications, but it cannot be easily adapted for other application fields.

Many real-time kernel extensions and middleware solutions have been proposed to support at the same time hard and soft real-time applications on Linux or other GPOSes by directly extending or patching the system scheduler. However, most implementations limit their support to fixed priority (FP) scheduling without providing any temporal isolation among real-time tasks and creating a potential disruption of the guarantees offered to tasks executing at lower priorities. These approaches are suitable for hard real-time tasks with strict requirements, but find little application in the multimedia field. A representative example of these approaches is KURT Linux [13], which consists in a significant modification of the Linux internal scheduling mechanisms. KURT introduces 3 distinct operational modes: in *normal mode* the system behaves like a GPOS; in *real-time mode* only real-time processes can run while normal processes are blocked; finally, in *mixed mode* both real-time and non real-time applications can be executed concurrently. In particular, in mixed mode normal processes can run only in the slack time left after scheduling all real-time applications; this guarantees that real-time applications have higher priority with respect to normal processes.

A similar concept was pursued by the OCERA EU project[3], where RT-Linux was integrated with custom modifications to the Linux kernel, so to achieve the coexistence of hard and soft real-time applications [14]. Hard real-time processes could rely on a POSIX API adapter to the RT-Linux functionality. Soft real-time processes could use a custom API to interact with a set of dynamically loadable modules realizing a variant [15] of the CBS algorithm to provide a reservation-based scheduler exploiting a minimally invasive patch to the Linux kernel.

In [16] authors present QRAM, an analytical model that uses quality of service (QoS) to allocate multiple resources to real-time applications, with the main goal of maximizing an overall QoS cost function for the entire system. This approach was also later extended [17] with an adaptive on-line optimization policy. Other works based on similar techniques are [18], in which a QoS middleware mediates application access to physical resources to support dynamic workloads, and Linux-SRT [19], which enhances Linux to provide predictable scheduling and QoS management tools. Another similar approach, called Firm-RT, was presented by Srinivasan et al. [20] to support firm real-time applications in which both soft real-time and time-sharing applications can run concurrently on the same system. This last approach consists in a set of modifications to the Linux kernel that provide support for the stringent timing requirements of these applications.

In more recent years, many Linux-specific libraries have been developed to enhance the capability of the Linux kernel to support real-time applications, especially without super-user privileges. A notable example is the RealtimeKit (RTKit) library[4], included as a dependency of the PulseAudio sound infrastructure for POSIX OSes[5]. RTKit is a D-Bus system service that can change the scheduling policy of user processes or threads on request. However, the RTKit daemon is only intended to be used to let user processes access the `SCHED_RR` scheduling policy. A similar mechanism is provided in the nowadays Linux kernel by the `prlimit()` system call and the related `limits.conf` configuration file.

LITMUS^{RT} [21] is a framework composed of a kernel patch and a related user-space interface that allows applications to schedule real-time tasks using a wide variety of schedulers. LITMUS^{RT} can be extended with a plugin mechanism that lets system designers write their own plugins, in which they can implement new scheduling algorithms. The main goal of LITMUS^{RT} is to provide the research community with a test-bench for real-time scheduling algorithms

[3]More information at: `http://www.ocera.net`.
[4]For more info see `https://github.com/heftig/rtkit`
[5]For more info see `https://www.freedesktop.org/wiki/Software/PulseAudio/`

on a real Linux-based platform. While in this sense LITMUS^{RT} can be successfully used to investigate the behavior of novel scheduling algorithms, it does not support multiple real-time scheduling algorithms at the same time.

Conversely, due to its nature, a GPOS should be capable of hosting a variety of applications with heterogeneous temporal characteristics, offering a scheduling layer that exposes a common interface for expressing application requirements. An effective way of providing both temporal requirements declaration and isolation in a GPOS is the combination of QoS contracts and resource reservation techniques. Several algorithms and implementations have been proposed in the literature along this direction. Aldea et al. proposed the Flexible Integrated Real-time Scheduling Technologies (FIRST) architecture [22], which is an OS-independent specification that organizes a number of scheduling algorithms to work in cooperation, including both FP and dynamic priority (DP) scheduling algorithms, in a hierarchical scheduling architecture. The resulting architecture allows system designers to build complex real-time systems by simply specifying the real-time requirements of the desired applications. FIRST also relies on reservation techniques to provide temporal isolation among real-time tasks: real-time applications can leverage this framework to establish QoS contracts with the system, which will then provide them a set of guarantees. In addition, FIRST organizes multiple scheduling algorithms, including both FP and EDF, to work in cooperation by assigning each scheduler a server with its own temporal budgets. In principle, any system can implement the FIRST Scheduling Framework (FSF) to provide multi-scheduler support with the guarantees included in the FIRST specification. FSF has been implemented in SHARK [23] and MaRTE [24], but not in other GPOSes like Linux.

FRSH/FORB [25] is another middleware based on CORBA that provides reservation scheduling across several physical resources, such as CPUs, disks and network interfaces, to real-time applications. This mechanism is made available through kernel-level extensions to the Linux OS, one of them provided by the AQuoSA architecture [26] supporting adaptive CPU reservations, and real-time extensions [27] for wireless communications compatible with the IEEE 802.11 standard series. FRSH/FORB has also been extended [28] with a transactional API for handling multi-resource reservations in a distributed system.

The ExSched project [29] tries to support different OSes with a plugin-based scheduler design. This framework is made of a kernel module and a set of plugins that can be chosen by the system administrator. The ExSched framework aims to provide a unified scheduler interface that can be leveraged to implement different schedulers without patching the underlying OS. However, this feature comes with a considerable cost in terms of performance: for example, the EDF scheduler plugin implementation introduces a huge overhead (about 180% in the worst case) on the system compared to `SCHED_DEADLINE` implementation on Linux [29]. Also, applications must be aware of their exact timing parameters like task period, worst case execution time, etc. to be effectively used with ExSched. Similar considerations apply for a noteworthy attempt [30] of realizing a reservation-based scheduler in the MINIX3 micro-kernel [31] as an external module that can be loaded at run-time.

Another solution proposed by Parmer and West [32] called Hijack is composed of a kernel module and an interposed execution environment between the process address spaces and the kernel. Hijack introduces some mechanisms to intercept system calls and interrupts via a kernel module, which in turn forwards control of the request to a user-level daemon. The daemon will elaborate the request and the response is finally provided to the original application traversing this stack in the opposite direction of the request.

Other frameworks target commercial operating systems. For example, Benham et al. presented HSF-VxWorks [33], which introduces support for Rate Monotonic (RM) and EDF hierarchical scheduling to VxWorks OS without modifying its kernel. Another example is HSF-FreeRTOS, proposed by Inam et al. [34], which implements a hierarchical scheduling framework for FreeRTOS, providing support to temporal isolation among applications running on a single processor.

In 2016, Wei et al. [35] proposed RT-ROS, a real-time ROS architecture that provides an integrated task execution environment that is able to run real-time and non-real-time tasks in the same system. The real-time tasks run on top of a real-time OS while non-real-time ones run on Linux. The real-time OS and Linux run on different processor cores, albeit with limited separation in terms of security.

Another work based on ROS was introduced in 2018 by Saito et al. [36]: they presented a real-time scheduling framework, called ROSCH, that added real-time features to ROS such as fail-safe functionality, fixed-priority based directed acyclic graph (DAG), and a synchronization system.

A solution for integrating real-time and non-real-time environments for cloud computing was presented in [37]. Here the authors designed RT-Open Stack, a cloud CPU resource management system for deploying real-time and non-real-time virtual machines. Based on a real-time hypervisor (RT-Xen) and on the Open Stack cloud infrastructure, it allows to execute both real-time and non-real-time virtual machines in the same host.

The same authors of this paper presented a framework [8] that supports heterogeneous sets of applications with different real-time characteristics by allowing applications to declare timing requirements, in an agnostic fashion with respect to the underlying available scheduling policies. This is in clear contrast with the other solutions described above, which support either FP or EDF/CBS scheduling or that require real-time applications to know which scheduling policies are provided by the underlying system and the relative parameters. In addition, this novel framework is entirely implemented in user space and requires no modification to the underlying OS: it is composed by a simple daemon running with root privileges and an application library that is used by real-time applications to interact with the daemon itself. The architecture of that framework is inspired from a prior preliminary workshop paper [38] which, to the best of our knowledge, was never actually implemented before.

This paper constitutes an extension of our prior work [8] just described above. In this paper, we highlight the features of that framework that characterize its portability across POSIX-compatible OSes and we introduce a new

Table 1: Comparison among real-time frameworks available for Linux.

Framework	**Multi-core**	**Kernel Modification**	**Portability**	**Sched. Alg.**
RT-Linux	No	Patch	Linux	FP
RTAI	Yes	Patch	Linux	FP-FIFO, RR, EDF
Xenomai	Yes	Patch	Linux	FP, RR
KURT Linux	Yes	Patch	Linux	FP (SRMS)
HSF-VxWorks	No	-	VxWorks	FP, EDF
HSF-FreeRTOS	No	-	FreeRTOS	FP
AQuoSA	No	Patch + Kernel module	Linux	CBS-EDF
Firm-RT	No	Patch	Linux	FP (SRMS)
LITMUS^{RT}	Yes	Patch + Kernel module	Linux	Module dependent
Linux-SRT	Yes	Patch + Kernel module	Linux	FP
OCERA	No	Patch	Linux	CBS-EDF
FSF	No	-	MaRTE, SHARK	Plugin dependent
Hijack	No	Kernel module	Linux	FP-RR
ExSched	Yes	Kernel module	Linux	FP, EDF
ReTiF	**Yes**	**No**[a]	**POSIX**[b]	**Plugin dependent**

a. For more details, refer to Section 3.4.

b. For more details, refer to Section 7.

access control mechanism that provides an additional level of security when accessing the real-time features provided by this framework. With this new extension, system administrators can now specify a set of constraints that can be used to ensure certain properties for the system, especially when multiple unprivileged users share the same machine.

Table 1 shows a recap of the main characteristics of each real-time framework described in this section with respect to the proposed one, which is listed as the last entry. In the table, the column "Requires System Modification" indicates whether each framework requires the target system to be either patched or to load custom kernel models, the "Multi-core" column indicates whether each framework supports scheduling of tasks across multiple cores, while the "Portability" column indicates whether each framework can be ported to other operating systems with little to no modifications. The last column, "Scheduling Algorithm", shows the supported scheduling algorithms by the corresponding work. For the sake of clarity, the algorithm names were abbreviated as follows: FP for Fixed Priority, FP-FIFO for Fixed Priority with First-In-First-Out policy for tasks with the same priority, FP-RR for Fixed Priority with Round-Robin policy for tasks with the same priority, EDF for Earliest Deadline First, CBS-EDF for Constant Bandwidth Server in conjunction with Earliest Deadline First, and SRMS for Statistical Rate Monotonic Scheduling. Some of the listed works can be extended with modules or plugins implementing additional scheduling algorithms.

3. Real-Time Framework (ReTiF)

In this section we present *Real-Time Framework (ReTiF)*, a software framework that we realized to ease access to real-time scheduling policies on Linux in a declarative fashion. The main focus of this project is to provide an abstraction level to real-time applications that enables them to declare a set of real-time task and their scheduling parameters. From these parameters, the framework takes care of selecting the most proper scheduling technique and configuring the actual scheduling parameters of the actual process/thread associated to each task specification, by interacting with the underlying OS.

This tool aims to meet at user-space level the requirements posed by complex real-time and multimedia applications by exposing a simple yet expressive interface. *ReTiF* allows system administrators to let unprivileged users the capability to run applications with real-time constraints, without worrying about the underlying features exposed by the OS; most of its implementation is entirely OS-agnostic, relying solely on POSIX-compliant features. All non-standard interactions between *ReTiF* and the underlying OS is limited to a set of plugins, which can be chosen at deployment time. For this reason, the framework itself does not require any modification of the underlying OS kernel, unless one of the selected plugins explicitly requires certain features.

To accomplish these goals, *ReTiF* architecture is organized in two main components: a *shared library* that applications can use to declare their requirements and a *daemon* running with superuser privileges. In this design, the daemon is the central authority that is in charge of managing all real-time applications in the system; unprivileged processes will declare their real-time requirements using the API exposed by the shared library and the daemon will set all their scheduling parameters accordingly. To avoid unwanted consequences of unregulated access to real-time features of the system from unprivileged users, *ReTiF* also implements a flexible access control model that system administrators can use to manage the amount of resources accessible to applications. More details on this model are provided in Section 4.

The API exposed by the shared library is purposely designed to be independent from the scheduling algorithms or policies that are used to meet the demands of each application. The set of parameters that can be declared by each application represents a set of typical parameters that characterizes simple independent real-time applications. It will then be the role of the daemon to select the proper scheduling policy to use to satisfy such requirements. *ReTiF* defines an API that should be implemented by scheduling services, in the form of *plugins* that will be loaded into the daemon application at deployment time. System administrators can choose which set of plugins should be loaded and what policies should be used to grant certain user/groups access to the individual plugins. Researchers and other developers can easily extend the functionality of the framework, developing their own plugins, each exposing a different algorithm or policy, all accessible with the generic user-space API. This approach is particularly useful to maintain the portability of the framework, limiting all special interactions with the underlying OS to each plugin.

Independently from the plugins loaded with the daemon at any time, applications can use the shared library to declare the parameters of real-time tasks that they would like to run. For each request, the daemon will then respond indicating whether or not it can be accepted by one of the plugins in the current condition of the system. On acceptance, a single execution flow—i.e. a POSIX process or thread—can be dynamically attached to the accepted specifications, which only represents a set of accepted real-time parameters. After this operation, the daemon instructs the selected plugin to set the actual scheduling parameters of the attached execution flow accordingly; for example, the EDF plugin (which relies on Linux `SCHED_DEADLINE` scheduler) will set up a CPU reservation when a process is attached to one of its accepted specifications. A rejected request can be re-submitted later after relaxing some of the real-time parameters or after releasing some resources in the system, by terminating other real-time applications.

Any process/thread managed by the framework can be dynamically detached from the accepted scheduling parameters; after this operation another process/thread can be attached to them or they can be released.

The real-time parameters that can be declared for each task are the following ones:

1. a period T, expressed in microseconds, which usually corresponds to the minimum inter-arrival period between consecutive task instances;
2. a runtime Q, in microseconds, which usually is equal to the worst-case execution time of each task instance;
3. a relative deadline D, that defaults to the same value as the period T, if specified;
4. a static priority P, in the range of standard real-time POSIX priorities.

The design principle that differentiates this framework from others shown in Section 2 is the *declarative* approach to real-time parameters specification. The way this framework is designed, applications can be implemented in a agnostic fashion with respect to the scheduling algorithms (each implemented by a different plugin) that may be available at runtime. For this reason, applications may declare from none to all of the real-time parameters described above for each task. The framework will then automatically match each scheduling request with the plugin that is the most suitable to handle the declared specifications. To achieve this goal, the daemon presents each request to each plugin in an ordered fashion and expects them to respond whether they can handle the request or some other plugin should. Each plugin uses a well-defined API to inspect the parameters included in each request and chooses independently from other plugins to either accept or reject the given request. For this reason, different plugins may have different requirements—e.g. some parameters may be mandatory, while others may be ignored—and they may even perform complex admission control tests before choosing whether or not accepting the new task. If at least one plugin accepts a request, the daemon chooses the most suitable among the accepting plugins and the task is added to the list of tasks in the current active task set. The requesting application can then bind the accepted scheduling parameters to its process or one of its threads.

In general, plugins can admit or reject a task based on the list of parameters included in each request. Since the admission policy may change from one plugin to another, the daemon delegates this operation to each plugin, interrogating each of them in a predetermined order to find the best plugin for each request. The admission request can simply be based on the presence/absence of certain parameters (e.g. period, static priority, etc.) or it may rely on more complex tests that take into account the current condition of the system, as represented by the tasks already accepted in the task set. In the latter situation, plugins should check for necessary conditions only—i.e. whether accepting the new task will inevitably lead the system into an unschedulable state. Tasks may specify whether they want to bypass this admission control test upon task declaration, accepting all the risks that may follow.

Accepted tasks can later change their parameters, if the change does not result in the disruption of any other accepted request. This operation is atomic—i.e. multiple parameters can be changed atomically—and there is no guarantee about which plugin will be used to schedule a task following a successful change operation—meaning that on success a different plugin may be selected to schedule the task. When a change request fails, the task will maintain the same scheduling parameters and plugin that were previously assigned by the framework. This operation can be used to dynamically request more computational resources to the system or to release them when not needed anymore.

Finally, tasks may also declare optionally two different values as their worst case execution time: in this case, Q is considered as the *minimum runtime* requested by the task, while the second value, Q^d will be a *desired runtime* (higher than Q). In this situation, each plugin is free to accept the request using any runtime in the range $[Q, Q^d]$, which will be called *accepted runtime* Q^a. Any task can query the user API to retrieve its own accepted runtime and it can use the returned value to enable/disable optional paths in the execution flow accordingly. To change the *accepted runtime*, a task can use another explicit request, which will follow the rules described above for changing other scheduling parameters.

3.1. Architecture Overview

ReTiF is composed by three main components that interact through well-defined APIs: the *ReTiF Daemon*, the *ReTiF Library*, and the set of plugins dynamically loaded on system initialization. Figure 1 shows the relationships among these components. The most important is of course the central decision authority of the system, represented by the *ReTiF Daemon*: this component is in charge of coordinating all interactions among applications and scheduling plugins loaded by the daemon itself. The *ReTiF Library* controls how each application can send requests to the daemon using the API described in Section 3.2; each request will then be forwarded to the daemon via a POSIX-compliant Inter Process Communication (IPC) mechanism, namely a UNIX socket connection. Figure 2 depicts the typical sequence of interactions between the user-process and the various components of the *ReTiF*, in response to a `rtf_spec_create` request.

Each dynamically loaded plugin represents one or more scheduling policies, allowing the framework itself to be completely agnostic with respect to both

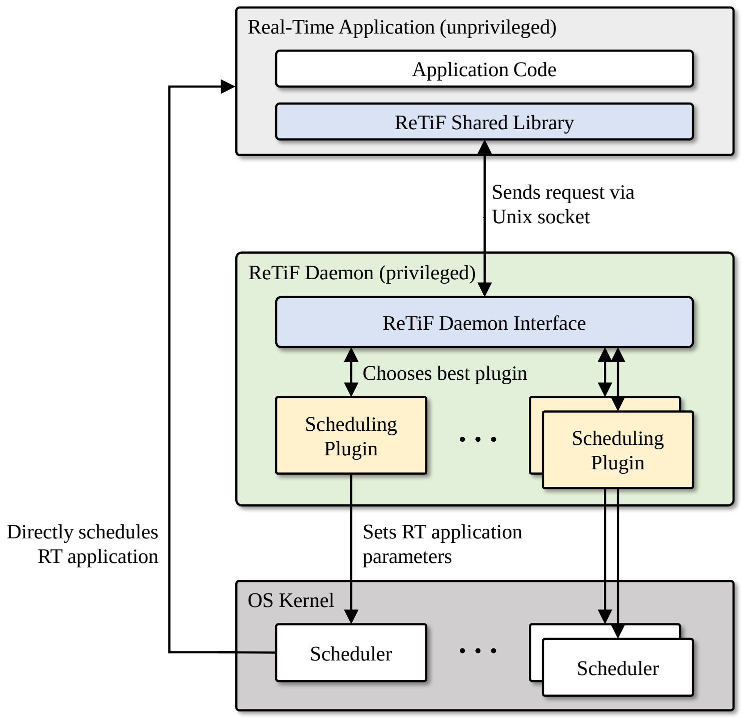


Figure 1: Architectural overview of the framework and interaction scheme among main framework components.

the scheduling model associated with each algorithm and the underlying implementation of the algorithm. In general, each plugin is supposed to analyze the current task set and check whether accepting the incoming request (which can ask to either include a new task or change the scheduling parameters of an already accepted request). Plugins are free to implement any admission criterion: acceptance can depend only on the presence/absence of certain scheduling parameters or on complete schedulability analysis of the resulting system condition. If at least one plugin accepts the incoming change, the request is successful and the priority which best fits the request criteria will be assigned in charge for the task. This choice is made taking into account a priority list that can be configured by system administrators that defines a total ordering among loaded plugins.

3.2. ReTiF Library API

The *ReTiF Library* implements a well-defined API that applications can use to leverage the real-time capabilities of the underlying OS through the *ReTiF* framework. A description of the main functions exposed by the library is included in Table 2. In addition to those functions, the library provides a set of

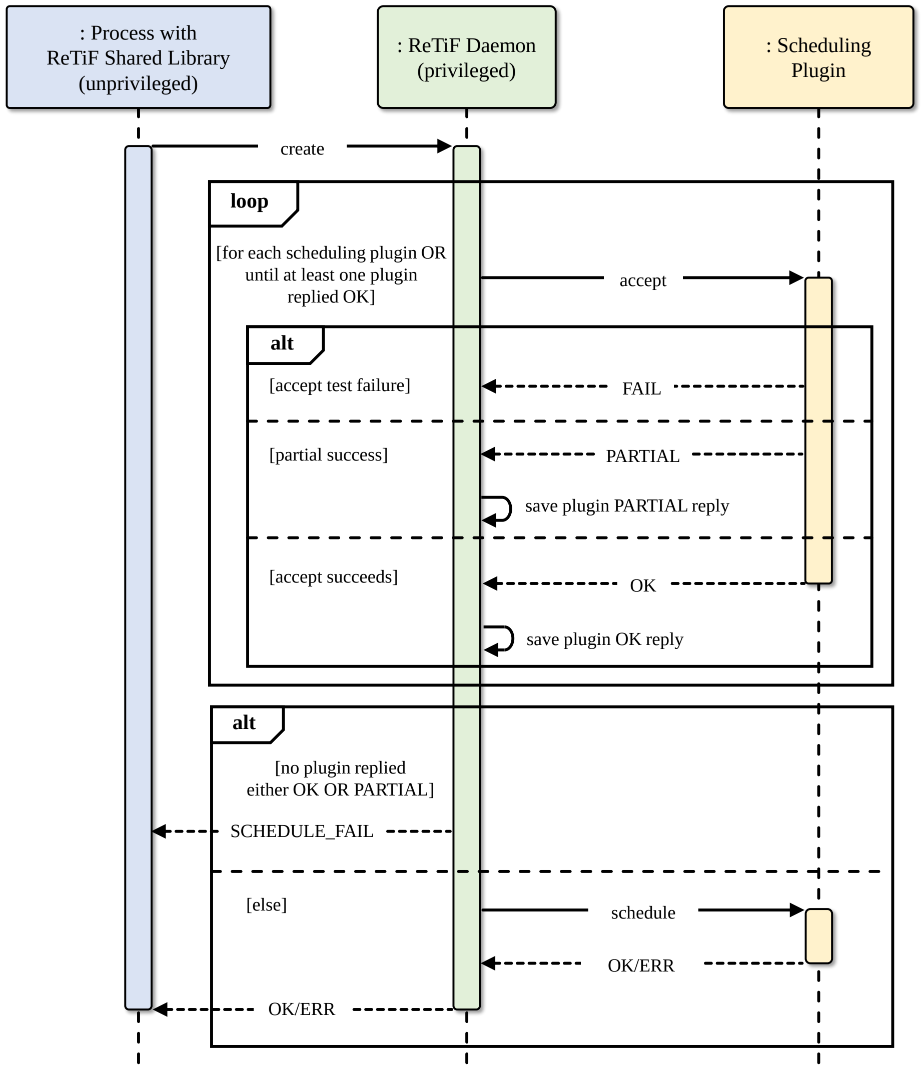

: Process with ReTiF Shared Library (unprivileged)
: ReTiF Daemon (privileged)
: Scheduling Plugin
create
loop
[for each scheduling plugin OR until at least one plugin replied OK]
accept
alt
[accept test failure]
FAIL
[partial success]
PARTIAL
save plugin PARTIAL reply
[accept succeeds]
OK
save plugin OK reply
alt
[no plugin replied either OK OR PARTIAL]
SCHEDULE_FAIL
[else]
schedule
OK/ERR
OK/ERR

Figure 2: Sequence diagram of a typical `rtf_spec_create` request from a user process. For simplicity, the scheme does not depict failing requests due to communication errors (timeout) between the application library and the daemon and omits checks performed to test compliance with the configured access control model.

Table 2: API exposed to real-time applications by the ReTiF Library.

Function	Description
`rtf_connect`	Establishes a connection to the *ReTiF Daemon* using Unix domain sockets.
`rtf_spec_create`	Creates a new task specification, specifying its parameters and requesting the *ReTiF Daemon* for its acceptance into the active task set.
`rtf_spec_change`	Requests a new task admission test using new parameters for an existing task; in case of failure, the task maintains its old parameters.
`rtf_spec_release`	Releases a task specification, freeing its resources and detaching the attached process/thread, if any.
`rtf_spec_attach`	Attaches a POSIX process/thread id to an accepted task specification.
`rtf_spec_detach`	Detaches the POSIX process/thread assigned to a task specification; from this point onward, the specified process/thread no longer runs with real-time priority and the same task specification can be re-assigned to another process/thread.

functions useful to interrogate the daemon and to implement a periodic task in C.

Each application must connect to the daemon using the `rtf_connect` function exposed by the *ReTiF Library* before submitting the first request. Then the connection is kept open for further requests.

The library provides an opaque data type called `rtf_params` that applications will fill with the scheduling parameters that they want to include in a new request using the functions described in Table 3. Once done filling the `rtf_params` object with the declared parameters, a new task admission request can be submitted to the *ReTiF Daemon* by executing

```
rtf_result_t result = rtf_spec_create(spec_ptr, params_ptr);
```

where `spec_ptr` is a pointer to a `rtf_spec` object, which is an opaque type that represents a real-time task specification, and `params_ptr` points to the just-filled parameters object.

The result of this function can be either `RTF_OK` on success, `RTF_SCHEDULER_FAIL` if is not possible to guarantee provided parameters or `RTF_ACL_FAIL` if the given request cannot be authorized. If a desired runtime Q^d was supplied with the parameters in the request, the application can query the daemon by calling

```
acc_runtime = rtf_spec_get_accepted_runtime(spec_ptr);
```

In case of failure, the request can be repeated after changing some of the parameters or waiting for the system to get into a different working condition. Accepted

Table 3: Parameters that can be declared in each task specification. Getters and setters are used to operate on `rtf_params` opaque data structures.

Parameter	Symbol	Unit	Getter/Setter
Period	T	µs	`rtf_params_get_period` `rtf_params_set_period`
Runtime	Q	µs	`rtf_params_get_runtime` `rtf_params_set_runtime`
Desired Runtime	Q^d	µs	`rtf_params_get_des_runtime` `rtf_params_set_des_runtime`
Relative Deadline	D	µs	`rtf_params_get_deadline` `rtf_params_set_deadline`
Priority	P	–	`rtf_params_get_priority` `rtf_params_set_priority`
Desired Scheduling Plugin		–	`rtf_params_set_scheduler` `rtf_params_get_scheduler`
Ignore Admission Test		–	`rtf_params_ignore_admission`

specifications can later change their parameters using the `rtf_spec_change` function, which responds in the same way as the `rtf_spec_create` does.

The list of mandatory/optional parameters is entirely dependent on the list of plugins loaded at runtime by the daemon and to the policy that applies to the current user of the application (see Section 4). Usually, the choice of the best plugin that will be selected to handle a task is made inspecting the list of supplied parameters and the limitations in the access control policy. However, applications can indicate when a specific plugin shall be used to schedule a specific task. In that case, only the requested plugin will be interrogated for task admission.

Each accepted task specification can be associated with an execution flow using the `rtf_spec_attach` call. On successful completion of this call, the user application is ensured that the process or thread identified by the supplied ID will be scheduled according to the accepted scheduling parameters, hence running as a real-time task. To accomplish this goal, the daemon will forward the attached request to the previously selected plugin for that task, which will interact with the underlying OS and set the scheduling policy of the given process/thread accordingly.

Listing 1 shows the body of a sample process that uses the *ReTiF Library* API to set its own scheduling parameters and run as a periodic task. The code shows also how some utility functions provided by the library to ease the implementation of periodic tasks on POSIX systems, when a `period` parameter is declared by the application. In particular, the `rtf_task_start` call marks the beginning of the first period of execution for the real-time task, while the `rtf_task_wait_period` call can be repeatedly invoked to suspend task execution waiting for the next activation point.

Listing 1: Body of a real-time process that uses the framework.

```c
/* Task representation */
struct rtf_spec s = RTF_SPEC_INIT;

/* Task parameters */
struct rtf_params p = RTF_PARAM_INIT;

/* Connect to the daemon via a UNIX socket */
if (rtf_connect() == RTF_CONNECTION_ERR)
  return; /* Unable to connect to the daemon. */

/* Set task parameters */
rtf_param_set_period(&p, T_PERIOD);
rtf_param_set_runtime(&p, T_RUNTIME);
rtf_param_set_des_runtime(&p, T_DES_RUNTIME);
rtf_param_set_deadline(&p, T_DEADLINE);

/* Test for admission */
int res = rtf_spec_create(&s, &p);

if (res == RTF_SCHEDULE_FAIL)
  return; /* Admission failed, we can retry with different parameters */
else if (res == RTF_ACL_FAIL)
  return; /* ACL check failed, not authorized */
else if (res == RTF_CONNECTION_ERR)
  return; /* Communication failed */

/* res = RTF_OK */

/* On success we attach an execution flow to the task specification */
rtf_spec_attach(&s, getpid());

/* Signals that a task begins its execution */
rtf_task_start(&s);

while(!computation_ended()) {
  /* Task runs mandatory actions */
  mandatory_computation();

  /* Enabling optional computation depending on the accepted runtime */
  if (rtf_spec_get_accepted_runtime(&s) > T_RUNTIME)
    optional_computation();

  /* Suspend execution waiting for the next period */
  rtf_task_wait_period(&s);
}

/* Cleanup */
rtf_spec_release(&s);
```

3.3. ReTiF Daemon

The *ReTiF Daemon* is the central component in charge of managing all user requests and forwarding them in an orderly fashion to each plugin. It also implements the security mechanisms described in Section 4 by inspecting each request and taking the required actions in order to meet the access control specification provided by the system administrator. Notice that the *ReTiF Daemon* is OS agnostic and as such it does not interact with the underlying OS, except for the POSIX-compliant mechanisms used for communication with the *ReTiF Library* and security checks.

It is instead the role of each plugin to implement a scheduling policy on top of the features exposed by the underlying OS. For this reason, some plugins may be compatible with multiple systems, while others may require a specific OS or even a certain OS module to be loaded on the target machine.

The list of all the plugins that shall be loaded by the daemon during its initialization is provided by the system administrator via a simple configuration file: the file lists all the names of the `.so` files to be loaded, each associated with a custom name, a range of POSIX priorities that can be used by the plugin and the CPU cores managed by the plugin (see Section 3.5). The custom name associated with each line is used to load each plugin under different logical names, each with a different access control policy associated. More details about this mechanism are provied in Section 4.

Each plugin must implement the API illustrated in Section 3.4. Those functions are called by the daemon to interrogate each plugin upon receiving a new request from the library, for example when a new task specification is declared. Each plugin analyzes the parameters associated with each request and responds whether it is capable to satisfy those parameters entirely (`RTF_OK`), it can satisfy the request even if some recommended parameters are missing (`RTF_PARTIAL`), or it cannot satisfy the request at all (`RTF_NO`), either because some mandatory parameters are missing or because some necessary admission test resulted in a failure.

For each request, the daemon collects the responses of each plugin and selects the one with the highest POSIX priority (specified by the system administration via the configuration file, see Section 3.5) that replied `RTF_OK`; if no plugin is found using this criterion, the daemon will proceed selecting the one with the highest priority that replied `RTF_PARTIAL`. If all plugins replied `RTF_NO`, the request is denied and no changes to current system configuration is applied. Finally, a response is sent back to the requesting application.

3.4. ReTiF Plugins API

Each plugin implements a single real-time scheduling policy (which may correspond to a specific real-time scheduling algorithm or multiple ones, depending on the plugin implementation), which will leverage the real-time functionality exposed by the underlying OS to schedule real-time tasks. To do so, each plugin must implement the functions described in Table 4, which regulate the interactions between the daemon and the plugin itself.

Table 4: Main functions belonging to the ReTiF Plugins API. Each of these functions is invoked by the *ReTiF Daemon* to interact with each scheduling plugin.

Function	**Description**
`rtf_plg_accept`	The plugin is inquired to perform the admission test for a new task using supplied parameters. Plugins may refuse to schedule tasks if an admission test fails or if the given specification misses key parameters to generate a feasible schedule.
`rtf_plg_change`	A new admission test for an already accepted task is performed using a new set of parameters.
`rtf_plg_schedule`	A task previously accepted (fully or partially) by the current plugin is assigned to it. From this point onward, the plugin is the manager of the task scheduling parameters.
`rtf_plg_release`	The specified task is no longer assigned to the current plugin. It may have left the task set or it may have been moved to another plugin after a change request.
`rtf_plg_attach`	Instructs the plugin to set the scheduling parameters of the given process/thread to match the corresponding task specification (which must be managed by the same plugin).
`rtf_plg_detach`	Instructs the plugin to demote the given process/thread to non real-time scheduling and cancel the association between the process/thread and its task specification.

In particular, each plugin should implement the admission test for each request in the two functions `rtf_plg_accept` and `rtf_plg_change`. This test, if present, should either be based on the simple check of the supplied parameters (e.g. checking that all mandatory parameters are present) or it can perform a necessary schedulability test that takes into account the current condition of the system.

If a plugin is selected to schedule a task after a successful request, the daemon will signal the selected plugin using the `rtf_plg_schedule` function. At that point, if the specification is already associated with an existing execution flow, the plugin proceeds to assign the actual real-time scheduling parameters to it. If no thread is associated with the assigned task specification, this operation is delayed until the daemon invokes `rtf_plg_attach` with the same task specification. The selected parameters will be in effect until either the request is assigned to another plugin after a successful change of its parameters or it is removed from the task set by the client.

Each plugin is implemented as a dynamic-link library that implements at least the set of functions shown in Table 4 and it is distributed as a `.so` file that will be loaded by the *ReTiF Daemon* during system initialization phase. While each plugin operates at user-space level, some of them may require cer-

Listing 2: Example of an *ReTiF Daemon* configuration file.

```
# Name  Plugin  Priority  Cores
  EDF   EDF.so  100-100   0
  RM1   RM.so    50-99    1,2
  RR    RR.so     1-50    1,2
  RM2   RM.so     1-99    3,4
  FP    FP.so     1-99    5-7
```

tain features provided only by certain OSes or by specific kernel modules. This design choice allows us to disentangle the *ReTiF Daemon* from the underlying OS kernel implementation as much as possible, relegating all OS-specific code to dynamically loaded plugins. For example, among the plugins described in Section 5, the EDF plugin is Linux specific—it relies on `SCHED_DEADLINE`—while the others use standard POSIX schedulers (like `SCHED_FIFO` or `SCHED_RR`) and real-time priorities. In case a specific kernel module is needed, the plugin can load it during its initialization phase. If some plugin detects that some mandatory features of the underlying kernel are missing, it can abort the whole initialization of the system and an error is returned to the user; the system administrator shall then either change the daemon configuration file or check that the requirements of each selected plugin can be met before starting it again.

3.5. ReTiF Daemon *Configuration*

The daemon reads a configuration file on startup to determine the list of plugins that shall be loaded during startup. The format of this file is similar to the one shown in Listing 2: each line specifies a name for the plugin to be loaded, the `.so` file containing the plugin implementation, a range of POSIX real-time priorities associated with the plugin, and the list of CPUs that the plugin can use to schedule tasks assigned to it[6]. Each plugin file may be loaded multiple times under different name and parameters: this is particularly useful to specify different access control policies for different instances of the same plugin, as described in Section 4.

When dispatching client requests to the plugins, the *ReTiF Daemon* will assign each plugin a priority based on the order in which they are specified in the configuration file.

4. Access Control Model

On embedded or dedicated real-time systems, typical applications run under strict timing requirements, and, often, their timing behavior is entirely under

[6]It must be noted however that task allocation to CPUs depends on the implementation of each plugin.

control of the system designer. Therefore, it is commonplace to run these applications with a high privilege level on the OS. On the other hand, this is not the case for general-purpose systems. Indeed, on GPOSes users may have the ability to install custom applications or libraries and run them at will, albeit with limited privileges. Using an expressive access control model, system designers can regain control of the available resources on general purpose environments.

In this section, we present a novel access-control model developed for *ReTiF*. The purpose of this model is to provide system designers with proper tools to regulate access to the real-time features available through the framework, which is especially useful when the host is shared by multiple applications or even multiple users. The interactions with the framework are regulated, at first, through the normal users/groups assignment in the OS, and the permissions associated to access the daemon IPC entry point. However, without a proper access-control mechanism, unprivileged users might leverage the framework to take control of more resources than they should, sending a simple request to the *ReTiF Daemon*.

4.1. Security Requirements for the Framework

The access control model that we identified for *ReTiF* should satisfy at least the following requirements:

1. **Explicit consent:** The framework shall expose access for unprivileged users to the kernel real-time scheduling features after an explicit grant from the system administrator. Access to these resources should be regulated by identifying the individual users or groups that can access these features altogether on daemon start-up time.
2. **Parameter bounds:** System administrators shall be able to specify appropriate user-specific bounds for each parameter that may be included in a request for the framework; this feature can be used to ensure that no user may over-allocate resources and consequently prevent others to access those resources.
3. **Ownership control:** Requests sent by a specific process should not target a task that does not belong to the same user (except for requests made by the system administrator who can change scheduling parameters without restrictions); this applies in particular to requests for attaching/detaching threads and for changing or releasing accepted scheduling parameters.
4. **Rules administration:** Administration of the access-control configuration should be allowed only to the system administrator.

In the following, we illustrate the security model that we devised and show how this model can effectively protect against unintended or malicious uses of the proposed framework.

4.2. Access Control Specification and Model

Here we identify the set of rules that characterize our access control model, in order to satisfy the aforementioned requirements. System designers can enforce certain policies by specifying a subset of these rules in a configuration file (see Section 4.4), which is parsed and interpreted by the *ReTiF Daemon* during system initialization.

Each rule in our model may target the entirety of the system, a single user, a group of users, or any entity that may use a specific plugin. In what follows, we will refer to a *domain* Δ to identify the owner of the application that originates a scheduling request to the framework. In general, a domain can correspond to the *global domain*, which includes any application on the system, a *user*, or a *group*. To identify each instance of a plugin loaded in the system Φ, for which multiple instances may be loaded at the same time, the name supplied as first parameter in the *ReTiF Daemon* configuration file will be used (see Section 3.5).

Given a rule R_i, we identify the *subject* σ_i of that rule with the pair (Δ_i, Φ_i), where Δ_i is a domain and Φ_i is an instance of a scheduling plugin. If the latter is omitted, then the rule applies to any request originated from the domain. We also identify the *target* of each rule t_i as a pair (L_i, V_i), where L_i is an *access control property* and V_i is the corresponding value.

Each access control property may impose a limit to the possible values that can be supplied for a parameter in the scheduling request or it may refer to *aggregate values* kept by the *ReTiF Daemon* as more resources are allocated for each real-time task. Each aggregate value is kept with respect to the corresponding subject σ_i. The access control properties defined to enable access to the features exposed by the framework are the following:

i. **Maximum aggregate utilization** U_i^{max} limits the maximum total utilization that may be requested for all tasks matching a given subject σ_i (i.e., a pair of domain and plugin). If not limited, any subject could potentially seize all available resources, leaving other subjects on the same system unable to run their own applications. For the definition of the utilization of each task, see Section 5.1.
ii. **Maximum runtime** Q_i^{max} limits the maximum runtime that can be assigned to each task. This rule prevents any subject from running high-priority applications that introduce long starvation periods on the system, preempting low-priority applications from running for extended periods of time.
iii. **Maximum period** T_i^{max} limits the maximum period that can be specified for each task. Long periods enable applications to access longer runtimes even while maintaining low system utilization. Hence this rule, combined with the limit on the maximum runtime, is essential to avoid long starvation periods in the system.
iv. **Minimum period** T_i^{min} limits the minimum period that can be specified for each task. Running tasks with very small activation periods would introduce a lot of overhead on the system, which has to deal with many task activations. Some plugins for soft real-time systems may also consider scheduling and other system overheads negligible; introducing lots of overhead would break this assumption.

v. **Minimum deadline** D_i^{min} limits the minimum deadline that can be specified for each task. Specifying very small relative deadlines is typically equivalent to indicating that a task has a very high priority (for deadline-based scheduling algorithms, like EDF). Tasks like this prevent tasks declaring longer deadlines from being accepted into the system if not kept in check.

vi. **Maximum deadline** D_i^{max} limits the maximum deadline that can be specified for each task. This limit has been added for completeness.

vii. **Maximum priority** P_i^{max} limits the maximum POSIX priority that can be specified for each task. This can be used to separate the set of priorities accessible by different subjects, de facto limiting the degree of privilege each subject may have access to.

viii. **Minimum priority** P_i^{min} limits the minimum POSIX priority that can be specified for each task. This is to be used in combination with the previous rule.

ix. **Ignore admission test** I_i is a boolean flag that indicates whether tasks from a subject can use the namesake parameter upon task creation/parameters change request to skip any admission test. This permission can be used to grant specific subjects a better degree of freedom, and it is intended to be used in special cases in which certain tasks may be okay with missing some deadlines from time to time. However, this rule should be handled with care, as it grants subjects the capability to introduce in an otherwise schedulable task set tasks that may bring the system in a non-schedulable condition.

4.3. Enforcing Security Policies

The framework enforces the model's application described above throughout all the different types of requests that might arrive at the daemon. For each type of request, different actions may take place.

In general, the framework applies a default *deny all* rule, which automatically rejects any request if no matching rules are found. By specifying rules using the mechanism in Section 4.4, system administrators can specify which actions are allowed (within the given limits). Note that it is not directly possible to specify an allow-all rule, albeit using multiple rules virtually any behavior can be enabled.

Upon receiving a new request from a specific user application, the daemon will identify the rules that may apply to that request by inspecting the domain corresponding to the application that generated the request: rules that target the *global domain* will be applied to any request; any rule that specifies as domain the *effective user* or the *effective group* associated with the originating application will also apply. In the implementation of the access control mechanism, the *effective user* and *effective group* are identified by retrieving the effective user ID (EUID) and the effective group ID (EGID) respectively from the process ID (PID) of the requesting application.

Rules that do not specify any plugin (i.e. that apply for any plugin) are checked immediately, while the others are checked only if the corresponding plugin should be interrogated to handle the current request. Notice that when multiple rules may apply, all rules should be satisfied for a request to pass, i.e. the resulting policy is obtained by intersection between active rules. For the sake of clarity, consider a situation in which there are two rules: the first one specifies that user u_1 cannot declare task specifications with a total utilization greater than 0.5 (for any plugin), while the second one specifies that the maximum cumulative utilization available for the plugin p_1 plugin should not be greater than 0.3 (for any user/group); in that case, tasks of user u_1 assigned to plugin p_1 cannot exceed 0.3 utilization.

To achieve requirement 3, requests originated from a certain application will be rejected if they target tasks having a different EUID. With this mechanism in place, applications cannot attach scheduling rules to processes or threads associated to other users, which would result in a vulnerability of the system.

Rules are enforced both upon receiving requests to create new tasks or when a request to change the parameters of an already accepted task specification is issued. In the latter case, the limits imposed on the parameters follow the same rules applied in the creation phase and the modification is accepted if the resulting condition of the system after the change satisfies all the active rules for that request.

4.4. Configuring Access Control Policies

To specify the set of rules that shall be enforced by the daemon at runtime, the system administrator can edit an access control configuration file, which is parsed by the *ReTiF Daemon* upon initialization, right after its general configuration file.

Listing 3 shows an example of this access control configuration file. Its structure resembles a whitelist, with each rule specified on a separate line, with a syntax inspired to the `limits.conf` system-wide configuration file on Linux. The syntax of each rule is composed of four columns, where the first two columns are used to identify the *subject* of the rule σ and the last two columns specify the *target* t.

The `Domain` field accepts as valid value either a username, a group name (distinguished from a username using a `@` character at the beginning), or the wildcard value `'-'`, to identify the *global domain.* The `Plugin-name` field is either a plugin name, which must correspond to the name of one of the plugins included in the *ReTiF Daemon* general configuration file, or the wildcard value `'-'`, to specify that the rule applies to any plugin. Finally, Table 5 shows the accepted keys for the `Property` field, alongside the unit or range that is used to specify the corresponding `Value` field.

5. Implementation and Plugins Suite

ReTiF implementation reflects the overall architecture described in Section 3.1. Currently, the implementation of the *ReTiF Daemon* partially supports

Table 5: List of access control properties L and respective keys used in the access control configuration file.

Property L	**Symbol**	**Unit or Range**	**Key**
Maximum Aggregate Utilization	U^{max}	$> 0.0^{\dagger}$	`max_utilization`
Maximum Runtime	Q^{max}	µs	`max_runtime`
Minimum Period	T^{min}	µs	`min_period`
Maximum Period	T^{max}	µs	`max_period`
Minimum Deadline	D^{min}	µs	`min_deadline`
Maximum Deadline	D^{max}	µs	`max_deadline`
Minimum Priority	P^{min}	$[1..100]^{*}$	`min_priority`
Maximum Priority	P^{max}	$[1..100]^{*}$	`max_priority`
Ignore Admission Test	I	true/false	`ignore_adm_test`

† Uses floating precision, accepts any value greater than zero, where each unit corresponds to a fully utilized core (e.g. 1.0 is one fully utilized core, 2.0 two full cores, etc.).
* Actual range depends on the number of priorities supported by the target platform, as advertised by `sched_get_priority_min/max` POSIX functions.

Listing 3: Example of access control configuration file.

```
# Domain    Plugin-name   Property         Value
# Limit the aggregate utilization of user1 on EDF to 70%
  user1     EDF           max_utilization  0.7
# Limit the CDROM group task period using any plugin to 9 ms
  @cdrom    -             max_period       9000
```

the access control mechanism described in Section 4. The software is freely available on GitHub, under a GPLv3 license, at: `https://github.com/gabriserra/retif`.

This section summarizes the characteristics of a suite of plugins that we provide alongside the framework to both test the functionality of our implementation and to provide access to common real-time features. Table 6 summarizes the task parameters supported by each plugin included in the suite.

5.1. EDF Plugin

This Linux-specific plugin provides an implementation of the Earliest Deadline First (EDF) scheduling algorithm, which is well known to be optimum for single processor systems [39], on top of the `SCHED_DEADLINE` scheduling class.

In particular, this implementation provides a fully partitioned version of EDF that employes a worst-fit task allocation strategy among the CPU cores specified via the *ReTiF Daemon* configuration file. `SCHED_DEADLINE` is an implementation of EDF that is offered by the Linux mainline kernel since version 3.14 [40] based on CPU reservations implemented through a variant of the

Constant Bandwidth Server (CBS) [7]; each task (represented by a single execution flow) is assigned its own reservation to be run into, based on its runtime Q and period T. The scheduler then uses the optional deadline parameter D to apply the CBS/EDF scheduling strategy among the reservations related to this scheduling class.

The plugin performs a simple task specification admission test based on the total utilization registered for each CPU. The *utilization* of each task τ_i is defined by the following ratio

$$U_i = \frac{Q_i}{min\{T_i, D_i\}} \tag{1}$$

Since each task can only be assigned to one core, each new scheduling request is allocated to the least loaded core. Given a core k and the set of all tasks assigned to that core Γ_k, the load of the core is identified by the sum of the tasks' utilizations belonging to Γ_k

$$U_k = \sum_{\tau_i \in \Gamma_k} U_i = \sum_{\tau_i \in \Gamma_k} \frac{Q_i}{min\{T_i, D_i\}} \tag{2}$$

Once the CPU is selected, a new task is accepted if the admission of the new task into the current task set does not lead the system to an overload condition, that is if the load of the selected core after the inclusion of the new task is still less than or equal to a threshold $U^{thr} \leq 1$. This test is a sufficient schedulability test for partitioned EDF [39]; hence, given the least loaded core $\overline{k}$, this plugin accepts a new task τ_j to be scheduled if the following condition holds true:

$$U_{\overline{k}} + \frac{Q_j}{\min\{T_j, D_j\}} \leq U^{thr} \tag{3}$$

If this condition is satisfied, the request is accepted and assigned to the least loaded core $\bar{k}$, otherwise it is rejected.

Notice that U^{thr} can be customized and also notice that admission for tasks that declare runtime and period is subject to other checks to ensure that all access control policies are always met. For these reasons, the current implementation of this plugin disables the in-kernel necessary test performed by `SCHED_DEADLINE` for Global EDF[7]. On task specification admission, the plugin sets all scheduling parameters and CPU affinity of the associated process modifying the CPU affinity mask using the `sched_setaffinity` system call.

If a desired runtime Q_j^d is also specified for the task, then the task is assigned an accepted runtime $Q_j^a \in [Q_j, Q_j^d]$ that is the highest value possible given the current load of the system that does not break the acceptance condition:

$$Q_j^a = min(max(Q_j, Q_j^d), (U^{thr} - U_{\bar{k}}) \cdot min\{T_j, D_j\}) \tag{4}$$

Users may also request this plugin to ignore task admission check on failure: in this condition, tasks accepted in spite of a failing test receive an accepted runtime always equal to their minimum runtime Q_j.

[7]Writing -1 to `/proc/sys/kernel/sched_rt_runtime_us`.

Table 6: List of plugins provided with the framework and relative parameters.

Parameter		Plugins			
		EDF	RM	RR	FP
Period	T	✓	✓	–	–
Runtime	Q	✓	†	–	–
Desired Runtime	Q^d	○	–	–	–
Relative Deadline	D	○	○	–	–
Priority	P	–	–	✓	✓

✓	Mandatory
†	Recommended, but not mandatory
○	Optional, default value is used if not supplied
–	Unused

5.2. RM Plugin

This plugin implements the Rate Monotonic (RM) scheduling algorithm, which is well known to be optimum for single processor systems among fixed priority (FP) scheduling strategies [39]. This particular implementation provides a fully partitioned version of RM that uses a worst-fit task allocation strategy on top of the POSIX `SCHED_FIFO` scheduling policy.

As shown in Table 6, the only parameter that is mandatory for this plugin is the task period T. Since tasks may not declare their expected runtime Q and still be admitted by this plugin, a proper admission test based on the schedulability of the system cannot be provided for all tasks. The strategy that we adopted is to apply a necessary-only single-CPU FP utilization test for all tasks that declare both their runtime and period parameters. In the future, this plugin might have an option to enable a sufficient test for RM.

Upon receiving a new request that includes both parameters, the least loaded core $\overline{k}$ is selected as the potential core to schedule the new task, using a worst-fit allocation strategy. Given the set of tasks assigned to that core $\Gamma_{\overline{k}}$, this plugin assigns each task a priority that is inversely proportional to their period:

$$P_i \propto 1/T_i \qquad \forall\, i \in \Gamma_k \tag{5}$$

The calculated priority for each task P_i, which is within the range of POSIX priorities assigned to this plugin via the *ReTiF Daemon* configuration file, is then used to schedule tasks using `SCHED_FIFO`.

5.3. RR and FP Plugins

The Round Robin (RR) and fixed priority (FP) plugins serve as wrappers, exposing underlying POSIX functionality to applications that rely on *ReTiF* to access real-time features. They respectively provide access to `SCHED_RR` and `SCHED_FIFO` scheduling policies and as such their only required parameter is the desired priority P.

Since no proper schedulability analysis can be perfomed with only that parameter, neither plugin performs any check upon receiving a new request, checking only the presence/absence of the priority parameter. Both plugins apply a worst-fit task allocation strategy, in this case resulting in each new task to be assigned to the CPU core with the minimum number of assigned tasks.

The priority requested via the *ReTiF Library* API may differ from the one actually used by either of these plugins to schedule a task; this happens when the range of priorities that each plugin may select (specified via the *ReTiF Daemon* configuration file) differs from the normal range of POSIX priorities. In this situation, each plugin attempts to maintain the ordering of the distinct priorities that have been requested for each real-time task when assigning actual POSIX priorities.

Given a core k and the set of all distinct priorities requested for that core Π_k, the resulting ordering among tasks once actual POSIX priorities have been selected reflects the one specified in input (i.e. total ordering among tasks is mantained) if

$$\mid \Pi_k \mid \leq R_k \tag{6}$$

where R_k is the number of distinct priorities available on CPU k for that plugin.

When this condition is not satisfied (i.e. if the destination range is smaller than the number of distinct priorities requested on that core), some tasks may receive the same priority even if they originally requested two distinct ones. In future versions of this plugin, we might introduce an option that forces reject of the request in such a case.

6. Performance Evaluation

This section presents results of our experiments involving *ReTiF*, with the goal of measuring the overheads that it introduces on real-time applications.

Overheads introduced by *ReTiF* typically depend on two factors that sum up for each operation requested by the application. The first one is the communication cost between the real-time application and the *ReTiF Daemon* through the IPC mechanism used (Unix sockets); this cost is the same for all kinds of requests and is independent from the plugin selected to serve it. The second cost depends on the kind of request performed and on the implementation of each plugin loaded by the *ReTiF Daemon*.

Note that applications have to pay these overheads only when directly sending requests to the *ReTiF Daemon*, to declare new tasks or to request changes to the accepted ones. The cost of scheduling these real-time applications once accepted is no different than the one of running them directly with the underlying OS, because that is what each plugin does: configure the system scheduler to manage each application. Hence, *no overheads are to be expected in the critical loops of real-time applications* with respect to implementations not leveraging *ReTiF*.

Table 7: Characteristics of the two reference platforms used in experiments.

Reference Machine	**Intel**	**ARM**
Server Model	Dell PowerEdge R630	Gigabyte R150-T62
CPUs	Two Intel® Xeon E5-2640 v4	Two Marvell® ThunderX® CN8890
Total Number of Cores	20	96
RAM	64 GB	64 GB
Distribution	Ubuntu 18.04.5 LTS	Ubuntu 20.04.2 LTS
Kernel Version	4.15.0	5.8.0

6.1. Experiments Setup

We performed experiments on two reference server platforms, an Intel and an ARM-based one, varying different parameters and requests to check how these changes affect the cost of each request in *ReTiF*'s user API. Table 7 summarizes the characteristics of each reference platform. During experiments, hyperthreading, CPU frequency scaling, and Turbo Boost were disabled to maximize reproducibility of the results, fixing CPU clock speed of both reference platforms to 2 GHz. For these tests, we did not provide any ACL rule to the framework, de facto disabling access control mechanisms in the *ReTiF Daemon*.

The application used during these benchmark is a single-threaded process that performs multiple requests to the *ReTiF Daemon*, each request adding one new task to the active taskset. The application and the *ReTiF Daemon* are configured to run with highest POSIX real-time priority (99) pinned on separate cores, to avoid interferences from other applications. Finally, on both platforms we used the Time Stamp Counter (TSC) register to measure time differences to maximize the precision of our measurements.

Tests were executed by configuring the *ReTiF Daemon* to load only one plugin at a time, using either EDF, RM, or RR/FP plugins (the last two share the same underlying implementation). As shown in Figure 2, the *ReTiF Daemon* iterates through all the plugins configured when handling `rtf_spec_create` or `rtf_spec_change` requests; in scenarios in which multiple instances of plugins are configured, expected overheads should be higher. For each plugin, we varied the number of CPU cores managed by the plugin from 1 to 20 and the number of tasks already present in the taskset before starting a new request from 0 to 1024. Tasksets are randomly generated (including task parameters) and for each possible configuration 500 different experiment runs were performed, to ensure reproducibility of obtained results.

Notice that in more realistic use case scenarios there would be multiple applications performing requests concurrently to the *ReTiF Daemon*, which is implemented for simplicity as a single process. This means that some applications may experience higher overheads depending on the size of the requests queue of the *ReTiF Daemon* (which in our reference scenario is always empty whenever a new request arrives). All operations handled by the *ReTiF Daemon* are typically performed outside critical loops (since *no scheduling decision is taken by the* ReTiF Daemon *itself or any of its plugins*), and most of them are performed only during initialization (to declare task specifications and attach POSIX threads/processes to them) or cleanup phases (to release allocated resources). Only one request (`rtf_spec_change`) does not fit in any of these two categories. Real-time tasks that plan to request repeatedly changes to their scheduling parameters should be prepared to handle the overhead of this operation if included in their critical loops.

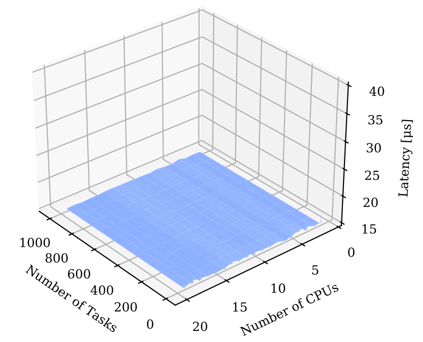
6.2. Declaring New Tasks

We first performed a number of experiments involving the `rtf_spec_create` or `rtf_spec_change` requests, which declares a new task specification. The main goal of this test is to measure how much time it takes from an application point of view to perform the task admission test, depending on the system configuration and plugin selected. To do so, we measure the time needed to perform a `rtf_spec_create` request when varying the system configuration and number of tasks in the active taskset as described above. The benchmarking application performs only task specification declarations, without attaching any actual POSIX process/thread to accepted requests. For `rtf_spec_change`, the cost is very similar and as such we did not include its corresponding results.

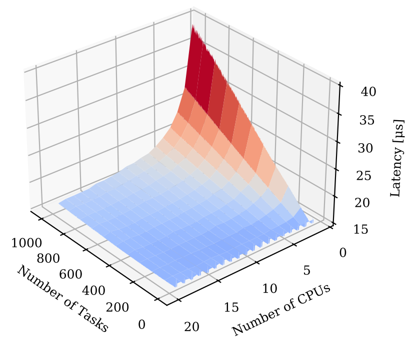
Experimental results for both reference platforms are shown in Figures 3 and 4, which show the time needed to perform each request depending on system configuration. Results vary depending on the plugin implementation.

For the EDF plugin, the time needed to perform each request does not vary with respect to the number of CPUs or the number of tasks already accepted, because the admission test is a simple test based on the total utilization accounted on each CPU, which is maintained across requests. The cost of this call is barely higher than the communication overhead between the benchmarking application and the *ReTiF Daemon* process via the Unix socket, which we measured to be on average around 12 and 50 µs on the Intel and ARM reference platform respectively in our testing configuration.

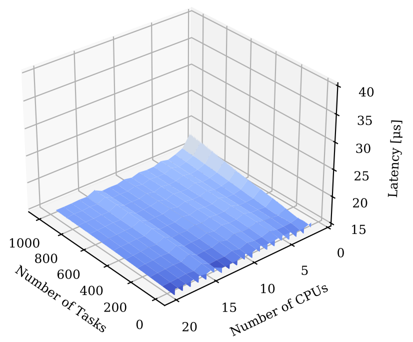
For the other plugins, results show that the time needed to respond each request increases linearly with the number of tasks present in the taskset and with the inverse of the number of CPUs available for the plugin to select from. The linear dependency with the number of tasks in the taskset is easily explained: both plugins try to maintain a partial ordering among tasks based on their priority, and in our implementations this is done by inserting tasks in an ordered linked list. While the implementations of RM and RR/FP plugins are very similar, the latter leverages more often an optimization that reduces the time needed to assign a priority to a task when it requests the same priority of another task already present in the taskset; this optimization is triggered more often for the latter plugin, reducing the average time needed to resolve these requests. The linear relationship with the inverse of the CPUs is also easily explained: when there are more CPUs to choose from, the size of each linked list maintained by the plugins is shorter, because tasks are distributed to the various CPUs using a worst-fit allocation strategy.



(a) EDF Plugin

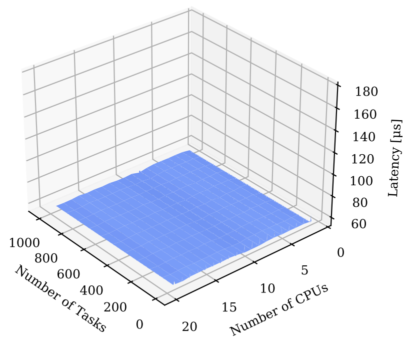


(b) RM Plugin

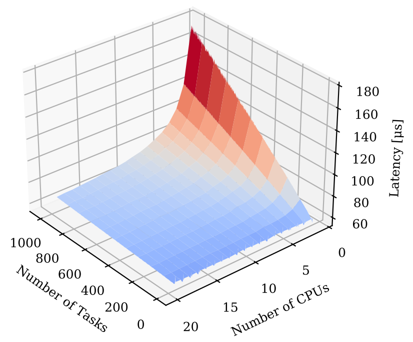


(c) RR/FP Plugin

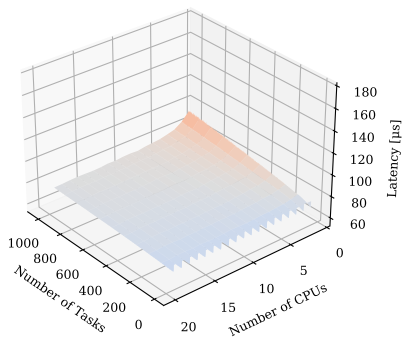
Figure 3: `rtf_spec_create` latency depending on system configuration on the Intel reference platform at 2 GHz.



(a) EDF Plugin

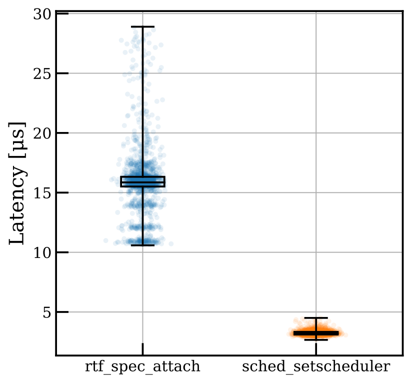


(b) RM Plugin

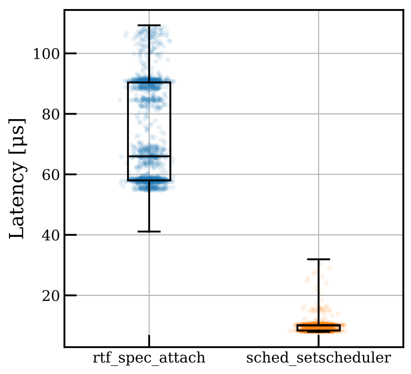


(c) RR/FP Plugin

Figure 4: `rtf_spec_create` latency depending on system configuration on the ARM reference platform at 2 GHz.



(a) Intel



(b) ARM

Figure 5: Comparison between the latencies of `rtf_spec_attach` and `sched_setscheduler` on the two reference platforms at 2 GHz.

6.3. Configuring Scheduling Parameters

In this experiment, we compare the cost of performing a `rtf_spec_attach` request with respect to calling directly `sched_setscheduler` from the real-time application. The two calls are very similar, since in both cases the result is that the POSIX thread/process given as argument will change its scheduling policy to the selected one with the specified priority. In fact, all plugins described in Section 5 use internally the POSIX `sched_setscheduler` to accomplish this goal.

Using *ReTiF*, there is the additional cost due to communication overhead and some checks performed by the *ReTiF Daemon* and the plugin that has been selected to manage the requesting task. Figure 5 shows the overheads measured on our two reference platforms for both functions. Implementation of the `rtf_spec_attach` does not depend on the number of tasks included in the system nor on the number of CPUs managed by each plugin, hence for simplicity we show only the distribution of measured values. All plugins in Section 5 implement this operation in a similar fashion and thus there is virtually no difference in cost depending on which plugin is managing the request. As expected, calling `rtf_spec_attach` takes considerably more time than performing the direct system call; we deem this an acceptable cost to exploit the functions provided by *ReTiF*, like unprivileged and controlled access to real-time scheduling features of the underlying platform.

7. Known Limitations

ReTiF effectively provides safe access to real-time scheduling features for unprivileged tasks on a set of OSes and platforms. It is not perfect though and some aspects of its architectural design and implementation lead to some limitations when it comes to cross-platform compatibility. This section illustrates all known limitations of the framework, how we plan to overcome them and what kind of impact they have on the applications that rely on *ReTiF*.

7.1. POSIX Compliance

We emphasized multiple times that the presented framework is intended to be used on generic (mostly) POSIX-compliant OSes. This includes not only Linux, but other OSes such as macOS-X, FreeBSD[8], Solaris, or others. That said, some aspects of the POSIX specification for real-time features of compliant OSes impose some limits on the effectiveness of *ReTiF*.

As discussed in Section 3.4, the core design principle of the framework distinguishes between its core components (represented by the *ReTiF Daemon* and its corresponding shared library), and the scheduling plugins. Core components are implemented using only portable POSIX features, while plugins encompass all OS-specific and non-portable features of the system, interacting directly with the user-space API of available schedulers. Among the plugins presented in Section 5, most of them are portable components, since they provide access to POSIX-defined schedulers such as `SCHED_FIFO` and `SCHED_RR`; in this sense, only the EDF plugin is Linux-specific, since it leverages `SCHED_DEADLINE`.

That said, there is one key element of the POSIX specification that critically changes the behavior of each of the plugins when running on different OSes. Each plugin implementation leverages the `sched_setscheduler` system call, which is used to assign scheduler and scheduling parameters of real-time tasks once an execution flow is attached to a task specification. Per original POSIX specification, this call accepts a PID and as such it can be used only to change the parameters of a whole process[9]. On the other hand, Linux is not entirely compliant with this specification and it accepts a thread ID (TID) as first parameter[10], easily obtainable using the `gettid` system call[11]. Leveraging this feature, *ReTiF* plugins can set the scheduling properties of individual threads, rather than processes, of unprivileged applications on Linux (and in general in systems which provide a similar behavior).

Strictly POSIX systems however do not have this capability. Even systems which implement POSIX threads using a 1:1 threading model like Linux (i.e. each thread corresponds to a single `task_struct` from the kernel point of view) may not expose TIDs to user-space and may not accept them as parameters for `sched_setscheduler`, since neither of these features are part of the POSIX specification. To set the scheduling properties of a single thread, POSIX pro-

[8]For more info see `https://people.freebsd.org/~schweikh/posix-utilities.html`

[9]For more info see `https://pubs.opengroup.org/onlinepubs/9699919799/functions/sched_setscheduler.html`

[10]For more info see `https://man7.org/linux/man-pages/man2/sched_setscheduler.2.html`

[11]In the case a PID is used, on Linux only the parameters of the main application thread are changed, and not the parameters of the whole process, since PID and TID of the main thread coincide.

vides the `pthread_setschedparam` function as the only option, which can only be called from the context of the process which the thread belongs to[12].

Per *ReTiF* design, an external privileged process (the daemon) is the only one capable of interacting with OS schedulers. In this condition, if an individual POSIX thread must be scheduled, there must be a way for an external process to access its thread ID and to use it to change its scheduling parameters. If that is not the case, then for that particular OS only scheduling properties for entire processes can be managed by *ReTiF*, while real-time threads cannot be used. For example, FreeBSD, which provides TIDs (albeit with a different API call), does not allow them as argument of `sched_setscheduler`, and as such only real-time process scheduling can be achieved with *ReTiF* on FreeBSD[13].

We understand that this represents a major limitation to the portability of applications relying on *ReTiF* to access real-time features, which should stick with multi-processes rather than multi-threading if cross-OS portability is desired, and we plan to investigate in the future other techniques to enable unprivileged multi-threaded real-time scheduling in the future, perhaps with the help of other non-standard OS schedulers.

7.2. Plugins Coexistence

The most straightforward way of configuring a system using *ReTiF* is to assign a separate set of CPUs to each plugin that are managed independently from the others. This way, each plugin can maintain a separate accounting of the resources already in use in the system without the need for caring about what other plugins may have already assigned. This imposes a limitation on the flexibility of the managed systems; while there is no rule in place to prevent allocating the same CPU cores (or even partially overlapping subsets of the CPUs on the system) to multiple plugins, the resulting behavior may not provide the desired level of guarantees with respect to hard partitioning of CPUs to plugins. For example, the RR/FP plugins do not implement any check on the tasks they manage and as such the coexistence of these plugins with others that do implement some checks may compromise the guarantees provided to user applications if not configured wisely.

We plan to investigate these situations in the future to devise some mechanism that will either prevent these situations or mitigate their effects to some degree. For now, while we do not explicitly forbid the overlap of CPU sets for different plugins, we strongly advise against it. Indeed, the daemon issues a warning if it detects partially overlapping CPU sets of the configured plugins on start.

[12]For more info see `https://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_setschedparam.html`

[13]For more info see `https://www.freebsd.org/cgi/man.cgi?query=sched_setscheduler&sektion=2&manpath=FreeBSD+13.0-current`

7.3. Partitioned Scheduling Only

As of now, *ReTiF* supports *partitioned scheduling* algorithms only via the implementations of the plugins described in Section 5. Support for other task allocation strategies like *global*, *clustered*, or even *semi-partitioned* scheduling may be introduced in the future with additional plugins. If each plugin is assigned a non-overlapping set of CPUs, multiple scheduling domains may leverage not only different scheduling algorithms, but also different task allocation strategies to CPUs, provided that the underlying OS implements these allocation strategies in one of its real-time schedulers in the first place. For example, the Linux `SCHED_DEADLINE` scheduler (leveraged by our EDF plugin) implements global scheduling and it can also be configured via `cgroups` to operate on non-overlapping scheduling domains independently[14]. A future implementation of another EDF-based plugin may leverage these features to implement global scheduling on a subset of the CPUs on the system.

7.4. Support for More Complex Scheduling Algorithms and Task Models

As it is now, *ReTiF* supports a simplified task model that does not take into account valuable aspects that characterize complex real-time systems, e.g., blocking times, task offsets, possible task dependencies, and others. We plan to introduce support to a broader set of (optional) declarative parameters in future revisions of the framework, providing plugin developers additional tools to perform more thorough evaluations of taskset schedulability on task arrival.

In addition, we plan to develop new plugins that may leverage other frameworks mentioned in Section 2 that operate at a lower level to provide access to a broader set of schedulers and policies, for example by supporting LITMUS$^{\mathrm{RT}}$. This way, user-space applications may leverage our simple and unified API to test their behavior under different schedulers, even those that are not part of the mainline version of the operating system kernel.

7.5. Handling Transients

ReTiF supports dynamic tasksets in which tasks can dynamically enter/leave the current taskset at any time. These operations can cause transients in the system that might lead to some missed deadlines if not properly addressed. For now, no mechanism to prevent these kinds of transient is implemented yet. In the future, some support for change protocols should be added to the various plugins for a proper handling of these transients.

7.6. Virtualization and Hierarchical Scheduling

Usability of *ReTiF* in virtualized contexts is very limited. As it is now, *ReTiF* assumes that it is running in an OS that has complete ownership of visible resources (active CPUs, etc.). If the OS hosting *ReTiF* runs as a guest under a virtual machine monitor (VMM) or hypervisor, this assumption is correct only

[14]For more info see `https://lwn.net/Articles/747088/`

if the VMM performs hardware partitioning, so that virtual CPUs in the guest are mapped exclusively to dedicated physical CPUs on the host. Otherwise, the *ReTiF* is not able to provide the expected guarantees to user applications. These problems are well known in the field of hierarchical scheduling and in future releases we may enrich the framework with knowledge of how vCPUs are scheduled within the host OS (e.g., knowing its supply-bound function), and apply accordingly an approriate hierarchical analysis when admitting new tasks.

8. Conclusions and Future Work

This work describes the architecture of *ReTiF*, a newfangled framework designed to improve the accessibility to real-time capabilities of POSIX compliant OSes. We described the *ReTiF* design and implementation, focusing on the aspects that improve accessibility to real-time features offered by modern GPOSes from user space. Thanks to its declarative approach, *ReTiF* greatly simplifies the interface between user applications and the underlying OS, providing applications a simple and portable API. We also described how the security mechanisms embedded in *ReTiF* give complete control to system designers over the resources managed by the framework itself. Finally, we showed the overheads introduced by this framework.

8.1. Future work

We are actively working to continue the development of *ReTiF*, to include further aspects that will improve the usability and robustness of the framework, especially with respect to the timing behavior of the managed real-time tasks. The current implementation can be improved in a number of aspects; we showed the most notable of these in Section 7, where we also mention some directions of future development of the framework, including tasks allocation strategies different than partitioned scheduling, the usage of the framework in virtualized environments, and others. To handle transients due to tasks dynamically entering or leaving the system, we plan to introduce some mode-change protocol [41–44] between the *ReTiF Daemon* and the loaded plugins to address this potential issue. Besides transients, the use of TID to identify each real-time task could lead plugins to mistakenly change the scheduling parameters of unrelated threads if the system reuses the TIDs of terminated real-time threads over time. In future extensions, we will consider the possibility to use *pidfds*[15], which have been added recently in the Linux kernel for cases like this, although the portability of this feature to other POSIX systems may be problematic.

Furthermore, we plan to include support to a broader set of features that improve the usability of the framework in certain scenarios. Energy efficiency is a topic of paramount importance for mobile and embedded systems. On

[15]For more info see `https://lwn.net/Articles/794707`

architectures that support power management techniques like Dynamic Voltage and Frequency Scaling (DVFS), the computation time may vary depending on the frequency of the CPUs or even the type of the CPU core selected to run tasks in heterogeneous computing platgforms (e.g. on ARM big.LITTLE or DynamIQ architectures). We plan to add support for energy awareness to the framework introducing an optional module that will let applications specify their runtime using a frequency-independent measurement unit.

Finally, further extensions might enrich the proposed API to accept additional parameters from user applications–like preferred or mandatory CPU affinity constraints, blocking times or task offsets–or to let implement other plugins that make use of more advanced task admission tests.

References

[1] J. Kiszka, Towards Linux as a real-time hypervisor, in: Proceedings of the 11th Real-Time Linux Workshop, Citeseer, 2009, pp. 215–224.

[2] D. B. de Oliveira, D. Casini, R. S. de Oliveira, T. Cucinotta, Demystifying the Real-Time Linux Scheduling Latency, in: 32nd Euromicro Conference on Real-Time Systems (ECRTS 2020), Vol. 165, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2020, pp. 9:1–9:23.

[3] K. M. Obenland, The use of POSIX in real-time systems, assessing its effectiveness and performance, The MITRE Corporation (2000).

[4] S. Rostedt, D. V. Hart, Internals of the RT patch, in: Proceedings of the Linux symposium, Vol. 2, Citeseer, 2007, pp. 161–172.

[5] F. Reghenzani, G. Massari, W. Fornaciari, The real-time Linux kernel: A survey on PREEMPT_RT, ACM Computing Surveys (CSUR) 52 (1) (2019) 1–36.

[6] J. Lelli, C. Scordino, L. Abeni, D. Faggioli, Deadline scheduling in the Linux kernel, Software: Practice and Experience 46 (6) (2016) 821–839.

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

[8] G. Serra, G. Ara, P. Fara, T. Cucinotta, An architecture for declarative real-time scheduling on Linux, in: 23rd International Symposium on Real-Time Distributed Computing (ISORC), IEEE, 2020, pp. 20–28.

[9] Ayers, B. V. Yodaiken, Introducing real-time Linux, Linux Journal 1997 (34) (1997).

[10] L. Dozio, P. Mantegazza, Real time distributed control systems using rtai, in: Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, IEEE, 2003, pp. 11–18.

[11] P. Gerum, Xenomai–Implementing a RTOS emulation framework on GNU/Linux, White Paper, Xenomai (2004) 1–12.

[12] S. Han, H.-W. Jin, Kernel-level ARINC 653 partitioning for Linux, in: Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12, ACM Press, 2012, pp. 1632–1637.

[13] A. Atlas, A. Bestavros, Design and implementation of statistical rate monotonic scheduling in KURT Linux, in: Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No. 99CB37054), IEEE, 1999, pp. 272–276.

[14] T. Cucinotta, L. Palopoli, L. Marzario, G. Lipari, L. Abeni, Adaptive reservations in a linux environment, in: Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004., 2004, pp. 238–245.

[15] L. Marzario, G. Lipari, P. Balbastre, A. Crespo, Iris: a new reclaiming algorithm for server-based real-time systems, in: Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004., 2004, pp. 211–218.

[16] R. Rajkumar, C. Lee, J. Lehoczky, D. Siewiorek, A resource allocation model for qos management, in: Proceedings Real-Time Systems Symposium, 1997, pp. 298–307.

[17] S. Ghosh, J. Hansen, R. Rajkumar, J. Lehoczky, Integrated resource management and scheduling with multi-resource constraints, in: 25th IEEE International Real-Time Systems Symposium, 2004, pp. 12–22.

[18] S. Brandt, G. Nutt, T. Berk, J. Mankovich, A dynamic quality of service middleware agent for mediating application resource usage, in: Proceedings 19th IEEE Real-Time Systems Symposium, 1998, pp. 307–317.

[19] S. Childs, D. Ingram, The Linux-SRT integrated multimedia operating system: Bringing QoS to the desktop, in: Proceedings Seventh IEEE Real-Time Technology and Applications Symposium, IEEE, 2001, pp. 135–140.

[20] B. Srinivasan, S. Pather, R. Hill, F. Ansari, D. Niehaus, A firm real-time system implementation using commercial off-the-shelf hardware and free software, in: Proceedings. Fourth IEEE Real-Time Technology and Applications Symposium, IEEE, 1998, pp. 112–119.

[21] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, J. H. Anderson, LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers, in: 27th IEEE International Real-Time Systems Symposium (RTSS'06), 2006, pp. 111–126.

[22] M. Aldea, G. Bernat, I. Broster, A. Burns, R. Dobrin, J. M. Drake, G. Fohler, P. Gai, M. G. Harbour, G. Guidi, J. J. Gutierrez, T. Lennvall, G. Lipari, J. M. Martinez, J. L. Medina, J. C. Palencia, M. Trimarchi, FSF:

A real-time scheduling architecture framework, in: 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06), 2006, pp. 113–124.

[23] P. Gai, L. Abeni, M. Giorgi, G. Buttazzo, A new kernel approach for modular real-time systems development, in: Proceedings 13th Euromicro Conference on Real-Time Systems, 2001, pp. 199–206.

[24] M. Aldea-Rivas, M. Harbour, MaRTE OS: An Ada kernel for real-time embedded applications, Lecture Notes in Computer Science 2043 (2001) 305–316.

[25] M. Sojka, P. Píša, D. Faggioli, T. Cucinotta, F. Checconi, Z. Hanzálek, G. Lipari, Modular software architecture for flexible reservation mechanisms on heterogeneous resources, Journal of Systems Architecture 57 (4) (2011) 366 – 382.

[26] L. Palopoli, T. Cucinotta, L. Marzario, G. Lipari, AQuoSA–Adaptive quality of service architecture, Software: Practice and Experience 39 (1) (2009) 1–31.

[27] M. Sojka, M. Molnar, Z. Hanzalek, Experiments for real-time communication contracts in IEEE 802.11e EDCA networks, in: 2008 IEEE International Workshop on Factory Communication Systems, 2008, pp. 89–92.

[28] D. Sangorrín, M. González Harbour, H. Pérez, J. J. Gutiérrez, Managing transactions in flexible distributed real-time systems, in: Reliable Software Technologiey – Ada-Europe 2010, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 251–264.

[29] M. Åsberg, T. Nolte, S. Kato, R. Rajkumar, ExSched: An external CPU scheduler framework for real-time systems, in: 2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2012, pp. 240–249.

[30] A. Mancina, Operating Systems And Resource Reservations – Ph.D. thesis (2009).

[31] J. N. Herder, H. Bos, B. Gras, P. Homburg, A. S. Tanenbaum, Minix 3: A highly reliable, self-repairing operating system, SIGOPS Oper. Syst. Rev. 40 (3) (2006) 80–89. doi:10.1145/1151374.1151391.
URL https://doi.org/10.1145/1151374.1151391

[32] G. Parmer, R. West, Hijack: Taking control of COTS systems for real-time user-level services, in: 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07), IEEE, 2007, pp. 133–146.

[33] M. Behnam, T. Nolte, I. Shin, M. Åsberg, R. Bril, Towards hierarchical scheduling in VxWorks, in: OSPERT 2008, Fourth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, 2008, pp. 63–72.

[34] R. Inam, J. Mäki-Turja, M. Sjödin, S. M. Ashjaei, S. Afshar, Support for hierarchical scheduling in FreeRTOS, in: ETFA 2011, IEEE, 2011, pp. 1–10.

[35] H. Wei, Z. Shao, Z. Huang, R. Chen, Y. Guan, J. Tan, Z. Shao, Rt-ros: A real-time ros architecture on multi-core processors, Future Generation Computer Systems 56 (2016) 171–178.

[36] Y. Saito, F. Sato, T. Azumi, S. Kato, N. Nishio, Rosch: Real-time scheduling framework for ros, in: 2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), IEEE, 2018, pp. 52–58.

[37] S. Xi, C. Li, C. Lu, C. D. Gill, M. Xu, L. T. Phan, I. Lee, O. Sokolsky, Rt-open stack: Cpu resource management for real-time cloud computing, in: 2015 IEEE 8th International Conference on Cloud Computing, IEEE, 2015, pp. 179–186.

[38] T. Cucinotta, D. Giani, D. Faggioli, F. Checconi, Effective real-time computing on Linux, in: 12th Real-Time Linux Workshop, 2010, pp. 1–11.

[39] G. Buttazzo, Hard real-time computing systems: predictable scheduling algorithms and applications, Vol. 24, Springer Science & Business Media, 2011.

[40] D. Faggioli, F. Checconi, M. Trimarchi, C. Scordino, An EDF scheduling class for the Linux kernel, in: 11th Real-Time Linux Workshop, 2009, pp. 1–8.

[41] K. W. Tindell, A. Burns, A. J. Wellings, Mode changes in priority preemptively scheduled systems, in: [1992] Proceedings Real-Time Systems Symposium, 1992, pp. 100–109.

[42] L. Sha, R. Rajkumar, J. Lehoczky, K. Ramamritham, Mode change protocols for priority-driven preemptive scheduling, Real-Time Systems 1 (10 1996). doi:10.1007/BF00365439.

[43] P. Pedro, A. Burns, Schedulability analysis for mode changes in flexible real-time systems, in: Proceeding. 10th EUROMICRO Workshop on Real-Time Systems (Cat. No.98EX168), 1998, pp. 172–179.

[44] D. Casini, A. Biondi, G. Buttazzo, Handling transients of dynamic real-time workload under EDF scheduling, IEEE Transactions on Computers 68 (6) (2018) 820–835.