An Architecture for Declarative Real-Time Scheduling on Linux

Gabriele Serra, Gabriele Ara, Pietro Fara, Tommaso Cucinotta
Scuola Superiore Sant'Anna, Pisa, Italy
Email: {name.surname}@santannapisa.it

***Abstract*—This paper proposes a novel framework and programming model for real-time applications supporting a *declarative* access to real-time CPU scheduling features that are available on an operating system. The core idea is to let applications declare their temporal characteristics and/or requirements on the CPU allocation, where, for example, some of them may require real-time POSIX priorities, whilst others might need resource reservations through SCHED_DEADLINE. The framework can properly handle such a set of heterogeneous requirements configuring an underlying multi-core platform so to exploit the various scheduling disciplines that are available in the kernel, matching applications requirements. The framework is realized as a modular architecture in which different plugins handle independently certain real-time scheduling features within the underlying kernel, easing the customization of its behavior to support other schedulers or operating systems by adding further plugins.**

I. Introduction

In the past decade, we witnessed a raising interest in the topic of running real-time applications in distributed or embedded systems by deploying them on General Purpose Operating Systems (GPOSes). Examples of application scenarios that can leverage real-time features included in modern GPOSes are multimedia applications like audio/video processing and streaming, gaming, etc. Typically, these application scenarios are characterized by the coexistence of both real-time and non-real-time applications on the same host.

Among the various GPOSes available that provide support for both real-time and non-real-time applications at the same time, Linux is a common choice for applications that must support a rich set of multimedia peripherals, thanks to the plethora of libraries and tools that have been developed over the past years. In addition, the use of the Android Operating System (OS), based on Linux, has become a popular choice for a number of embedded systems for multimedia services, from tablets to infotainment systems deployed in modern cars.

Similarly to any other GPOS, the Linux kernel has been focusing on minimizing average OS overheads and optimizing in-kernel operations so to lead to the maximum possible performance for applications, while keeping good responsiveness for interactive workloads, notably user interactions and multimedia applications. However, Linux has been consistently improving its support for real-time systems through a set of interesting features [1]: the inclusion of POSIX real-time extensions [2] and the support for real-time mutexes; high-resolution timers with nano-second precision; 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 bookeeping timer; the PREEMPT_RT [3], [4] 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 [5], implementing a global Earliest Deadline First (EDF) algorithm (but it can also be configured as partitioned or clustered EDF) that uses the Constant Bandwidth Server (CBS) [6] 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.

Due to its open nature, Linux may be required to host a variety of different applications with heterogeneous temporal characteristics and real-time requirements, ranging from interactive applications, to multimedia and virtual-reality tools, to real-time control applications for factory automation. These applications may activate periodically or sporadically, they may require access to the real-time scheduling priorities, or sometimes these should be someway inferred by their periodicity as compared to the one of other co-located applications, or they may require SCHED_DEADLINE reservations. 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.

A. Contribution

In this work, we propose a novel framework that can be used to provide access to real-time CPU scheduling features that are available on Linux, improving the usability of existing real-time capabilities by providing a unified API.

The main focus is to enrich the OS with a new middleware that can be used to declare the temporal characteristics of real-time applications without enforcing the use of a specific scheduling technique. Instead, a *declarative* approach has been adopted to allow applications characterized by heterogeneous requirements to coexist on the same host. Leveraging this

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

new framework, applications can simply declare a set of timing characteristics and scheduling requirements that may span from informing the OS about their minimum periodicity and/or worst-case execution times, to requesting the use of specific POSIX real-time priorities or SCHED_DEADLINE reservations, without worrying about the techniques that will be used to match the declared attributes by each application.

To achieve this goal, the proposed framework adopts a modular architecture that properly handles a number of heterogeneous requirements, in which a set of plugins are used to translate from the attributes declared by each application to proper configurations of the real-time features exposed by the underlying kernel. The framework is capable of partitioning the tasks requiring real-time scheduling services among the CPUs available in an underlying multi-core platform, so that for example rate-monotonic is used on some CPUs, while SCHED_DEADLINE reservations is used on others. This modularity can be leveraged not only to support a plethora of different real-time applications on Linux, but also to provide a common API that can be exploited in the future to support portability of real-time applications across other OSes, by adding platform-specific implementations of certain plugins.

As a proof of concept, in this work we present a first open-source implementation of this framework for the Linux OS. The software is freely available on GitHub, under a GPLv3 license, at: https://github.com/gabriserra/declarative-rtd.

II. Related Work

In this section, we briefly summarize available mechanisms to run real-time applications on GPOSes, with a particular focus on Linux, along with some of the extensions appeared in the research literature. The discussion considers the most relevant works only, while a comprehensive review of existing state-of-the-art approaches is postponed to future works.

Historically, real-time applications support has been introduced in Linux by using techniques that embed a real-time micro-kernel layer between the hardware and the kernel, that in practice acts like a hypervisor. In this scenario, there is a distinct separation between real-time and "normal" tasks, in which real-time tasks are handled by the corresponding micro-kernel while the others are scheduled at a lower priority by Linux. The two major implementations of this paradigm have been RT-Linux, proposed by Yodaiken et al. [7], and RTAI, proposed by Mantegazza et al. [8]; the latter has also been later forked by Gerum et al. into another project called Xenomai [9]. These solutions usually require real-time applications to be written using specific APIs and they must be distributed as kernel modules instead of user-space applications. For this reason, these solutions are not suitable for certain application scenarios, like audio/video processing applications with soft real-time requirements that should be executed by unprivileged users and in user-space context. A similar approach characterizes the implementation of a kernel-level partitioning mechanism for Linux that abides by the ARINC-653 specification [10]. This implementation provides a high level of isolation among applications, as needed for the avionic field [11], but they are not suitable for other application scenarios.

Many real-time kernel extensions and middleware solutions have been proposed to support both hard and soft real-time applications on Linux or other GPOSes. A representative example is KURT Linux [12], which proposes a major modification to the scheduling mechanisms by introducing 3 distinct operational modes: in *normal mode* the system behaves like any traditional GPOS; in *real-time mode* only real-time processes are executed and normal processes are blocked; finally, in *mixed mode* both real-time and non real-time applications can be executed concurrently. In particular, the last mode allows for the execution of non real-time processes during the slack time of real-time applications, hence real-time applications have a strict precedence over other ones. In [13] authors present DQM, a quality of service middleware for mediating access to physical resources by applications that supports dynamic workloads. In QRAM [14], an off-line optimization for allocating multiple resources across real-time applications is proposed to maximize an overall QoS cost function for the system. The approach has also been extended [15] with an adaptive on-line optimization policy.

Another real-time extension for the Linux kernel with a certain relevance within the research community is represented by LITMUS^{RT} [16] by Calandrino et al., which can be used to implement different scheduling algorithms and other real-time policies in the form of plugins. The framework is composed by a set of patches to be applied to the Linux kernel and a user-space library that is used by real-time applications to exploit the added functionality. The main goal of LITMUS^{RT} is to provide the research community with a test bench for real-time scheduling algorithms so to ease their implementation on Linux. However, it is out of the scope of LITMUS^{RT} to provide the support for multiple scheduling policies at the same time, as only one plugin at a time can be loaded in the system.

Another very similar approach is represented by the RealtimeKit Library (RTKit) [17], which is a D-Bus system service that can be used to request user processes or threads to be executed with the SCHED_RR scheduling policy. This service however does not provide any real-time guarantee by itself, it only allows user applications to be executed with a given set of real-time scheduling parameters.

Another project that we deem worth mentioning is also the Flexible Integrate Real-time Scheduling Technologies (FIRST) Scheduling Framework (FSF) [18], which is an operating system-independent framework that organizes a number of scheduling algorithms to work in cooperation (including both fixed priority (FP) and dynamic priority (DP) scheduling algorithms) in a hierarchical scheduling architecture. This framework also relies on a set of reservation techniques to provide temporal isolation among real-time tasks and applications can establish with the system a contract so that they will be provided a set of guarantees. FSF does not have a Linux implementation, but it has been implemented both for SHARK [19] and MaRTE [20] operating systems.

FRSH/FORB [21] is a middleware based on CORBA that

lets real-time applications avail of reservation scheduling across different physical resources, such as CPUs, disks and network interfaces, made available through proper kernel-level extensions to the Linux OS, such as the AQuoSA architecture [22] supporting adaptive CPU reservations and real-time extensions [23] for wireless communications compatible with the IEEE 802.11 standard series. FRSH/FORB has also been extended [24] with a transactional API for handling multi-resource reservations in a distributed system.

Finally, the ExSched project [25] is an extensible framework that aims to support real-time applications over multiple operating systems. This framework is composed of a kernel module and a set of plugins that can be chosen by the system administrator. The goal of ExSched is to provide a unified scheduler interface that can be used to implement different schedulers (each with its own plugin implementation) without patching and recompiling the underlying operating system. However, this functionality is achieved with a great cost in terms of performance: for example, their implementation of an EDF scheduler plugin introduces an overhead on the system that is about 180% in the worst case with respect to SCHED_DEADLINE implementation on Linux [25]. Furthermore, applications must be aware of their exact timing parameters (task period, worst case execution time, etc.) to be effectively used with ExSched.

Most of the solutions illustrated above rely on either fixed-priority or EDF/CBS scheduling, with applications requiring from the OS some specific policy and its parameters. This work aims to support heterogeneous sets of applications with different real-time characteristics and requirements, letting applications declare to the OS just what they know about their timing characteristics, leaving the OS free to use and configure the available OS/kernel schedulers so to make an optimal use of an underlying multi-core platform. The framework that is presented in this work is inspired to the concept architecture appeared in [26], which to the best of our knowledge has never been actually implemented.

III. Proposed Framework

In this section we present the declarative framework that we realized to ease access to real-time scheduling policies on Linux. The main focus of this framework is to provide real-time applications with an abstraction level that can be used to declare a set of scheduling parameters that shall be associated with each real-time task. From these parameters, the framework takes care of selecting the most proper scheduling technique and configuring actual scheduling parameters of the Linux thread associated to each task specification.

The design of this framework takes into account the possibility to port its implementation onto different POSIX-compliant operating systems other than Linux, while at the same time exposing at user-level an interface that can be used to develop complex real-time and multimedia applications. As it will be shown later, this framework can be installed over an unmodified kernel to provide unprivileged users with the capability to run applications that rely on the framework's functionality to meet their real-time requirements.

To accomplish these goals, this framework is not designed to be part of the Linux kernel, but it is composed of a *shared library* that applications can use to declare their requirements to a centralized authority. This is a *daemon* running with superuser privileges, which is in charge of managing all real-time applications in the system. With this solution, applications running without any particular privilege can simply request the daemon to set their own scheduling parameters.

The framework has been purposely developed to provide an API which is completely independent from the scheduling algorithm and policies that are actually used to meet the demands of each application. As it will be shown later in more details, the framework specifies an API that can be used to provide actual scheduling services in the form of *plugins*, which can be chosen at deployment time via a simple configuration file by the system administrator. This approach has been taken to provide a mean for researchers and other developers to extend the functionality of the framework, developing scheduling algorithms and policies to be used via the generic user-level API provided by the framework itself. This approach can be used to play with various real-time scheduling policies on the Linux operating system. The framework can also be ported to other POSIX-compliant OSes.

Independently of the plugins loaded with the *daemon* at any time, the API provided by the *shared library* can be used inside applications to declare a set of real-time parameters to be used by real-time tasks. For each request, the framework responds indicating whether it can be accepted by one of the plugins or not. On acceptance, a single execution flow—i.e. a POSIX thread—can be dynamically attached to the set of accepted scheduling parameters. After this operation, the framework sets the actual scheduling parameters of the attached thread accordingly, depending on which plugin accepted it; for example, a task that has been accepted by a plugin that relies on SCHED_DEADLINE sets up a CPU reservation for the thread. In case a request is rejected instead, it can be re-submitted after relaxing some of the real-time parameters.

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

The real-time parameters that can be declared by 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 *declarative* approach that characterizes this framework allows applications to specify from none to all of these parameters for each task; each plugin documentation specifies which parameters are mandatory, which are desired, but not

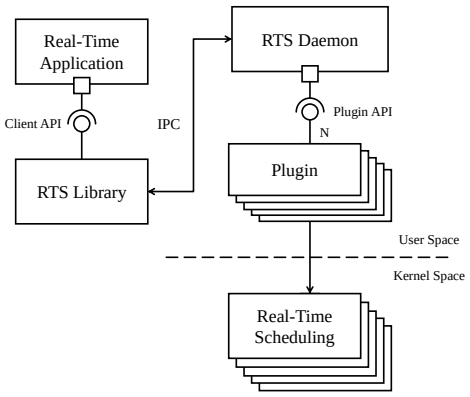

Figure 1. Architectural overview of the framework.

mandatory, and which parameters are not used. Hence, the set of real-time parameters specified for each task are used to automatically determine which plugin is used to schedule each task. If at least one plugin accepts a task, the task is considered as part of the scheduling task set and its parameters are used by the selected plugin to generate the actual parameters that are used to schedule the associated POSIX thread.

In addition, plugins can decide whether accept or reject tasks depending on the result of some admission control policy. This feature is optional and checks performed by each plugin shall test only for necessary conditions—i.e. failure, acceptance of the new task will inevitably lead to system unschedulability. Tasks can specify whether they want to bypass this admission control test upon task declaration.

An accepted task can later change its parameters without disrupting any of the other accepted tasks. This operation is atomic—i.e. multiple parameters can be changed atomically—and the plugin that is selected to schedule a task may change following this operation, if successful. If this change is not accepted, the task will maintain the scheduling parameters that were accepted last by the framework. This 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 for their worst case execution time: in this case, Q is interpreted as the *minimum runtime* requested by the task, while the second value is a *desired runtime* Q^d (higher than Q), which may be accepted by the system if enough resources are available. However, the system may be free to assign any *accepted runtime* $Q^a \in [Q, Q^d]$. If this functionality is used, each task can query at any time the accepted runtime by the system and plan its own execution accordingly, enabling or disabling optional paths in the execution flow if possible. Once a value for the *accepted runtime* is selected by the framework, it cannot change in the future without another explicit request by the task itself.

A. Architecture Overview

The architecture of the framework is depicted in Figure 1, where the relationships among the main framework components are illustrated. The most important component of the whole architecture is the central decision authority, the *RTS Daemon*: this component is in charge of connecting and coordinating all interactions among individual applications and the scheduling algorithms that are loaded by the framework as separated plugins. This is accomplished by providing each application with a shared library that can be used to communicate with the *RTS Daemon*. This library is called *RTS Library* and it provides the set of APIs described in section III-B used to declare the scheduling parameters associated to each real-time task. The parameters are then forwarded to the *RTS Daemon* via an Inter Process Communication (IPC) mechanism, namely a UNIX socket connection.

The received information will then be delegated to the plugins that are currently loaded along with the *RTS Daemon* in execution. This kind of architecture was designed to support various scheduling algorithms, each implemented by a corresponding plugin. In particular, each plugin associated to a specific scheduling algorithm will analyze the current task set and the requested parameters for the new task and will decide whether to accept it or not into the task set scheduled with that algorithm. In case the plugin embeds an admission criterion, acceptance can depend also on how many CPUs are currently managed by each plugin, which can be customized through the framework configuration file (see later). If at least one algorithm will accept the task, it is accepted and assigned to the highest-priority algorithm among the ones that can schedule that particular task set in the current conditions. The following sections provide further details for each component.

B. RTS Library API

Applications that want to leverage the real-time capabilities of a Linux system using the provided framework will communicate with the *RTS Daemon* through a well defined interface, which is implemented as a shared library and linked with the application binary. The main functions exposed by the *RTS Library* are illustrated in Table I. In addition, the provided API contains some utility functions to implement periodic task execution and to query the accepted parameters for each task.

Applications can declare the scheduling parameters of each real-time task by filling an instance of the opaque type `rts_params`, using the functions described in Table II. Once the `rts_params` object has been filled with the declared parameters, a new task admission request can be submitted to the *RTS Daemon* by executing

```
rts_result_t result = rts_task_create(
  struct rts_task* t, struct rts_params* p);
```

where `rts_task` is an opaque type that represents a task in the system. A connection between each application using the *RTS Library* and the *RTS Daemon* is performed automatically on submission of the first request and kept for further requests.

This function returns `RTS_OK` on success and `RTS_FAIL` on failure. If a desired runtime Q^d is supplied among the task parameters then the application can check the accepted runtime assigned to the task by calling the `rts_task_get_accepted_runtime` function with the same

Table I
MAIN FUNCTIONS EXPOSED BY THE RTS LIBRARY API.

Function	Description
`rts_task_create`	Performs task admission test and applies the specified `rts_params` to the new task.
`rts_task_change`	Performs a new task admission test with the specified `rts_params`; in case of failure the task maintains its old parameters.
`rts_task_release`	Releases a task, freeing its resources and detaching the attached POSIX thread, if any.
`rts_task_attach`	Attaches a POSIX thread id to the given task.
`rts_task_detach`	Detaches the POSIX thread assigned to a task; after this call, the thread runs with a non real-time priority and the task reference can then be attached to another POSIX thread.

Table II
LIST OF PARAMETERS THAT CAN BE DECLARED FOR EACH TASK.

Parameter	Unit	Getter / Setter
Runtime	µs	`rts_params_get_runtime` `rts_params_set_runtime`
Desired Runtime	µs	`rts_params_get_des_runtime` `rts_params_set_des_runtime`
Period	µs	`rts_params_get_period` `rts_params_set_period`
Relative Deadline	µs	`rts_params_get_deadline` `rts_params_set_deadline`
Priority	–	`rts_params_get_priority` `rts_params_set_priority`
Scheduling Plugin	–	`rts_params_set_scheduler` `rts_params_get_scheduler`
Ignore Admission Test	–	`rts_params_ignore_admission`

```
/* Task representation */
struct rts_task t = RTS_TASK_INIT;

/* Task parameters */
struct rts_params p = RTS_PARAM_INIT;

/* Set task parameters */
rts_param_set_period(&p, T_PERIOD);
rts_param_set_runtime(&p, T_RUNTIME);
rts_param_set_des_runtime(&p, T_DES_RUNTIME);
rts_param_set_deadline(&p, T_DEADLINE);

/* Test for admission */
if (rts_task_create(&t, &p) != RTS_OK)
  /* We can abort, or retry with different parameters */
  return;

/* On success we attach an execution flow to the
 * task specification */
rts_task_attach(&t, gettid());

/* Signals that a task begins its execution */
rts_task_start(&t);

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

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

  /* Suspend execution waiting for the next period */
  rts_task_wait_period(&t);
}

/* Cleanup */
rts_task_release(&t);
```

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

`rts_task` object as argument. In case of failure, the request can be repeated after changing some of the parameters.

Which parameter is mandatory and which is not is entirely dependent on the plugins loaded with the daemon at runtime. As it will be better shown later, the *RTS Daemon* will interrogate each plugin to ask them whether the requested parameters are suitable to execute a task with current system configuration and each algorithm can either accept or refuse a task depending on the supplied parameters. It is however possible to specify for a certain task the plugin that shall be used to schedule it. In that case, only the requested plugin will be interrogated for admission.

Once a task is accepted, an execution flow (a thread) can be associated with it using the `rts_task_attach` call, which instructs the *RTS Daemon* to apply the actual scheduling parameters to the given thread (identified via its Linux thread ID) to match the results of the admission test. After that point, the selected thread will run as a real-time task and will be scheduled according to the rules implemented in the plugin automatically selected for it.

For periodic tasks, the library provides some additional functions to mark the beginning of the first period of execution of the real-time task (`rts_task_start`), as well as a call that can be used to suspend task execution waiting for the next activation point (`rts_task_wait_period`).

The body of a sample thread that uses the API described in this section is shown in listing 1.

C. RTS Daemon

The *RTS Daemon* is in charge of forwarding each request received by the various applications to the right plugin, each implementing a different scheduling strategy. The *RTS Daemon* does not interact directly with the Linux kernel to satisfy the requests received by each application. Instead each plugin shall implement its own scheduling policy on top of the capability of the Linux kernel.

The list of plugins that are loaded at daemon initialization time is provided by the system administrator via a simple configuration file: this file is also used to assign each plugin a different priority and a range of POSIX priorities that could be used by the plugin itself when configuring the parameters of Linux threads assigned to it (see section III-D).

Once a task request is received, the daemon will proceed to interrogate each plugin, following the priorities provided in the configuration file, to check whether it is possible to satisfy that request by using the selected algorithm. The API that each plugin should implement is illustrated in section III-D.

Each plugin can reply that it is perfectly capable to satisfy the request as it is (`RTS_OK`), that the request satisfies its mandatory requirements even if some recommended parameters are missing (`RTS_PARTIAL`), or that it is not suitable to satisfy the request, either because some mandatory parameters

are missing or because some necessary admission test resulted in a failure (RTS_NO). Among all the responses, the daemon will select the plugin with the highest priority (specified by the user by means of a configuration file, see section III-D) that replied with a RTS_OK value; if no plugin is found with this criterion, the task is delivered to the plugin with the highest priority that replied RTS_PARTIAL and if none can be found then the request is denied. Finally, a response is sent back to the requesting application.

D. RTS Plugins API

The structure of the *RTS Daemon* takes advantage of a plugin-based and modular architecture. Each plugin must implement a set of functions that are used by the *RTS Daemon* to dispatch client requests. Each plugin will represent 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 kernel to schedule the provided real-time tasks.

Each plugin may implement an admission control mechanism that will be used to check whether a new task can be admitted to the current task set or not. This test, if implemented, shall be performed again each time a real-time task will request a change to its current set of parameters. In addition, a plugin can signal that the set of parameters provided by the client cannot be used by that particular plugin to schedule a task.

Table III shows the main functions exposed by each plugin to exchange data with the *RTS Daemon*. Among the plugins that responded at least RTS_PARTIAL to rts_plg_task_accept or rts_plg_task_change, the daemon selects one plugin to assign the given real-time task and it signals the selected plugin via the rts_plg_task_schedule function. The plugin then assigns the real-time scheduling parameters to the POSIX thread associated with that real-time task specification, if any, until either the task is assigned to another plugin after a change in 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 III and is distributed as a .so file that will be loaded by the *RTS Daemon* on start up. These plugins operate at user-space level and typically they can be executed on top of an unmodified Linux kernel. In case a plugin relies on the features introduced by a specific kernel module, the plugin can load it during the initialization phase of the *RTS Daemon*. The set of plugins that shall be loaded, as well as the pool of POSIX real-time priorities that shall be used by each plugin to schedule assigned tasks, is specified by the system administrator via a configuration file. The format of the file is similar to the example shown in listing 2: each line specifies the name of a plugin to be loaded and a few parameters for each plugin. These parameters are, in order, the range of POSIX real-time priorities and the list of CPUs that the plugin can use to schedule tasks assigned to it, although task allocation to CPUs depends on the implementation of each plugin.

Table III
MAIN FUNCTIONS EXPOSED BY EACH PLUGIN TO THE RTS DAEMON.

Function	Description
rts_plg_task_accept	Performs a new task admission test with the specified rts_params. The plugin may refuse to schedule a task if the (optional) admission test fails or the supplied parameters are not suitable to generate a schedule.
rts_plg_task_change	Performs a new task admission test with the specified rts_params.
rts_plg_task_release	Notifies the plugin that a task left the task set. Plugins shall perform here cleanup operations.
rts_plg_task_schedule	The *RTS Daemon* indicates to the plugin that the given task has been assigned to it and that it should manage its scheduling parameters accordingly.
rts_plg_task_attach	The *RTS Daemon* instructs the plugin to set the scheduling parameters of the given POSIX thread to match the corresponding real-time task parameters.
rts_plg_task_detach	The *RTS Daemon* instructs the plugin to change the POSIX thread priority to a normal one and that it is no longer associated with the given task.

```
EDF 100-100 0
RM  50-99   1,2
RR  1-49    1,2
FP  1-99    3-7
```

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

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

IV. IMPLEMENTATION

The framework's implementation reflects the architecture illustrated in section III-A. In this section we summarize the main characteristics of each plugin that we developed to test the functionality of our implementation when multiple scheduling algorithms are loaded at the same time.

A. EDF Plugin

This plugin implements the EDF scheduling algorithm, which is well known to be optimum for single processor systems [27]. In particular, we implemented a fully-partitioned version of EDF applying a worst-fit task allocation strategy among the CPU cores specified via the *RTS Daemon* configuration file. Its implementation leverages the EDF implementation offered by Linux mainline kernel since version 3.14 [28] via the SCHED_DEADLINE scheduling class; this class assigns each task its own reservation to be run into, based on its runtime Q and period T. For this reason, an application that would like to schedule a real-time task through this plugin shall declare at least the period and the runtime of the task, otherwise the task cannot be assigned to the EDF plugin. The

optional deadline parameter D can also be specified, otherwise it defaults to the same value as the period T. The plugin calculates the *utilization* of each task τ_i in the system that declared both its runtime and period, which is defined as the ratio between $U_i = Q_i/min\{T_i, D_i\}$.

This plugin implements also a simple utilization-based task admission test. Since each task can only be assigned to one core, the least loaded core is selected and then an admission test is performed to check whether the admission of the new task into the current task set leads the system to an overloaded condition. The load of each core k, given the set of tasks assigned to that core Γ_k, is defined as the sum of the utilizations of all the tasks belonging to Γ_k, that is $U_k = \sum_{\tau_i \in \Gamma_k} U_i = \sum_{\tau_i \in \Gamma_k} Q_i/min\{T_i, D_i\}$.

The plugin currently disables the in-kernel necessary G-EDF test performed by SCHED_DEADLINE, but it sets task affinities and it implements a sufficient schedulability test for each core k ensuring U_k is less than or equal to 1 [27] (the value is customizable). Hence, given the least loaded core $\bar{k}$, this plugin accepts a new task τ_j to be scheduled if the following condition holds true:

$$U_{\bar{k}} + \frac{Q_j}{\min\{T_j, D_j\}} \leq 1 \tag{1}$$

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

If a desired runtime Q_j^d has been 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:

$$Q_j^a = max(Q_j, min(Q_j^d, (1 - U_{\bar{k}}) \cdot min\{T_j, D_j\})) \tag{2}$$

In case the acceptance condition is not satisfied, but the task has been accepted by this plugin anyway, then its accepted runtime is equal to its minimum runtime Q_j.

B. RM Plugin

This plugin implements the Rate Monotonic (RM) scheduling algorithm, which is well known to be optimum for single processor systems among FP scheduling algorithms [27]. In particular, it implements a fully-partitioned version of RM applying a worst-fit task allocation strategy on top of the POSIX SCHED_FIFO scheduling policy among the CPU cores specified via the *RTS Daemon* configuration file.

The only required parameter that a real-time task shall declare to be eligible to be scheduled with this plugin is its period T. For this reason, it is possible that for some tasks this plugin might not be aware of their runtime Q, hence it is not possible to perform a proper admission test for some tasks that could be assigned to this plugin. The adopted strategy is to apply the well-known single-CPU FP utilization test at least for all tasks that specify both runtime and period parameters, unless otherwise specified by the requesting client.

Once a task is assigned to this plugin by the *RTS Daemon*, a core k is selected to schedule that task using a worst-fit allocation strategy. Given the set of tasks assigned to that core Γ_k, this plugin assigns each task a priority that is inversely proportional to the their period:

$$P_i \propto 1/T_i \qquad \forall\ i \in \Gamma_k \tag{3}$$

The calculated priority for each task P_i, which is within the range of POSIX priorities assigned to this plugin via the *RTS Daemon* configuration file, is then used to schedule tasks using SCHED_FIFO. Future versions of this plugin will let applications choose between FIFO and RR scheduling policies.

C. FP and RR Plugins

The FP and Round Robin (RR) plugins serve as wrappers to expose underlying POSIX functionality to applications that use this framework. They respectively provide access to SCHED_FIFO and SCHED_RR scheduling policies and as such the only required parameter that shall be specified to be accepted by either of these plugins is the desired POSIX priority of the task P. For this reason, no admission test is performed when submitting a task to these plugins, although a task may still specify other parameters that may be considered by other plugins' admission tests. 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 least number of assigned tasks.

Notice that the priority requested via the *RTS Library* API may differ from the one actually used by either of these plugins to schedule the task, since the range of priorities that each plugin might select may be smaller than the normal range of POSIX priorities. In this situation, the ordering of the distinct priorities that have been requested for each real-time task is maintained when assigning actual POSIX priorities. However, if the destination range of priorities is smaller than the number of distinct priorities that have been requested some tasks may receive the same POSIX priority even if they originally requested two distinct ones.

V. Performance Evaluation

In this section we report experimental results showing the overhead introduced by the framework when declaring a new real-time task or modifying the parameters of an existing one. In particular, we will show that the framework introduces only two types of overhead when used: the first depends on the IPC mechanism that is used to exchange data between the clients and the *RTS Daemon* component and the second depends on the type of operations performed by each plugin on task acceptance.

Experiments were performed on a desktop platform equipped with a multi-core Intel Core i5-2300. The machine is configured with an Ubuntu 18.04.1 LTS distribution, Linux kernel version 4.15.0. To maximize the reproducibility of results, our tests have been run disabling hyperthreads, disabling CPU frequency scaling (governor set to `performance` and frequency set to 1.6 GHz) and Turbo Boost disabled.

The benchmarking application is a single-threaded process that performs multiple requests to the *RTS Daemon*, each time declaring an additional real-time task to be added to the current

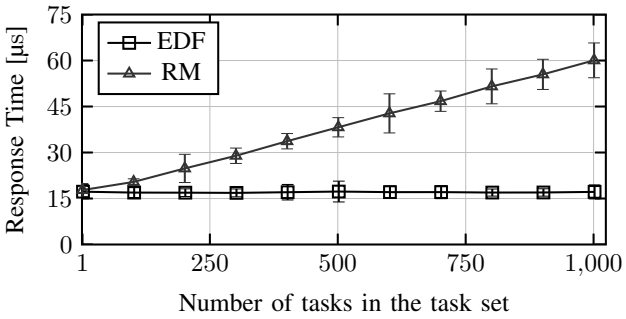

Figure 2. Average response time of a new real-time task allocation request depending on the number of tasks in the current task set. Error bars represent standard deviation over 500 experiments.

task set. Both the benchmarking application and the *RTS Daemon* were set to run with the highest POSIX real-time priority (99) and each was pinned on a separate core. Throttling of real-time applications by the kernel was disabled by setting `/proc/sys/kernel/sched_rt_runtime_us` to -1. Finally to maximize the precision of our time measurements we used the Time Stamp Counter (TSC) register to track elapsed time within the benchmarking application.

The application performs several real-time task declarations (without attaching any actual thread to accepted tasks) over and over, varying the number of tasks in the task set from 1 to 1024, to evaluate whether the cost of the task admission tests performed by each plugin depends on the number of tasks present in the current task set at any time. Experiments were repeated 500 times each, using a different set of 1024 random real-time task parameters for each run.

A. Results

Figure 2 shows experimental results. The figure compares the cost in terms of average response time of a new task allocation request for each plugin, depending on the number of tasks currently present in the accepted task set. From the plot it is clear that while the response time of the EDF plugin does not depend on the number of tasks, the RM plugin has an increasing cost with respect to the task set size. This can be attributed to the operations performed by the RM plugin to update the POSIX priorities assigned to real-time tasks each time a new task is introduced. For both solutions, the minimum average round-trip time measured is about 16 µs, which is clearly the cost of the UNIX socket used to exchange data between the benchmarking application and the *RTS Daemon*.

Notice that this cost must be paid only when declaring a new task or when changing the real-time parameters of an already declared task, while during normal task scheduling operations the overhead introduced is zero with respect to using directly the scheduling policies exposed by the Linux kernel, since they are the same as used by the framework to schedule each thread. This is an advantage with respect to other similar frameworks reviewed in section II, such as ExSched, which increases the overhead introduced by the scheduler when used. However developers may want to avoid sending too many parameters change requests to the framework on the critical path of a real-time thread, since the cost of each request is non-negligible.

VI. Conclusions

This work describes the architecture and implementation of a novel framework that aims at simplifying access to real-time capabilities of the Linux kernel, adopting a component-based system design, using a declarative API model. The realization of this framework is motivated by the recent advances in the kernel features for real-time tasks in Linux, that need to be followed by corresponding advancements in associated middleware and user API services.

A. Future work

We plan to continue the development of the framework to improve it and extend it to support more features. Energy efficiency is one of the directions in which we intend to improve the framework's implementation. On architectures that support power management techniques like Dynamic Voltage and Frequency Scaling (DVFS), computation times may vary depending on the frequency of the CPUs or (for architectures like ARM big.LITTLE) by the type of the CPU core selected to run each task. To improve the energy awareness of the framework, a possible extension of this work could use architecture and frequency-independent computation time specifications that enable more energy-efficient policies for task allocation. Also, a monitoring system might easily add to the RTS Daemon adaptive capabilities to self-detect or correct some of the managed task parameters and realize adaptive strategies that can make better use of the physical resources.

The current implementation neglects possible transients due to new tasks entering or existing tasks leaving the system. In these cases, mode-change protocols should be added within the framework or plugins, using techniques such as [29].

The use of thread ids by the *RTS Daemon* could lead plugins to mistakenly change the 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* [30], that have been added recently in the Linux kernel for cases like this. Also, the API can be extended to accept additional parameters, like preferred or mandatory CPU affinity constraints, or blocking times to be considered by some more advanced admission test in the plugins.

Finally, we plan to enrich the *RTS Daemon* configuration to include an access control model for the features provided by the framework. The current implementation relies on a UNIX socket for IPC communication (see section III-A), so permission bits and ACLs can be leveraged to limit access to the communication channel only to specific users/groups. Direct access to real-time features of the system by unprivileged users (thus enforcing the use of the framework) can be inhibited via existing features of Linux like `limits.conf`, while access from root processes remains unrestricted. In the future, the framework will allow system administrators to configure per-user or per-application security policies.

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] K. Obenland, "The use of POSIX in real-time systems, assessing its effectiveness and performance," 01 2000.

[3] S. Rostedt and D. V. Hart, "Internals of the RT patch," in *Proceedings of the Linux symposium*, vol. 2, 2007, pp. 161–172.

[4] F. Reghenzani, G. Massari, and W. Fornaciari, "The real-time Linux kernel: A survey on Preempt_RT," *ACM Computing Surveys*, vol. 52, pp. 1–36, 02 2019.

[5] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli, "Deadline scheduling in the Linux kernel," *Software: Practice and Experience*, vol. 46, no. 6, pp. 821–839, 2016. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2335

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

[7] Ayers and B. V. Yodaiken, "Introducing real-time Linux," *Linux J.*, vol. 1997, no. 34es, p. 5–es, Feb. 1997.

[8] L. Dozio and P. Mantegazza, "Real time distributed control systems using RTAI," in *Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2003.*, May 2003, pp. 11–18.

[9] P. Gerum, "Xenomai–implementing a RTOS emulation framework on GNU/Linux."

[10] S. Han and H.-W. Jin, "Kernel-level arinc 653 partitioning for linux," *Proceedings of the ACM Symposium on Applied Computing*, 03 2012.

[11] ARINC, "Avionics application software standard interface part 1 – required services," 2015.

[12] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and 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 (Cat. No.98TB100245)*, June 1998, pp. 112–119.

[13] S. Brandt, G. Nutt, T. Berk, and J. Mankovich, "A dynamic quality of service middleware agent for mediating application resource usage," in *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, Dec 1998, pp. 307–317.

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

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

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

[17] RealtimeKit (RTKit) GitHub Repository. Accessed January 10, 2020. [Online]. Available: https://github.com/heftig/rtkit

[18] 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, and M. Trimarchi, "FSF: A real-time scheduling architecture framework," in *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, April 2006, pp. 113–124.

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

[20] M. Aldea-Rivas and M. Harbour, "MaRTE OS: An Ada kernel for real-time embedded applications," vol. 2043, 05 2001, pp. 305–316.

[21] M. Sojka, P. Píša, D. Faggioli, T. Cucinotta, F. Checconi, Z. Hanzálek, and G. Lipari, "Modular software architecture for flexible reservation mechanisms on heterogeneous resources," *Journal of Systems Architecture*, vol. 57, no. 4, pp. 366 – 382, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762111000282

[22] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "Aquosa—adaptive quality of service architecture," *Software: Practice and Experience*, vol. 39, no. 1, pp. 1–31, 2009. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.883

[23] M. Sojka, M. Molnar, and Z. Hanzalek, "Experiments for real-time communication contracts in ieee 802.11e edca networks," in *2008 IEEE International Workshop on Factory Communication Systems*, May 2008, pp. 89–92.

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

[25] M. Åsberg, T. Nolte, S. Kato, and 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*, Aug 2012, pp. 240–249.

[26] T. Cucinotta, D. Giani, D. Faggioli, and F. Checconi, "Effective real-time computing on Linux," in 12*th Real-Time Linux Workshop*, 2010.

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

[28] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino, "An EDF scheduling class for the Linux kernel," in *11th Real-Time Linux Workshop*, 2009.

[29] D. Casini, A. Biondi, and G. Buttazzo, "Handling transients of dynamic real-time workload under EDF scheduling," *IEEE Transactions on Computers*, vol. 68, no. 6, pp. 820–835, 2018.

[30] J. Corbet. (2019, July) Completing the pidfd API. Accessed January 10, 2020. [Online]. Available: https://lwn.net/Articles/794707