Effective Real-Time Computing on Linux

Tommaso Cucinotta, Dhaval Giani, Dario Faggioli, Fabio Checconi

Real-Time Systems Laboratory (ReTiS)
Centre of Excellence for Information, Communication and Perception Engineering (CEIICP)
Scuola Superiore Sant'Anna, Pisa, Italy
{t.cucinotta,d.giani,d.faggioli,f.checconi}@sssup.it

***Abstract*—In this paper we present an architecture design for supporting real time computing on Linux. This architecture focuses on improving the usability of real time capabilities for applications by providing a unified Application Programming Interface. Applications can therefore use it without having to know exactly what the underlying scheduling algorithm is. Still, the real-time computing capabilities of the platform may be exploited to the maximum extent.**

The main aim of the paper is to gather feedback from the community about the design and directions for development.

I. Introduction

General-Purpose Operating Systems are being increasingly considered for realizing complex embedded, time-sensitive and distributed systems. Example application domains are constituted by multimedia applications, digital A/V processing systems, interactive distributed collaboration systems, real-time control applications with timing constraints that are not excessively strict, etc. The wide variety of libraries, tools, middle-ware and applications available nowadays on a GPOS like Linux, as well as the rich set of supported multimedia peripherals including acquisition and playback devices and memory cards, and the wide variety of supported networking devices and protocols, make it the ideal development environment for these types of systems. However, these soft real-time applications pose challenging requirements the Operating System in terms of the ability to provide proper guarantees on the timely execution of the hosted processes.

Focusing on Linux, there have been significant advances in the last few years in terms of the capabilities required for precise timing of applications. From the years of the 2.4 kernel series, where kernel code was not preemptable, the current kernel supports fully preemptable kernel sections. The High Resolution Timers subsystem, now integrated into the mainline kernel, allows for a nanosecond-precision time accounting and posting of timers. The increased usage of advanced synchronization primitives such as Read-Copy-Update optimizes the access to shared data structures across multiple cores and processors on SMP systems. The kernel has an almost complete support for the POSIX real-time extensions concerning management of clocks, timers, signals and process scheduling policies. The `PREEMPT_RT` [1] project of the kernel re-engineers the interrupt management subsystem. The interrupt handlers run in dedicated real-time kernel threads with the actual kernel logic which runs in interrupt context limiting itself to just waking up the appropriate handler thread. This dramatically improves the responsiveness of the Operating System to external events, at a cost of a slight decrease in overall system throughput. Finally, the current *real-time throttling*[2] mechanism and its integration within the `cgroup` framework allows for a minimum degree of temporal isolation across concurrently running real-time tasks or groups of tasks.

However, the very last aspect is somewhat still incomplete, at least in the official mainline kernel version. Despite the growing need for a proper mechanism enabling *temporal isolation*, to provide scheduling guarantees to concurrently running real time tasks in the system, the kernel still lacks it. The set of applications mentioned earlier as well as the multimedia application domain would greatly benefit by task scheduling mechanisms at the kernel level with precise real-time guarantees. For example, with priority-based scheduling as mandated by POSIX and implemented in the kernel a high-priority task may indefinitely delay lower-priority ones, hindering the possibility to provide proper guarantees to individual real-time tasks. While this may not constitute a problem for simple embedded real-time system where nearly everything is under control of the designer, for more complex systems the lack of a proper temporal encapsulation support by the kernel is an important issue. The POSIX Sporadic Server scheduling policy would address such issues, but unfortunately it is not yet implemented in the mainline Linux kernel (even though there exists a patch [1] supporting it). The real-time throttling mechanism mentioned earlier partially mitigates such a lack of feature providing the possibility to associate groups of tasks with certain scheduling guarantees. However, the granularity over the time period these guarantees are provided is solely system-wide (and it defaults to 1 second). Also, the current code base, being based on the `cgroup` interface, is oriented towards static configuration of the scheduling parameters, rather than a much more dynamic exploitation of it as it would be required, for example, by multimedia and adaptive real-time

The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7 under grant agreement n. 214777 "IRMOS—Interactive Realtime Multimedia Applications on Service Oriented Infrastructures".

[1]More information is available at:
https://rt.wiki.kernel.org/index.php/Main_Page.

[2]http://www.mjmwired.net/kernel/Documentation/scheduler/sched-rt-group.txt

applications.

A. Paper Contributions

In this paper, a software architecture, designed with the purpose of supporting real-time computing on the Linux Operating System, has been described. A multitude of projects focussing on enriching Linux with proper real-time task scheduling policies already exist (see Section II for an overview). In this paper the discussion builds over the experience gained while participating in some of these projects, yet what is proposed is something quite different. The focus is on the requirements posed by complex real-time and multimedia applications, the proper level of abstraction which needs to be exposed at the Application/OS interface as well as the minimum set of core functionality needed for the purpose of building higher level complex and possibly distributed infrastructures for real-time systems.

B. Paper Structure

This paper is organized as follows. In Section II we overview related work in the literature, focusing on architectures and APIs for real-time scheduling. In Section III we sketch out a set of requirements posed by real-time and multimedia applications on scheduling services provided by the OS. In Section IV, we present our architecture design aiming to fulfill those requirements. Finally, in Section V we present directions for future work and draw conclusions.

II. Related Work

In this section, related work in the area of real-time support for general-purpose operating systems (GPOS) is presented. Various modifications of GPOSes have appeared in the literature for supporting real-time scheduling policies at the kernel-level, for various types of resources. Various GPOS kernels exist that are compliant with the POSIX real-time extensions [2]. However, most of the implementations limit themselves to Fixed-Priority scheduling, sometimes with the addition of high-resolution timers and the Priority Inheritance protocol [3] for avoiding Priority Inversion. Also, one key feature which is usually not implemented is the *temporal isolation* property [4], such as provided by the Sporadic Server [1] scheduling policy. Without such a mechanism, a higher priority task runs undisturbed until it blocks, independently of the computation time that may have been considered at system analysis/design time. This results in the potential disruption of the guarantees offered to lower priority tasks. Therefore, such approaches are suitable for the traditional hard real-time settings where everything running into the system has been thoroughly checked, if not formally proved.

For real-time scheduling of the CPU, hard real-time modifications to the Linux kernel have been proposed, like RT-Linux[3], proposed by Yodaiken et al. [5] and RTAI[4], proposed by Mantegazza et al. [6] or Xenomai[5], by Gerum et al. [7]. In these approaches, a real-time micro-kernel layer is added between the real hardware and the Linux OS, which runs as the background/idle activity whenever there are no hard real-time tasks active in the system. This allows for respecting the very tight timing constraints (microsecond-level) typical of industrial automation and robotic applications. However, (hard) real-time applications usually need to be written in a special way, for example they are typically required to be written as kernel modules. Therefore, these hardly constitute solutions suitable for the large class of multimedia and interactive applications, which would greatly benefit from real-time scheduling policies.

In order to overcome these limitations, other approaches targeted explicitly soft real-time applications, by adding a real-time scheduling policy as an extension to a GPOS kernel itself, typically comprising a temporal isolation mechanism. Such an approach allows for the coexistence of soft real-time and best-effort applications, all within a GPOS kernel with potentially long non-preemptive sections, what leads to the impossibility to provide hard real-time guarantees. However, for soft real-time applications like multimedia ones, that approaches allowed for great enhancements achieved on the side of the predictability in the temporal behavior exhibited by applications, which resulted in significant improvements in the QoS experienced by users. An overview of these approaches has been carried out by Gopalan in 2001 [8]. Just to mention a few, remarkable works are the ones for adding resource reservations [9] to Microsoft Windows NT by Jones [10] (Rialto/NT), and the ones by Rajkumar et al. in the Linux/RK project[11], a modification to the Linux kernel inspired by prior work of the same authors on RT-Mach. More recently, Rajkumar et al. proposed Distributed Resource Kernels [12], an extension of Linux/RK adding support for distributed real-time applications directly inside the kernel, for improved efficiency. Linux/RK is geared towards a model of non-modifications to applications, therefore the benefits it can carry are somewhat limited. An effort on portability of a real-time scheduler across various Operating Systems (from Microsoft, Unix and Linux families), is constituted by the DSRT scheduler[6] by Nahrstedt et al. [13] and the GRACE [14] series of architectures.

Also, soft real-time schedulers for Linux have been investigated and implemented in the context of various European Projects like the CBS [15] implementation on Linux developed during the OCERA Project[7], and its subsequent evolution, the AQuoSA [16] scheduler for Linux, developed during

[3]More information is available at http://www.rtlinuxfree.com.

[4]More information is available at http://www.rtai.org.

[5]More information available at http://www.xenomai.org

[6]More information is available at: http://cairo.cs.uiuc.edu/software/DSRT-2/dsrt-2.html.

[7]Open Components for Embedded Real-time Applications (OCERA), European Project n.IST-2001-35102. More information is available at: http://www.ocera.org.

the FRESCOR Project[8]. More recently, the IRMOS Project[9] is also investigating on the use of real-time scheduling on high-performance Linux machines, with a strong focus on virtualized distributed real-time applications. In the context of IRMOS, the most recent real-time extensions to the Linux scheduler have been proposed: Faggioli et al. proposed a POSIX compliant implementation [1] of the FP-based Sporadic Server algorithm; Checconi et al. designed a novel hierarchical hybrid scheduling framework [17], based on a combination of partitioned EDF and global FP, designed so as to fit as much as possible (and impact as less as possible) in the current real-time scheduling class code base. A similar work is being done in the context of the ACTORS Project[10], but with a focus on single-threaded applications scheduled by global EDF on embedded multi-core systems [18]. Finally, the LITMUS-RT Project[11] is noteworthy to mention as it embeds an implementation on Linux of the Pfair [19] algorithm, which is theoretically optimum for multi-processor systems, along with other EDF-based scheduling algorithms for multi-processors (both partitioned and global).

Also, prior works exist that integrate real-time scheduling of heterogeneous resources and an architecture for their management for real-time applications, like the one by Stankovic et al. [20], or Hola QoS [21] by Valls et al. The latter is an architecture specifically tied to the needs of consumer electronics embedded multimedia systems, providing flexible resource management and adaptability. The Eclipse/BSD [22] Project integrates real-time scheduling of CPU, network and disk access, and exposes to applications a file-system based user-space interface. More recently, Gopalan et al. [23] proposed MURALS, a distributed real-time architecture built upon TimeSys Linux[12], supporting real-time applications with end-to-end constraints making use of distributed heterogeneous resources, such as disks, CPUs and network links. The architecture embeds a global admission control scheme that takes into account the entire dependency graph of the application. The above mentioned Nahrstedt research group also worked on QualMan [24], a distributed real-time resource allocation architecture supporting also network, disk and memory allocation, with prototype implementation on the Solaris OS.

Other approaches exist for higher-level QoS control for time-sensitive applications, such as TAO [25], Real-Time CORBA [26], Quality Objects (QuO) [27, 28], and HiDRA [29]. These constitute higher-level middle-ware components that may represent valuable approaches to be built on top of the architecture described in this paper.

[8]Framework for Real-Time Embedded Systems based on Contracts (FRESCOR), European Project n.FP6/2005/IST/5-034026. More information is available at: http://www.frescor.org.

[9]Interactive Real-Time Multimedia Applications on Service-Oriented Infrastructures, European Project FP7-214777. More information is available at http://www.irmosproject.eu.

[10]Adaptivity and Control of Resources in Embedded Systems, European Project n.216586. More information is available at http://www.actors-project.eu

[11]Linux Testbed for Multiprocessor Scheduling in Real-Time Systems ($LITMUS^{RT}$). More information is available at http://www.cs.unc.edu/~anderson/litmus-rt.

[12]More information at https://linuxlink.timesys.com.

III. Requirements

In this section we sketch out the fundamental requirements over which the architecture and the Application Programming Interface (API) for exposing the real-time scheduling functionality is built.

This discussion draws heavily from the requirements analysis documents from the FIRST, OCERA and the FRESCOR European Projects. Among these, the D-RA2 FRESCOR deliverable [30] is the most recent and relevant study. It is noteworthy to mention that the FRESCOR architecture has been designed with such a multitude of challenging goals as:

- support for multiple heterogeneous resources (CPU, disk, network, memory);
- support for transactional negotiation of groups of reservations, both at a single-node level, and in a distributed network;
- support for both hard and soft real-time systems;
- support for complex synchronization protocols for accessing shared resources;
- portability of the architecture and applications using it among heterogeneous operating systems, ranging from a hard real-time OS like Marte OS to a soft General-Purpose one like Linux;
- support for adaptive reservations and application-level QoS control
- support for QoS power-aware optimization of the system configuration.

On the other hand, in this paper we focus on a much smaller set of requirements, which we believe constitute a fundamental core that enables the possibility to build all other CPU reservation functionality mentioned above by means of higher-level software components. In fact, in the context of FRESCOR, it was possible to implement the extremely complex FRSH API [31] on Linux on top of the much simpler AQuoSA API[13]. Still, this API was missing fundamental features, such as support for multi-core systems and power management, which we address in this paper. Also we focus on exploiting increasingly available soft real-time enhancements in task scheduling on the Linux kernel. We also focus on how exactly to expose this functionality to the application by the means of a library and a software architecture which is planned to be implemented as detailed in Section IV.

The set of core requirements that we identified has been split into the following main categories:

- *core* requirements deal with the basic capability of reserving some amount of the available computing power to an application, in a sufficiently *abstract* way, so that application developers do not have to deal with how the system exactly provides guarantees;
- *adaptive scheduling* requirements deal with the possibility of realize feedback-scheduling loops for adapting the

[13]More information is available at http://aquosa.sourceforge.net.

allocation to the dynamic workload of the application, as needed by multimedia applications;

- *robustness and security* requirements deal with the ability of the OS to maintain stability and correct operation of the OS despite possible malicious applications trying to subvert the OS functionality exploiting the new real-time scheduling capabilities;
- *power management* requirements deal with the possibility to coupling power management logic with the available real-time features, provided that a proper coordination exists between them;
- *monitoring* requirements deal with the possibility to monitor the current state of the system with respect to the existing real-time applications and availability of resources.
- *hierarchical scheduling* requirements deal with the capability to handle nested groups of tasks, and to possibly associate a scheduler at each level or intermediate node of the hierarchy.

Now the individual requirements are described in more detail.

A. Core Requirements

K.1. The ability to submit a CPU reservation request independent of the scheduling algorithm being used by the kernel.

K.2. The ability to specify the minimum scheduling guarantees to be provided by the kernel, in terms of at least the following parameters (specification of the parameters is optional, unless explicitly stated otherwise):

   a) A budget Q and a period P which is mandatory. This provides a guarantee of (at least) Q time units every P time units.
   b) A relative deadline D (defaults to the same value as the period P). This provides a guarantee that the reservation will be granted Q time units within the first D time units, for every period of P time units.
   c) A maximum tolerance δ for the schedule precision i.e., the kernel is needed to actually grant to the reservation only $Q + / - \delta$ every specified period P.
   d) Unambiguous specification of the reference clock for P[14]

K.3. The ability to *attach* and *detach* a single flow of execution (single-threaded process or single thread) to/from the CPU reservation.

K.4. The ability to specify the *degree of parallelism* for the requested resource reservation.

K.5. The ability to query the system for available optional capabilities.

K.6. The ability to write multi-threaded applications that concurrently use the envisioned scheduling functionality.

K.7. The ability to specify a *default reservation* for resources available to POSIX real-time tasks and/or to non real-time tasks.

B. Adaptive Scheduling

A.1. The ability to dynamically change the reservation budget Q and/or period P while the attached task is running, without disrupting the scheduling guarantees for the other admitted tasks.

A.2. In case of required changes to both the budget and the period, a way to change them *atomically* is needed.

A.3. The ability to dynamically request a *desired budget* Q^d (possibly higher than Q), which may be assigned by the system if enough resources are available. However, the system may be free to assign any budget in the range $\left[Q, Q^d\right]$, because only the originally reserved budget Q should be subject to admission control.

A.4. The ability to query, for each period, the actually assigned budget, if the system supports specification of a desired budget.

A.5. The ability to query, during each period, the residual available budget.

C. Hierarchical Real-Time Scheduling

H.1. Optionally, the ability to attach additional threads to a resource reservation; in such cases:

   a) the default scheduling policy adopted among tasks belonging to the same reservation should be clearly stated;
   b) optionally, it should be configurable.

H.2. The possibility to manage nested resource reservations.

D. Robustness and Security Requirements

S.1. The ability to allow unprivileged users to exploit real time capabilities provided by the system.

S.2. Ability to configure per-user and per-group quotas

S.3. The ability consider the requested budget Q as a hard limit, in such a way that no more than that can be granted, unless explicitly requested.

S.4. The ability to define the lifetime of a resource reservation. This includes cases where a task dies prematurely as well.

S.5. The provision of an Access Control Model to restrict access to resource reservations.

E. Power Management Requirements

P.1. Power Management logic should be aware of the strict timing requirements of the running real-time tasks:

   a) Allow the specification of the reference CPU frequency which the budget Q specified in K.2a refers to, or implicitly refer to the current frequency.
   b) Allow the specification of individual budgets for available CPU frequencies.
   c) Possible exploitation of the knowledge about future wake-up times of real-time tasks so as to improve overall power saving.

[14]Such as the `MONOTONIC` or `REALTIME` clock implementations on Linux.

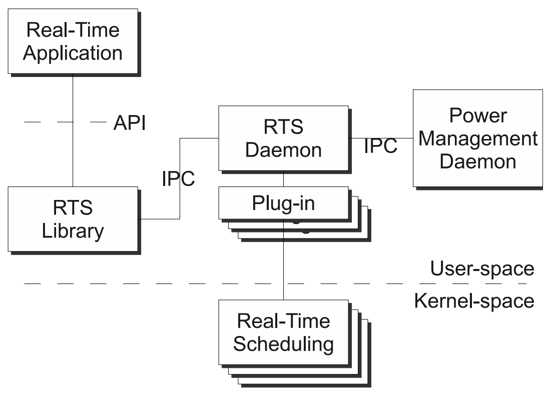

Figure 1: Proposed Architecture

F. Monitoring Requirements

M.1. The availability of a per-reservation counter for *budget overruns*.

M.2. The availability of a per-reservation counter for *deadline misses*.

M.3. The possibility to list all reservations active in the system, to inspect their parameters and the attached tasks (e.g., the AQuoSA system monitor).

M.4. The possibility to query the system about the overall *residual capacity*.

IV. Architectural Description

Figure 1 illustrates the main components of a possible implementation of the proposed framework. As can be seen, the key portion of the entire architecture is the central decision authority, the *RTS Daemon*.

The *RTS API*, described in Section IV-A, is the primary means an application has of communicating its real-time requirements to the framework. These requirements are then passed to the RTS Daemon, which in turn communicates requirements to the various plugins as needed.

It is possible for different scheduling models to run at the same time. The daemon shall prioritize between the different algorithms, and depending on the algorithm, admit the task to the appropriate model.

It is also possible for different adaptive approaches to exist while setting up the resource reservations. Similar to the case of the scheduling plugins, the daemon priorities and selects the appropriate model for the task.

Finally since the framework is power aware, any power management component must be RTS aware. The framework must have the ability to prevent some of the actions of the power management component, depending on the requirements of the application in the system In particular, CPU frequency and voltage scaling will affect the budget of the reservations, while at the same time, since the daemon has the knowledge of the periods of the applications, it is possible for it to provide some suggestions on whether the CPU should go into a deep idle state.

The following Sections give further details for each component.

A. Real-Time Scheduling API

Applications communicate with the framework through a well defined interface which is implemented as a shared library and linked with the application binary.

In order to request real-time scheduling guarantees to the system, an application needs to provide a set of parameters by means of the opaque `rts_params` type. Each one of the parameters enumerated in Section III may be set and retrieved by means of proper getters and setters, for example:

- `rts_set_param(rts_params *p, enum PARAM, void *value);`
- `rts_get_param(rts_params *p, enum PARAM, void *value);`

The reservation request is submitted to the framework by means of the function `rts_create_rsv()`

For the most frequently used parameters helpers such as `rts_set_period()`, `rts_get_budget()` are available.

The functions comprising this API can grouped in the following categories:

1) General Functions:
   - *Capability querying* functions, such as `rts_cap_query(enum CAP)`, to discover if a specific capability of the framework is available in the current configuration.
   - *Algorithm specific* functions, for setting additional parameters specific for a particular scheduling algorithm in order to optimize its performance.
2) Per-reservation Functions:
   - Functions providing reservation *setup* such as configuring, creating and destroying a reservation. `rts_set_param()`, `rts_get_params()` (as described above), `rts_create_rsv(rts_params *p)`, `rts_destroy_rsv(rts_params *p);`
   - Reservation *inquiry and modification* functions, query and modify the actual reservation parameters. `rts_get_rsv_params(rts_params *p)`, `rts_set_rsv_params(rts_params *p);`
   - Reservation *attachment* functions, which associate a thread to a reservation. `rts_attach_thread(tid t, rts_params *p);`

All the parameters specified as mandatory in section III must be setup for a reservation before trying create it. In addition, the application are free to specify the optional parameters.

Figure 2 shows a simple example using the proposed API.

B. Real-Time Scheduling Daemon

The RTS API acts as a gateway between the application and the RTS Daemon, which is the central decision authority of the framework. It is this component which is aware of the system state and interacts with the kernel, mapping application

```c
void RT_task(void)
{
  /* Parameters for reserving CPU */
  rts_params p;
  /* Identifier of the reservation, if accepted */
  rts_rsv rsv_id;

  if (!rts_cap_query(RTS_CAP_BUDGET))
    exit_err("notion of budget unsupported!");

  if (!rts_cap_query(RTS_CAP_REMAINING_BUDGET))
    exit_err("remaining budget retrieval unsupported!");

  rts_params_init(&p);

  /* 40 milliseconds period */
  rts_set_period(&p, 40000);
  /* 25 milliseconds budget */
  rts_set_budget(&p, 25000);

  if (rts_create_rsv(&p, &rsv_id) != RTS_GUARANTEED)
    exit_err("can't get proper scheduling guarantees!");

  rts_rsv_attach_thread(&p, gettid());

  /* params not needed anymore */
  rts_params_cleanup(&p);

  while (!computation_ended()) {
    int rmng_budget;

    compute();
    rts_rsv_get_remaining_budget(rsv_id, &rmng_budget);
    if (rmng_budget > 15000)
      compute_optional();

    wait_next_activation();
  }
  rts_rsv_destroy(rsv_id);

  exit(EXIT_SUCCESS);
}
```

Figure 2: Example C code using the library

requirements to actual parameters of the available scheduling algorithm(s) and issuing actual system calls.

The internal structure of the RTS Daemon takes advantage of a modular, plugin-based architecture. Each *scheduling plugin* needs to provide some of the functionality envisioned in the application-level API. Among these, the most remarkable functions are:

1) *Real-Time Scheduler Support*: Each module supports a precise real-time scheduling algorithm available in the underlying kernel, for example Fixed Priority, Sporadic Server, EDF, etc.
2) *Capability Matching*: Each module is able to check whether or not it can handle all of the parameters explicitly set in a reservation request by the application through the API;
3) *Admission Control*: Each module checks whether the scheduling guarantees requested by applications can be provided or not; this may be done by either simple or complex tests based on the precise knowledge of the underlying scheduling algorithm;
4) *Support for Adaptive Reservations*: Each module may support the requirements about adaptive reservations, and in such case it is responsible for properly managing overload conditions arising from independent reservations willing to get unfeasible desired budget figures;
5) *Spare Capacity Distribution*: Each module may embed a policy for the redistribution of available system capacity amongst active reservations, whenever the overall requested resources do not saturate the available resources.

The admission test process inside the daemon may query multiple plug-ins in some predetermined order, until it finds one that perfectly matches requirements of the application. Also, in case no plugin is able to admit the application request, we allow for the possibility that a plugin that may partially support the requested requirements provides a positive answer.

Therefore, we foresee three possible results as the outcome of a `rts_create_rsv()` call, to be communicated back to the application:

- `RTS_GUARANTEED`, if all of the application requirements can be satisfied;
- `RTS_NO_GUARANTEES`, if only some but not all of the requirements can be guaranteed, and the application allowed this possibility when setting up the parameters;
- `RTS_REJECTED`, if it is not possible to admit the application.

The intermediate return code is provided to give the application an opportunity to relax some of its requirements so that it can be accepted in the system as opposed to retrying continously.

C. Scheduling Plugin

In this section we describe the plugins that are among the most representative and important we are willing to support with the described architecture.

a) EDF Plugin: The EDF scheduling algorithm is well-known to be optimum for single-processor systems[32]. Therefore, given its availability at least in the partitioned flavor for recent Linux kernels [17, 18], we want to develop a plugin supporting this policy. In this case, the admission test may be as simple as utilization-based, or it may become more complex and based on demand-bound function techniques. The latter ones become particularly useful when specified deadlines are different from periods, because a utilization-based bound may be overly pessimistic in such cases.

b) Sporadic Server Plugin: The Sporadic Server algorithm can be used to enforce a CPU reservation in the Q over P form. However, the Sporadic Server implementation available for Linux [1] follows the POSIX specifications, and requires the server parameters to be mapped to the ones exposed by the kernel interface. In particular, the daemon needs to manage the priorities at which tasks are executed. As it is well-known, it is convenient to enforce a Rate Monotonic ordering of the tasks submitting reservation requests. Note that this may imply the need for changing the priorities of all of the running reservations, in order to host a new reservation request.

Concerning admission control, it is possible to use well-known analysis techniques for real-time FP systems, ranging from simple utilization bound, to the hyperbolic bound, to the more complex tests based on Response-Time Analysis.

One of the capabilities that is not supported by the POSIX standard is the enforcement of the hard limit on the granted reservation, due to the low priority. This information needs to be made available via capability querying. The plugin may for example assign a default low priority (e.g., the lowest one) as the low priority to be specified in the POSIX interface. Finally, as per the POSIX standard, Sporadic Server does not support attaching more than one Linux task to each reservation.

c) Fixed Priority Plugin: As a border-line real-time scheduler, we proposed an API that may be able to exploit an underlying POSIX Fixed priority real-time scheduler, which is the most common scheduling algorithm available on Linux and on other General-Purpose Operating Systems. This algorithm cannot handle of course temporal encapsulation nor admission control, so this information needs to be made available to applications through the capability querying interface, in case no other real-time schedulers are available on the system. Also, this is the typical case when the kernel may perform real-time scheduling of the reservations, but perhaps it cannot fulfill entirely the set of specified requirements, so the `RTS_NO_GUARANTEES` return code is used. One of the burden that the direct use of the POSIX API imposes when using FP scheduling, is the management of the priorities. For example, multimedia applications (e.g., sound daemons and CD recording software) use to launch themselves at a statically configured priority. However, it is well-known that the optimum priority assignment for periodic tasks is the Rate Monotonic [32] one. These applications may more effectively exploit FP scheduling by declaring their activation period through the `rts` API, and leave the `rts` daemon decide what is the best priority assignment considering the overall set of reservations active in the system.

V. Future Work and Conclusion

We have presented an architecture allowing applications to exploit real-time CPU scheduling capabilities available in the Linux kernel, both the ones of the mainline kernel and the ones that may be available applying a set of recently appeared well-engineered patches. The paper focused on the requirements and issues that drove the design of such an architecture, and on the abstraction level needed in an API offering such capabilities to applications.

We have to point out that the recent advances in the support for real-time tasks in Linux have not been paired with an increase in the usability of that support. For example, a `cgroup` based interface cannot easily be leveraged by application programmers. We feel that this is a major issue, and even knowing that no perfect interface exists, and that the journey we started is a long one, we hope this paper will constitute the basis of a discussion leading to a more usable (and used) real-time "experience" for both Linux programmers and users.

To reach our goal—designing a *practical* interface between application programmers and the real-time capabilities of Linux—we would like to begin interacting with the community and to gather feedback from the potential users of the architecture we are proposing. Both the requirements and the interface would benefit from being expanded and refined through the experience coming from application programmers and from the community in general. This constitutes the ultimate goal of the present paper: opening up a fruitful discussion that hopefully may refine the design while we proceed with the actual implementation of the proposed solution.

References

[1] D. Faggioli, A. Mancina, F. Checconi, and G. Lipari, "Design and implementation of a POSIX compliant sporadic server," in *Proceedings of the* 10^{th} *Real-Time Linux Workshop (RTLW)*, Mexico, October 2008.

[2] IEEE, *Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) Amendment: Additional Realtime Extensions.*, 2004.

[3] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, September 1990.

[4] G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo, *Soft Real-Time Systems Predictability vs. Efficiency*, ser. Series in Computer Science. Springer, 2005, no. 10.1007/0-387-28147-9-3.

[5] Ayers and B. V. Yodaiken, "Introducing real-time linux," *Linux J.*, p. 5, 1997.

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

[7] P. Gerum, *Xenomai - Implementing a RTOS emulation framework on GNU/Linux*, April 2004.

[8] K. Gopalan, "Real-time support in general purpose operating systems," 2001.

[9] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves for multimedia operating systems," Carnegie Mellon University, Pittsburg, Tech. Rep. CMU-CS-93-157, May 1993.

[10] M. B. Jones, "Cpu reservations and time constraints: Implementation experience on windows nt," in *In Proc. of the 3rd USENIX Windows NT Symposium*, 1999, pp. 93–102.

[11] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: A resource-centric approach to real-time and multimedia systems," in *In Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, 1998, pp. 150–164.

[12] K. Lakshmanan and R. Rajkumar, "Distributed resource kernels: Os support for end-to-end resource isolation," in *RTAS '08: Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 195–204.

[13] K. Kim, “Extended dsrt scheduling system,” Master’s thesis, Yonsei University, Korea, 1997.

[14] V. Vardhan, D. G. Sachs, W. Yuan, A. F. Harris, S. V. Adve, D. L. Jones, R. H. Kravets, and K. Nahrstedt, “Integrating finegrain application adaptation with global adaption for saving energy,” in *In Proceedings of the 2nd International Workshop on Powe-Aware Real-Time Computing (PARC)*, 2005.

[15] L. Abeni and G. Buttazzo, “Integrating multimedia applications in hard real-time systems,” in *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[16] 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.

[17] F. Checconi, T. Cucinotta, D. Faggioli, and G. Lipari, “Hierarchical multiprocessor CPU reservations for the linux kernel,” in *Proceedings of the* 5^{th} *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2009)*, Dublin, Ireland, June 2009.

[18] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino, “An edf scheduling class for the linux kernel,” in *Proceedings of the* 11^{th} *Real-Time Linux Workshop (RTLW 2009)*, Dresden, Germany, October 2009.

[19] J. H. Anderson, “Pfair scheduling: Beyond periodic task systems,” in *Proceedings of the* 7^{th} *International Workshop on Real-Time Computing Systems and Applications (RTCSA 2000)*, Cheju Island, South Korea, December 2000.

[20] R. Zhang, C. Lu, T. F. Abdelzaher, and J. A. Stankovic, “Controlware: A middleware architecture for feedback control of software performance,” in *Proc. of International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.

[21] M. García-Valls, A. Alonso, J. Ruiz, and A. M. Groba, “An architecture of a quality of service resource manager middleware for flexible embedded multimedia systems.” in *SEM*, ser. Lecture Notes in Computer Science, A. Coen-Porisini and A. van der Hoek, Eds., vol. 2596. Springer, 2002, pp. 36–55. [Online]. Available: http://dblp.uni-trier.de/db/conf/edo/sem2002.html#VallsARG02

[22] J. Blanquer, J. Bruno, E. Gabber, M. Mcshea, B. Ozden, A. Silberschatz, and A. Singh, “Resource management for qos in eclipse/bsd,” in *In Proceedings of the FreeBSD’99 Conference*, 1999.

[23] K. Gopalan and K.-D. Kang, “Coordinated allocation and scheduling of multiple resources in real-time operating systems,” in *Proc. of Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), Pisa, Italy*, June 2007.

[24] K. Nahrstedt, H.-h. Chu, and S. Narayan, “Qos-aware resource management for distributed multimedia applications,” *J. High Speed Netw.*, vol. 7, no. 3-4, pp. 229–257, 1998.

[25] D. C. Schmidt, D. L. Levine, and S. Mungee, “The design of the tao real-time object request broker,” *Computer Communications*, vol. 21, pp. 294–324, 1997.

[26] V. F. Wolfe, L. C. DiPippo, R. Ginis, M. Squadrito, S. Wohlever, I. Zykh, and R. Johnston, “Real-time corba,” in *IEEE Real Time Technology and Applications Symposium*. IEEE Computer Society, 1997, pp. 148–.

[27] Y. Krishnamurthy, V. Kachroo, D. A. Karr, C. Rodrigues, J. P. Loyall, R. E. Schantz, and D. C. Schmidt, “Integration of QoS-enabled distributed object computing middleware for developing next-generation distributed application,” in *LCTES/OM*, 2001, pp. 230–237.

[28] C. D. Gill, J. M. Gossett, D. Corman, J. P. Loyall, R. E. Schantz, M. Atighetchi, and D. C. Schmidt, “Integrated adaptive QoS management in middleware: A case study,” *Real-Time Systems*, vol. 29, no. 2-3, pp. 101–130, march 2005.

[29] N. Shankaran, X. D. Koutsoukos, D. C. Schmidt, Y. Xue, and C. Lu, “Hierarchical control of multiple resources in distributed real-time and embedded systems,” in *ECRTS’06: Proceedings of the 18th Euromicro Conference on Real-Time Systems*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 151–160.

[30] The FRESCOR Consortium, “Requirements Analysis,” FRESCOR EU project (FP6/2005/IST/5-034026), Deliverable D-RA2, Jan 2008.

[31] M. G. Harbour and M. T. de Esteban, *Framework for Real-time Embedded Systems based on COntRACTS – Deliverable: D-AC2v2 – Architecture and contract model for integrated resources II*, version 1.0 ed., Universidad de Cantabria, January 2008.

[32] C. L. Liu and J. Layland, “Scheduling alghorithms for multiprogramming in a hard real-time environment,” *Journal of the ACM*, vol. 20, no. 1, 1973.