Hierarchical Multiprocessor CPU Reservations for the Linux Kernel*

Fabio Checconi, Tommaso Cucinotta, Dario Faggioli, Giuseppe Lipari
Scuola Superiore S. Anna, Pisa, Italy

Abstract

This paper presents ongoing work in the development of a scheduling framework that will improve the service guarantees for soft real-time applications deployed on Linux. The scheduler has been designed around the current kernel infrastructure, trying to keep the changes minimal, and basing the scheduling policy on strong theoretical results. The main goal is to achieve hierarchical distribution of the available computing power on multiprocessor platforms, avoiding alterations to the existing user interfaces.

The proposed framework exploits the hierarchical arrangement of tasks within groups and subgroups that is already possible within the Linux kernel. However, it adds the capability for each group to be assigned a precise fraction of the computing power available on all the processors, using existing uni-processor resource reservation techniques. Tasks are scheduled globally within each single group, and the partitions assigned to each group need not to be static, but can be dynamically balanced. Furthermore, the proposed mechanism can be used to support a variety of possible partitioning schemes using processor affinities.

1 Introduction

Nowadays, the Linux Operating System is being enriched with more and more real-time capabilities. In the last few years, valuable efforts have been spent for decreasing the scheduling and interrupt latencies of the kernel, by embedding such features as full preemption, priority inheritance, reduced computation complexity of the scheduler, support for high-resolution timers. Also, the `linux-rt` branch adds such experimental features as running interrupt handlers in dedicated kernel threads rather than in interrupt context, so as to allow system designers to have an improved control over the interference of the peripheral drivers with respect to the running applications.

While such features make the Linux kernel a very appealing platform for multimedia applications, still the support for real-time scheduling is somewhat inappropriate for dealing with requirements posed by the challenging scenarios of the upcoming years, that demand for predictable scheduling mechanisms able to achieve a good degree of temporal isolation among complex concurrent software components, low response times and high interactivity. One such scenario is the one in which multiple virtual machines run within the same OS, hosting software components realizing professional services that need to run with predictable QoS levels and high interactivity requirements, possibly managed through a service-oriented approach, as discussed for example in [1].

The Linux kernel embodies the POSIX compliant priority-based real-time scheduling classes (`SCHED_FIFO` and `SCHED_RR`). These may be sufficient for dealing with embedded real-time applications, but they turn out to be inadequate for providing temporal isolation among complex software components such as the ones mentioned above. In fact, the implementation of such policies in Linux has been enriched by non-standard features such as support for hierarchies of tasks and *throttling*. However, lacking of a sound design in the domain of real-time scheduling, such capabilities struggle at constituting a solid base for providing an adequate real-time scheduling support.

This paper makes one step further in this direction, presenting a novel real-time scheduling strategy for the Linux kernel, that may be analyzed by means of hierarchical real-time schedulability analysis techniques. The proposed infrastructure has a good degree of flexibility, allowing for a variety of configurations between two traditionally antithetic settings: on one side, the perfect compatibility with the current POSIX compliant priority-

*The research leading to these results has been supported by the European Commission under grant agreement n.214777, in the context of the IRMOS Project. More information at: http://www.irmosproject.eu.

based semantics, and on the other side an improved usage of resources by means of a partitioned EDF.

1.1 Paper Contributions

This paper presents a hierarchical multiprocessor scheduling framework for the Linux kernel. The main advantages of the presented approach over prior works are:

- tight integration with the existing Linux code;
- no need for the introduction of new interfaces nor new scheduling classes;
- support for multiple configuration schemes, including fully partitioned approaches;
- strong theoretical background justifying the relevance of the approach, mainly inspired to [2], with the derivation of an appropriate admission test for the tasks to be scheduled;
- capability to handle accesses to shared resources.

1.2 Paper Outline

The rest of the paper is organized as follows. Section 2 reviews related work in the area, then Section 3 introduces considered system model and scheduling algorithm, summarizing its formal properties. Section 4 describes the implementation of the framework in the Linux kernel, and Section 5 reports experimental results that validate the approach. Finally, Section 8 contains a few concluding remarks.

2 Related Work

The growing interest in having more advanced real-time scheduling support within the Linux kernel has been witnessed in the last years by various research projects. The first approach that has been undertaken has been the addition of a hypervisor to the Linux kernel, so as to obtain a highly predictable hard real-time computing platform where real-time control tasks are scheduled very precisely, and the entire Linux OS is run in the background. Such an approach, adopted in the RTLinux [3] and RTAI [4] projects, however is not adequate for interactive nor multimedia applications, due to the high limitations it poses on the services available to real-time applications.

An alternative trend is constituted by the addition of a (soft) real-time scheduling policy directly within the Linux kernel, that allows for a more predictable execution of unmodified Linux applications. Projects that fall in this category comprise the following.

The Adaptive Quality of Service Architecture [5] (`AQuoSA`) for Linux provides hard CBS [6], an EDF based real-time policy, which has also been enhanced with the Bandwidth Inheritance protocol [7] for dealing with shared resources. However, having been developed in the context of the FRESCOR [1] European Project for embedded systems, `AQuoSA` suffers from the main limitation of not supporting SMP systems.

The $Litmus^{RT}$ project [8, 9] provides (among others) Pfair [10], a real-time scheduling strategy theoretically capable of saturating SMP systems with real-time tasks. However, it contains major changes of the Linux kernel internals, and it is currently more a testbed for experimenting with real-time scheduling within Linux, rather than something that aims at being integrated in the mainline kernel.

Recently, an implementation of the POSIX `SCHED_SPORADIC` [11] real-time policy for Linux has been proposed to the Linux kernel community [12]. This scheduler has been developed with the aim of being integrated into the mainstream kernel, by proposing a very limited set of modifications to the kernel scheduler, and exploiting existing user-space APIs such as the `cgroups`. The great advantage of such scheduling policy is the one of having been standardized by POSIX, however it suffers of the limitations typical of priority-based policies, such as the well-known utilization limit of 69% on uni-processor systems.

3 Scheduling Algorithm

The design of the scheduling algorithm started in a quite unusual way, analyzing the existing Linux scheduler, and trying to derive a formal model for the policy it is implementing for real-time scheduling, especially concerning the part of hierarchical scheduling. It turned out that the model in [2] is not far from matching the Linux implementation. The work we present in this paper aims to achieve a convergence between a hierarchical scheduling infrastructure that is minimally invasive as compared to the current Linux scheduler code base, and a theory of hierarchical real-time schedulers that is quite generic to be adapted to the Linux case.

We exploited the current user-space interface for the *throttling* mechanism, which offers to applications the possibility to assign a pair (Q_i, P_i) to the i-th group of tasks. However, these parameters are reinterpreted as the scheduling parameters (the budget and period, respectively) to be assigned to the group according to the well known resource reservation paradigm [6]: Q_i units of time are available to the group every period of length

[1] Framework for Real-Time embedded Systems based on Contracts (FRESCOR), European Project No. FP6/2005/IST/5-034026, more information at: http://www.frescor.org.

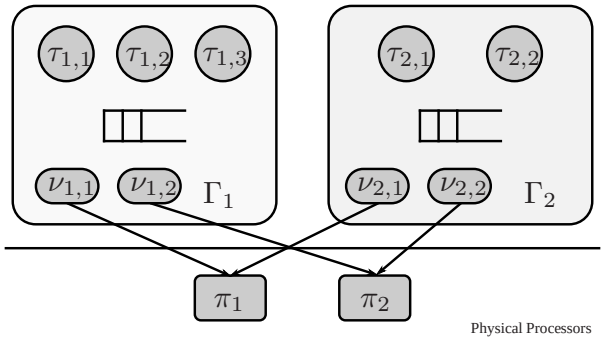

Figure 1: System Architecture.

P_i. The scheduling guarantee is given to each group as a whole, including all the tasks attached to the group itself and to all the nested subgroups. However, the framework allows each group and subgroup to posses its own set of scheduling parameters. On multiprocessor systems, the Q_i/P_i assignment is replicated on all the processors in the system, but the resulting schedulers on the various CPUs run independently from one another, minimizing synchronization overheads.

3.1 System Model and Terminology

Throughout the paper we stick to the Linux terminology as much as possible; when referring to entities that do not have a counterpart in the current Linux code yet, we derive our notation and terminology from [2].

In the model we consider, a task group Γ_i is composed by set of n_i sporadic tasks $\Gamma_i = \{\tau_{i,j}\}_{j=1,\ldots,n_i}$. Each task is described by its worst-case execution time $C_{i,j}$, its relative deadline $D_{i,j}$ and its minimum inter-arrival time $T_{i,j} : \tau_{i,j} = (C_{i,j}, D_{i,j}, T_{i,j})$. A task $\tau_{i,j}$ is a sequence of jobs $\tau^k_{i,j}$, each characterized by its own release time, computation time and deadline, denoted by $r^k_{i,j}, c^k_{i,j}$ and $d^k_{i,j}$, respectively.

Following [2], we call *virtual platform* $\mathcal{V}_i$ a set of m_i virtual processors $\mathcal{V}_i = \{\nu_{i,l}\}_{l=1,\ldots,m_i}$. Each virtual processor $\nu_{i,l}$ is characterized by a supply function $Z_{i,l}(t)$ representing the amount of service $\nu_{i,l}$ can provide in any time interval of duration t.

Tasks are grouped in task groups, organized in a hierarchical fashion. Each task group is assigned a virtual platform, one per physical processor $\pi_m \in \{\pi_m\}_{m=1,\ldots,M}$ in the system.

Fig. 1 depicts the global structure of our model, in the case of two physical processors, π_1 and π_2, and five tasks organised in two groups: $\tau_{1,1}, \tau_{1,2}, \tau_{1,3}$ inside Γ_1 and $\tau_{2,1}, \tau_{2,2}$ inside Γ_2; each task group is assigned two virtual processors, one for each physical processor.

3.2 Main Algorithm

The proposed algorithm can be described as a two-layer hierarchical scheduler, with the first layer scheduler selecting which task group to execute on each processor, and the second layer selecting which task to run within the selected task group.

Each task group Γ_i is assigned a set of virtual processors; these virtual processors are scheduled using partitioned resource reservation techniques. Each virtual processor is allocated a share of one of the physical processors in the system. The algorithm used to schedule virtual processors on physical processors is the Hard Constant Bandwidth Server (H-CBS) [6].

In other words, the first layer is composed by M independent partitioned H-CBS schedulers which manage all the virtual processors $\nu_{i,l}$ assigned to their respective physical processor. Looking again at Fig. 1, there are two H-CBS schedulers, one to schedule virtual processors running on π_1, and one for the ones running on π_2. The H-CBS on π_1 schedules the first virtual processors of the groups in the system ($\nu_{1,1}$ and $\nu_{2,1}$), while the H-CBS on π_2 schedules the second ones ($\nu_{1,2}$ and $\nu_{2,2}$).

Within each group tasks are kept in a global fixed-priority queue[2], and tasks belonging to the same task group are scheduled globally according to their priority. At every time instant, if a virtual platform $\mathcal{V}_i$ is in execution on m physical processors, then its m highest priority tasks are executing. Note that $m \leq M$ changes over time due to the asynchronous scheduling of virtual processors over the physical ones.

3.3 Formal Properties

The proposed scheduling strategy falls within the class of schedulers identified in the theoretical schedulability analysis framework presented in [2]. Therefore, for purposes related to schedulability analysis, the same system model and analysis techniques may be adopted, with an additional extension to support hierarchical scheduling.

When an arbitrary hierarchy is considered, the problem of scheduling an application Γ on a group with bandwidth allocated on multiple processors is reduced to the problem of scheduling Γ on the $\mathsf{M}\alpha\Delta$ abstraction corresponding to the service provided by the given group. Known techniques can be used to derive the parameters for the $\mathsf{M}\alpha\Delta$ abstraction representing the group.

In this section we present well-known results and adapt them to our framework; a schedulability test will be derived from Theorem 1 and Theorem 3 in [2].

[2] As Section 4 will explain, the global policy of the queue is implemented using per-processor queues.

3.3.1 The Supply Function

An abstraction to model the minimum CPU time provided in a given interval of time is the *supply function* [2, 13]. To introduce the supply function first we need the concept of *time partition*.

Definition 1 *A* time partition $\mathcal{P}$ *is a countable union of non-overlapping time intervals*

$$\mathcal{P} = \bigcup_{i \in \mathbb{N}} [a_i, b_i) \quad a_i < b_i < a_{i+1}. \tag{1}$$

Without loss of generality, we set the time when the first virtual processor starts in the system equal to 0.

Given a time partition $\mathcal{P}$, its supply function [2, 13] measures the minimum amount of CPU time provided by the partition in any time interval.

Definition 2 *Given a time partition* $\mathcal{P}$, *its* supply function $Z_{\mathcal{P}}(t)$ *is the minimum amount of CPU time provided by the partition in any time interval of length* $t \geq 0$, *i.e.,*

$$Z_{\mathcal{P}}(t) = \min_{t_0 \geq 0} \int_{\mathcal{P} \cap [t_0, t_0+t]} \mathrm{d}x. \tag{2}$$

Since, given a virtual processor ν, it is not possible to determine the time partition $\mathcal{P}$ it will provide, the above definition cannot be used in practice; the following two definitions generalize the considered time partition to all the possible partitions that can be generated by a virtual processor, and extend Def. 2 to be actually usable.

Definition 3 *Given a virtual processor* ν, $\mathsf{legal}(\nu)$ *is the set of time partitions* $\mathcal{P}$ *that can be allocated by* ν.

Definition 4 *Given a virtual processor* ν, *its* supply function $Z_\nu(t)$ *is the minimum amount of CPU time provided by the server* ν *in every time interval of length* $t \geq 0$,

$$Z_\nu(t) = \min_{\mathcal{P} \in \mathsf{legal}(\nu)} Z_{\mathcal{P}}(t). \tag{3}$$

A virtual processor ν implemented through a H-CBS with budget Q and period P, when active, conforms to the Explicit Deadline Periodic model [14] with deadline equal to the period. As a consequence, we can use the well-known supply function:

$$Z_\nu(t) = \max\{0, t - (k+2)(P-Q), kQ\}, \tag{4}$$

with $k = \left\lfloor \frac{t-P+Q}{P} \right\rfloor$.

3.3.2 The (α, Δ) Abstraction

A simpler abstraction, still able to model the CPU allocation[3] provided by a virtual processor, but using fewer parameters, and easier to derive is the "bounded delay partition," described by two parameters: a bandwidth α, and a delay Δ. The bandwidth α measures the rate at which an active virtual processor provides service, while the delay Δ represents the worst-case service delay.

The formal definitions of α and Δ, from [13], are given below.

Definition 5 *Given a virtual processor* ν *with supply function* $Z_\nu(t)$, *its* bandwidth α_ν *is defined as*

$$\alpha_\nu = \lim_{t \to \infty} \frac{Z_\nu(t)}{t}. \tag{5}$$

Definition 6 *Given a virtual processor* ν *with supply function* $Z_\nu(t)$ *and bandwidth* α_ν, *its* delay Δ_ν *is defined as*

$$\Delta_\nu = \sup_{t \geq 0} \left\{ t - \frac{Z_\nu(t)}{\alpha_\nu} \right\}. \tag{6}$$

Using the two definitions above, the supply function $Z_\nu(t)$ of a virtual processor ν can be lower bounded as follows:

$$Z_\nu(t) \leq \max\{0, \alpha_\nu(t - \Delta_\nu)\}, \tag{7}$$

which gives an intuitive definition of the (α, Δ) abstraction, as a way to extract a lower bound for the actual supply function of a virtual processor; α represents the share of the physical processor time assigned to the virtual processor, while Δ represents the responsiveness of the allocation. In the case of a H-CBS virtual processor of budget Q and period P, we have:

$$\alpha = \frac{Q}{P}, \quad \Delta = 2P - 2Q. \tag{8}$$

3.3.3 (α, Δ) Abstractions and Multiprocessors

In [2] an extension of the (α, Δ) abstraction for multiprocessors is given, along with the calculation of the (α, Δ) parameters for several algorithms described in literature.

Definition 7 *The Multi-*(α, Δ) *(*$\mathsf{M}\alpha\Delta$*) abstraction of a set* $\mathcal{V} = \{\nu_j\}_{j=1,\ldots,m}$ *of virtual processors, represented by the* m *pairs* $\{(\alpha_j, \Delta_j)\}_{j=1,\ldots,m}$ *is a multi-supply function defined by the set of supply functions* $\{Z_{\nu_j} : Z_{\nu_j}(t) = \max(0, \alpha_j(t - \Delta_j))\}_{j=1,\ldots,m}$.

[3] The same abstraction does not apply to CPU time only [15], but here we consider only CPU time.

3.3.4 Schedulability Analysis

We consider the schedulability of a single task group Γ (composed of n tasks) over a set $\mathcal{V} = \{\nu_j\}_{j=1,\ldots,m}$ of virtual processors, with supply functions $Z_j(t) = Z_{\nu_j}(t)$. First, assuming to know the time partition $\mathcal{P}_j$ provided by each ν_j, we define *the characteristic function* $S_j(t)$, defined as follows:

$$S_j(t) = \begin{cases} 1 & t \in \mathcal{P}_j \\ 0 & t \notin \mathcal{P}_j \end{cases} . \tag{9}$$

Without loss of generality, assume the tasks $\{\tau_k\}$ within Γ are ordered by non-increasing priority. Consider a single task $\tau_k \in \Gamma$. L_ℓ denotes the sum of the duration of all the time intervals over $[0, D_k)$ where ℓ virtual processors provide service in parallel:

$$\forall \ell : 0 \leq \ell \leq m, L_\ell = \left| \left\{ t \in [0, D_k) : \sum_{j=1}^{m} S_j(t) = \ell \right\} \right| . \tag{10}$$

With W_k we denote the workload of jobs with higher priority interfering with τ_k, and I_k denotes the total duration in $[0, D_k)$ in which τ_k is preempted by higher priority jobs. From [16] we know that, for a fixed priority scheduler, the workload $\overline{W}_k^{\mathsf{FP}}$ can be bounded using:

$$\overline{W}_k^{\mathsf{FP}} = \sum_{i=1}^{k-1} \overline{W}_{k,i}, \tag{11}$$

where

$$\overline{W}_{k,i} = N_{k,i} C_i + \min\{C_i, D_k + D_i - C_i - N_{k,i} T_i\}, \tag{12}$$

with $N_{k,i} = \left\lfloor \frac{D_k + D_i - C_i}{T_i} \right\rfloor$.

The following theorems, proved in [2], allow us to build a schedulability test.

Theorem 1 *Given a multi-supply function characterized by the lengths $\{L_\ell\}_{\ell=0,\ldots,m}$ over a window $[0, D_k)$, the interference I_k on τ_k produced by a set of higher priority jobs with total workload W_k cannot be larger than*

$$\overline{I}_k = L_0 + \sum_{\ell=1}^{m} \min \left(L_\ell, \frac{\max\left(0, W_k - \sum_{p=1}^{\ell-1} p L_p\right)}{\ell} \right) . \tag{13}$$

Now that we know how to calculate an upper bound to the interference $\overline{I}_k^{\mathsf{FP}}$, substituting $W_k = \overline{W}_k^{\mathsf{FP}}$ in the equation above, we can use the following theorem (again, from [2]) to derive a schedulability test.

Theorem 2 *A task set $\Gamma = \{\tau_i\}_{i=1,\ldots,n}$ is schedulable by a fixed priority algorithm on a set of virtual processors $\mathcal{V} = \{\nu_j\}_{j=1,\ldots,m}$ modeled by $\{Z_j\}_{j=1,\ldots,m}$, if*

$$\forall k \in \mathbb{N} : 1 \leq k \leq n \qquad C_k + \overline{I}_k^{\mathsf{FP}} \leq D_k, \tag{14}$$

using the following values for the lengths $\{L_\ell\}_{\ell=0,\ldots,m}$:

$$\begin{aligned} L_0 &= D_k - Z_1(D_k) \\ L_\ell &= Z_\ell(D_k) - Z_{\ell+1}(D_k) \\ L_m &= Z_m(D_k). \end{aligned} \tag{15}$$

The symmetry of our bandwidth distribution allows for a simplification in the above test. In fact we assign the same bandwidth and the same period to all the virtual processors corresponding to the same task group; thus the intermediate lengths L_ℓ are zero, and Eq. (13) can be simplified, resulting in the following equation for the interference $\overline{I}_k$:

$$\overline{I}_k = L_0 + \min \left(L_m, \frac{\max(0, W_k - m L_m)}{m} \right) . \tag{16}$$

As a final note, to multiplex different task groups on the same set of physical processors, the basic (necessary and sufficient) H-CBS admission test over the virtual processors $\{\nu_i\}_{i=1,\ldots,n}$ executing on the same physical processors must be verified:

$$\sum_{i=1}^{n} \frac{Q_i}{P_i} \leq 1. \tag{17}$$

3.4 Shared Resources

In order to support access to shared resources, the priority inheritance and boosting mechanisms, already present in the Linux kernel, may be exploited. Within the Linux kernel, the fact that internal mutexes do not adopt priority inheritance mechanisms limits the possibility of giving formal upper bounds to blocking times.

In our implementation, we are exploring the usage of non-preemptive critical sections, realized raising the priority of the task executing in critical section to the maximum one available in the system. We are trying to adapt the approaches and the analysis in [17] and [18] to our model; a formal treatment of the topic is left as a future work.

3.5 Policy and Mechanisms

The proposed scheduling framework can be used as the basis for implementing several different variations, using mechanisms already present in the kernel, or introducing small modifications.

As an example, consider a user willing to adopt a purely partitioned approach: said user needs only to use the `cpuset` mechanism to specify a CPU affinity for the task groups, and no changes are required to the scheduler itself. The partitioned queues make handling this case quite efficient, while the H-CBS scheduler takes care of partitioning the bandwidth among the task groups on the same physical processor, according to the specified timing constraints.

Another open issue is the optimal bandwidth assignment between virtual processors. We stick to the current Linux model of using the same assignment on each physical processor, both for its simplicity and for lack of an interface to express different assignments. Anyway the H-CBS scheduler would support asymmetric partitions too, and exploiting this capability would came at the cost of adding the user interface to set Q_i/P_i on a per-virtual processor basis, again, with no modification to the scheduler structure.

4 Implementation

We implemented our framework in the Linux kernel. We modified the existing real-time scheduling class, changing how task groups are selected for service.

The existing code represents groups of tasks using `struct task_group` objects; tasks can be grouped on the basis of their user id or on the basis of the cgroup they belong to. Each task group contains an array of per-processor runqueues and scheduling entities. Each runqueue contains the scheduling entities belonging to all its (active) child nodes in the hierarchy. Tasks are leaf nodes, represented only by their own scheduling entity. Each processor has its own runqueue, containing the scheduling entities belonging to tasks and groups from the highest level in the hierarchy; a task group has a different scheduling entity on each processor it can run on.

Fig. 3 shows the main differences introduced to the kernel data structures: the priority array in `struct rt_rq` has been substituted with a red-black tree, and a new field (`rt_deadline`) had to be added. The per-group high-resolution timer previously used for implementing the throttling limitation was replaced by a per-runqueue timer. The `rt_rq`s of a same task group are scheduled independently on the processors with H-CBS, thus the limitation periods are asynchronous among each other. If the high resolution tick is enabled on the system, the scheduler will use it to deliver accurate end-of-instance preemptions.

The two arrays added to `struct task_group` are used to store all the tasks for the given task group on each processor. The problem here is that tasks are not scheduled using H-CBS, and there is no easy way to mix their entities with the ones associated to intermediate nodes in the hierarchy. Our solution was to add a leaf runqueue to each intermediate runqueue, to store its tasks.

```
1  static inline int
2  rt_entity_before(struct sched_rt_entity *a,
3                   struct sched_rt_entity *b)
4  {
5          struct rt_rq *rqa = group_rt_rq(a), *rqb =
               group_rt_rq(b);
6
7          if ((!rqa && !rqb) || (rqa->rt_nr_boosted &&
               rqb->rt_nr_boosted))
8                  return rt_se_prio(a) < rt_se_prio(b);
9
10         if (rqa->rt_nr_boosted)
11                 return 1;
12
13         if (rqb->rt_nr_boosted)
14                 return 0;
15
16         return (s64)(rqa->rt_deadline -
               rqb->rt_deadline) < 0;
17 }
```

Figure 2: Entity Ordering.

```
1  struct rt_edf_tree {
2          struct rb_root rb_root;
3          struct rb_node rb_leftmost;
4  };
5
6  struct rt_rq {
7          struct rt_edf_tree active;
8          u64 rt_deadline;
9          struct hrtimer rt_period_timer;
10         /* ... */
11 };
12
13 struct task_group {
14         struct sched_rt_entity **rt_task_se;
15         struct rt_rq **rt_task_rq;
16         /* ... */
17 };
```

Figure 3: Data Structures.

Just to give a rough sketch of how the active tree is handled, Fig. 2 shows the function used to order entities. When inserting into a leaf runqueue both entities are tasks, so their priorities are compared. When both entities are runqueues they are ordered by priority if both of them are boosted (i.e., executing inside a critical section), otherwise boosted runqueues are favored over non-boosted ones. If none of them is boosted, they are ordered by deadline.

The cgroup interface exported by the scheduler has been extended, in order to allow the definition of the CPU reservation for the tasks belonging to a group. As we previously said, all the tasks in a group are scheduled using a “ghost” runqueue, which gets its own CPU share; the only change we made to the current cgroup user interface was adding the filesystem parameters to specify the bandwidth allocated to this ghost queue, i.e., the bandwidth allocated to the tasks in each given group.

5 Experiments

This section presents some preliminary results obtained with our implementation of the scheduler described so far. Our primary focus is evaluating the overhead introduced by the mechanism, thus we compare it to the current throttling implementation.

We measured the time spent by the scheduler inside each of the class-specific hooks, filtering out the callbacks registered by the other scheduling classes. For our measurements we instrumented the scheduler code and then we used an ad-hoc minimal tracer[4], that measured the time spent in the main scheduling functions using the timestamp counter present in all the modern x86 CPUs. The values acquired using the TSC were stored in a per-processor ring buffer and copied to userspace using a daemon reading from a character device; the fact that all the functions we profiled are called under the runqueue locks assured that measurement errors due to interrupts or preemptions were avoided.

The functions we measured are:

- `check_preempt_curr_rt()`, which, given the current task and a newly woken one, checks if the latter is entitled to preempt the former;
- `task_tick_rt()`, which handles the system tick for RT tasks (mainly it checks for timeslice expiration of round-robin tasks);
- `enqueue_task_rt()`, which adds a task to the RT runqueues. In our implementation this function is responsible of updating the deadline according to the H-CBS rules, if necessary;
- `dequeue_task_rt()`, which removes a task from the RT runqueues;
- `put_prev_task_rt()`, which moves the running task back to the ready (but not running) state;
- `pick_next_task_rt()`, which selects the next task to run (if any).

The system used was a quad-core Intel Q6600, clocked at 2.40GHz, equipped with 2GB of RAM. The synthetic load we chose was the Fixed Time Quanta [19] benchmark, executed at RT priority.

Fig. 4 shows the execution times for the RT scheduling-related functions mentioned above, in the case there are four (one per core) application threads running. With our approach, the enqueue and dequeue paths are slower, as one would expect with the substitution of the previous $O(1)$ priority array implementation.

[4]We didn't use the `ftrace` infrastructure because on our configuration it introduced non-negligible overheads.

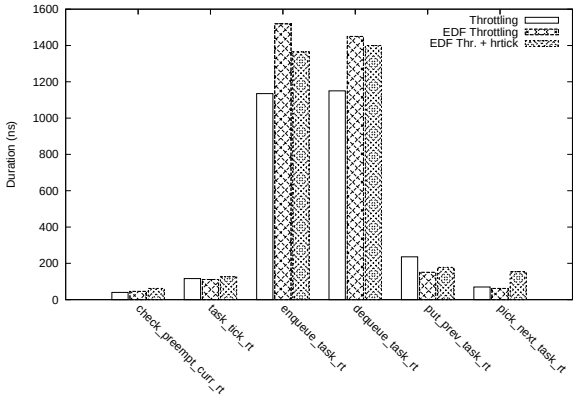

Figure 4: Scheduling Overhead—Flat Hierarchy.

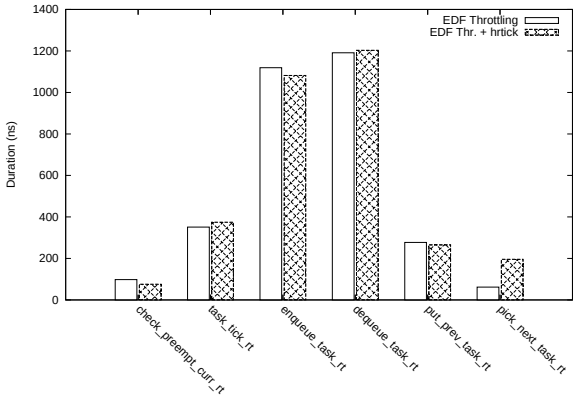

Figure 5: Scheduling Overhead—Full Hierarchy.

It is also worth noting that using the high resolution tick does not seem to affect the performance of the scheduling functions, except for `pick_next_task_rt()`, which is the function that programs the timer.

Fig. 5 shows the execution times of the RT scheduling class methods when there are more than a single root group, and with groups using different periods. We could not show the current scheduler behavior, as it does not support non-uniform periods. Performance is not too far from what shown in Fig 4 for native Linux, while in this case the overhead for posting the high resolution tick is more evident.

6 Availability

The implementation of the scheduler described in this paper is available as a patch to the Linux kernel, version 2.6.30-rc8, the latest available at the time of writing. It can be downloaded from

`http://feanor.sssup.it/~fabio/linux/edf-throttling/`

7 Future Work

The scheduler presented in this paper is still a work in progress. Our final objective is obtaining an implementation that can be considered for merging by the Linux community, yet based on sound theoretical principles.

About the implementation, we need a detailed study of the introduced overheads, along with the analysis of the computational cost given by keeping partitioned queues to implement a global scheduling strategy. From the theoretical standpoint, the biggest hole that needs to be filled in our opinion is the analysis of shared resources access.

8 Conclusion

In this paper we introduced a scheduling framework extending the Linux scheduler in order to improve its support for real-time workloads on multiprocessor systems. The main contribution of the paper is the synthesis between known theoretical results and the simplicity of the scheduler implementation, along with the specification of a complete strategy to solve the different issues that must be considered when designing a CPU scheduler (i.e., it deals with shared resources, CPU affinities, partitioned data structures and so on).

References

[1] T. Cucinotta, G. Anastasi, and L. Abeni, "Respecting temporal constraints in virtualised services," in *To appear in Proceedings of the* 2^{nd} *IEEE International Workshop on Real-Time Service-Oriented Architecture and Applications (RTSOAA 2009)*, Seattle, Washington, July 2009.

[2] E. Bini, G. Buttazzo, and M. Bertogna, "The multi supply function abstraction for multiprocessors," *to appear,available online at http://feanor.sssup.it/∼marko/RTCSA09.pdf*, 2009.

[3] RTLinux homepage, http://www.rtlinux.org.

[4] RTAI homepage, http://www.rtai.org.

[5] 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.

[6] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proc. IEEE Real-Time Systems Symposium*, Madrid, Spain, 1998.

[7] D. Faggioli, G. Lipari, and T. Cucinotta, "An efficient implementation of the bandwidth inheritance protocol for handling hard and soft real-time applications in the Linux kernel," in *Proceedings of the* 4^{th} *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2008)*, Prague, Czech Republic, July 2008.

[8] "Linux Testbed for Multiprocessor Scheduling in Real-Time Systems ($LITMUS^{RT}$)," http://www.cs.unc.edu/ anderson/litmus-rt/.

[9] B. Brandenburg, J. M. Calandrino, and J. H. Anderson, "On the scalability of real-time scheduling algorithms on multicore platforms: A case study," in *Proceedings of the Real-Time Systems Symposium*, Barcelona, 2008.

[10] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 6, 1996.

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

[12] 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.

[13] A. K. Mok, X. A. Feng, and D. Chen, "Resource partition for real-time systems," *Real-Time and Embedded Technology and Applications Symposium, IEEE*, vol. 0, p. 0075, 2001.

[14] A. Easwaran, M. Anand, and I. Lee, "Compositional analysis framework using edp resource models," *Real-Time Systems Symposium, IEEE International*, vol. 0, pp. 129–138, 2007.

[15] D. Stiliadis and A. Varma, "Latency-rate servers: A general model for analysis of traffic scheduling algorithms," in *IEEE/ACM Transactions on Networking*, 1996, pp. 111–119.

[16] M. Bertogna, M. Cirinei, and G. Lipari, "Schedulability analysis of global scheduling algorithms on multiprocessor platforms," *IEEE Transactions on Parallel and Distributed Systems*, 2008.

[17] M. Bertogna, F. Checconi, and D. Faggioli, "Nonpreemptive access to shared resources in hierarchical real-time systems," in *1st Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, Barcelona, Spain, December 2008.

[18] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson, "A flexible real-time locking protocol for multiprocessors," in *RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 47–56.

[19] FTQ http://rt.wiki.kernel.org/index.php/FTQ.