An implementation of a multiprocessor bandwidth reservation mechanism for groups of tasks*

Andrea Parri

Scuola Superiore Sant'Anna
Via Moruzzi 1, Pisa
andrea.parri@sssup.it

Mauro Marinoni
Scuola Superiore Sant'Anna
Via Moruzzi 1, Pisa
mauro.marinoni@sssup.it

Juri Lelli
Scuola Superiore Sant'Anna
Via Moruzzi 1, Pisa
juri.lelli@sssup.it

Giuseppe Lipari
Scuola Superiore Sant'Anna
Via Moruzzi 1, Pisa
giuseppe.lipari@sssup.it

Abstract

Hierarchical scheduling is a promising methodology for designing and deploying real-time applications, since it enables component-based design and analysis. Such techniques are also helpful for providing temporal isolation and timing guarantees in open systems, and for enabling application-specific schedulers. The Bounded-Delay Multipartition (BDM) interface was proposed by Lipari and Bini in "A framework for hierarchical scheduling on multiprocessors: from application requirements to run-time allocation" (2010) to allow the designer to balance between flexibility in resource allocation and the cost of resource over-provisiong necessary for meeting the timing constraints.

In this paper, we present an implementation within the Linux kernel of a multiprocessor bandwidth reservation mechanism for control groups based on the BDM interface, and we report on a first experimental evaluation. Our work is based on `SCHED_DEADLINE`, a scheduling class in the Linux kernel that provides task-level resource reservation using the Constant Bandwidth Server algorithm, and it extends Linux's current structures and interface by replacing the control groups throttling mechanism with an EDF-based reservation algorithm. Results show agreement with theoretical analysis, and overheads comparable with the current implementation of cgroups throttling in Linux.

*This work has been partially supported by the 7th Framework Programme JUNIPER (FP7-ICT-2011.4.4) project, founded by the European Community under grant agreement n. 318763.

1 Introduction

Thanks to the recent advances in the field of computer architectures, it is now common practise to concurrently execute different real-time applications in the same system. The motivations are costs reduction and reuse of legacy applications on new and faster multicore platforms.

When executing many real-time applications in the same system, a problem to be solved is how to schedule them efficiently while guaranteeing that their timing requirements are not violated. A possible solution is the use of an unique scheduling paradigm for the whole system and the design of all applications accordingly to it. However, such an approach increases the complexity of the schedulability analysis and it is also unable to isolate an application from the misbehaviour of the others.

A wiser and more robust way of composing applications with specific timing constraints is to use a two-level scheduling paradigm. At the *root level*, a scheduler selects the application that will be executed and its assigned processor time. Each application uses a *local scheduler* that selects which task of the application will be scheduled next. The local scheduler has visibility of the corresponding application's tasks only, and it is invoked when the root-level scheduler allocates the resource to the application.

The computational requirements of a real-time application are abstracted by means of a *temporal interface*. At design time, the application designer must characterize the temporal requirements of the application, and derive the appropriate parameters values that summarizes these requirements.

At root level a feasibility analysis to check if the application can be safely admitted without compromising the guarantees of the existing applications is performed. The root-level scheduler "protects" each application from all others, by ensuring that no application can execute more than declared in the interface. As a consequence, the feasibility of each application can be analysed independently.

Some authors have addressed the problem of how to specify the temporal interface for an application to be executed on multiprocessor systems. Leontyev and Anderson ([7]) proposed to consider the application overall bandwidth requirement as the interface for soft real-time applications, providing only an upper bound of the tardiness of tasks scheduled on such interface. Shin et al. ([8]) proposed the multiprocessor periodic resource model (MPR), consisting in a set of periodic reservations, all with the same period. Even though this interface model is rather intuitive, it requires a complex synchronization between reservations running on different processors. Chang et al. ([9]) proposed to partition the resource available from a multiprocessor by a static periodic scheme. The amount of resource is then provided to the application through a contract specification. Lipari and Bini ([3]) proposed an interface model, called bounded-delay multipartition (BDM) interface, that allows to balance the consumed bandwidth vs. the flexibility of the interface. Burmyakov et al. ([10]) proposed the generalized multiprocessor periodic resource model (GMPR) which extends the MPR model by specifying the minimal budgets for each level of parallelism.

The current Linux kernel supports hierarchical scheduling of tasks through the control groups *throttling* mechanism; however, this mechanism does not provide isolation among different task groups. Recently the new `SCHED_DEADLINE` scheduling class, providing temporal isolation among tasks, has been included in Linux; however, it only supports reservation for individual tasks.

This paper presents an implementation of the bounded-delay multipartition model ([7]) within the Linux kernel. Our implementation enables resource reservation for groups of tasks.

The rest of the paper is organized as follows. In Section 2, we describe the system model and recall some known results from the theory of hierarchical real-time scheduling. In Section 3, we describe the details of our implementation of a virtual platform model in the Linux kernel, including basic data structures and the user interface. In Section 4, we describe some experimental results aimed at validating the proposed implementation and at evaluating its ovehead. Finally, in Section 5 we state the concluding remarks and we overview future works.

2 Foundations

This section introduces the terminology used throughout the paper, and recalls some known results from the theory of real-time scheduling.

2.1 Virtual Platforms

The overall system is composed by a set of (real-time) applications that run concurrently onto a multiprocessor machine M with m identical processors.

Definition 1. An *application* A is a set of n independent sporadic tasks $\{\tau_1, \dots, \tau_n\}$, $\tau_i := (C_i, D_i, T_i)$ $(i = 1, \dots, n)$, with constrained deadline.

Every time a task is activated, a job must be executed. The *minimum inter-arrival time* T_i is the minimum separation between two consecutive jobs of τ_i; each job of τ_i has a *computation time* C_i and must be completed within a *(relative) deadline* $D_i \leq T_i$ from its activation.

To improve composability and isolation, each application is executed onto a dedicated *virtual platform.*

Definition 2 ([1])**.** A *virtual platform* Y on the multiprocessors M is modeled by a sequence of m functions $(Y_k)_{k=1}^m$, $Y_k\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ $(k = 1, \ldots, m)$. For each $t \in \mathbb{R}_{\geq 0}$, $Y_k(t)$ represents the "minimum amount of CPU time with parallelism at most k" provided to the application by the virtual platform Y in any (time) interval of lenght t.

The form of the functions Y_k $(k = 1, \ldots, m)$ depends on the particular algorithm that the operating system or the reservation manager is adopting to implement the virtual platform Y. We call this algorithm the global or the *root scheduling algorithm*, in order to distinguish it from the *local scheduling algorithm* used within an application to decide which of its jobs (among those eligible for execution) are to be executed at each time-instant.

2.2 Schedulability Test

The notion of virtual platform enables an approach to the schedulability analysis at the "application level". We focus on the case of a (local) *global fixed-priority* (G-FP) scheduling algorithm but the analysis can be extended to other policies. Bini et. al. demonstrated the following theorem on the schedulability of such systems:

Theorem 1. *Consider an application A and a dedicated virtual platform Y as in Definition 1 and in Definition 2, respectively. Assume G-FP as the local scheduling policy and define the* interfering workload *on task τ_i by:*

$$W_i = \sum_{j \in \mathrm{hp}(i)} W_{ji}, \tag{1}$$

where hp(i) *denotes the set of the indices of tasks with higher priority than τ_i and*

$$W_{ji} = N_{ji}C_j + \min\{C_j, D_i + D_j - C_j - N_{ji}T_j\},$$

$$N_{ji} = \left\lfloor \frac{D_i + D_j - C_j}{T_j} \right\rfloor.$$

Then A is (G-FP) schedulable (i.e., each job of A meets its timing constraints) if the following is true:

$$\bigwedge_{i=1}^{n} \bigvee_{k=1}^{m} kC_i + W_i \leq Y_k(D_i). \tag{2}$$

Notice that Equation 2 (and Theorem 1) says nothing on the actual implementation of the virtual platform for a given application or about the schedulability at the "root level", especially if multiple applications are present in the system. In the next section, we will expand on this issue by describing our implementation of the a virtual platform model within the Linux kernel.

3 Implementation

Our implementation is built upon Linux 3.14, patched with RT-Preempt 3.14.0-rt1. We assume `CONFIG_RT_GROUP_SCHED=y` throughout the rest of the paper. The source code of our implementation is available as a patch at `retis.sssup.it/juniper-project/BDM/`.

3.1 General Approach

In our implementation we consider virtual platforms which are "consistent with a specific *interface*" ([1, 2, 3]). Specifically, given arbitrary $\alpha \in \mathbb{Q} \cap [0, 1)$ and $\Delta \in \mathbb{Q}_{>0}$, our implementation provides the capability to create a virtual platform $Y := (Y_k)_{k=1}^m$ such that

$$Y_k(t) \geq k\,\alpha \cdot \max\{0, t - \Delta\}, \tag{3}$$

for all $k = 1, \ldots, m$ and $t \geq 0$. Informally, we say in this case that Y "dominates" the *bounded-delay multipartition* (BDM) defined in [3] as:

$$\left(Y_{k\,\alpha}^{\Delta}\right)_{k=1}^{m}. \tag{4}$$

In order to achieve this result, m new scheduling entities $\pi_1, \ldots, \pi_m$ (one for each "physical" processor), named "virtual processors", are associated with each virtual platform. A *virtual processor* represents a Hard Constant Bandwidth Server (H-CBS) (e.g., see [4]) which is *statically* allocated to a processor where this is scheduled in Earliest Deadline First (EDF) order. The (Q, P)-parameters of the virtual processors associated with the platform Y in Equation 3 are all equal to each other and can be computed according to the transformation:

$$\begin{aligned} Q &= \frac{\Delta}{2\,(1-\alpha)} \cdot \alpha, \\ P &= \frac{\Delta}{2\,(1-\alpha)}. \end{aligned} \tag{5}$$

It is known from the analysis of the H-CBS algorithm proposed by Abeni and Buttazzo [5], that these servers are schedulable iff

$$\sum_{a=1}^{N_A} \frac{Q_a}{P_a} = \sum_{a=1}^{N_A} \alpha_a \leq 1, \qquad (6)$$

N_A being the number of applications in the system.

We stress that, while virtual processors are statically partitioned upon the physical ones, the jobs of the application executing within those virtual processors *can* be "migrated" to different processors in conformity with the local scheduling algorithm. Our implementation considers the case of a *local FP scheduling* algorithm at the application level, which is not necessarily "global" (see Section 3.3).

3.2 Root Scheduler

It is now described the implementation of the *root scheduling algorithm* and its main data structures are displayed in Listing 1.

A virtual platform is represented as a `task_group` object; this includes an array of pointers to virtual processors entities (`sched_dl_entity`) and an array of pointers to "real-time" run-queues (`rt_rq`): as already described, there is one virtual processor entity for each physical processor/CPU; moreover, conforming to Linux's current implementation of the FP scheduling policy (`rt_sched_class`), each platform mantains a per-CPU (local) run-queue used to implement a "distributed" global scheduling algorithm that will be described in in Section 3.3. The "reservation parameters" of a virtual platform are encoded in a `dl_bandwidth` object (and "cached" in the corresponding `sched_dl_entity`'s): $Q =$ `dl_runtime` (ns) and $P =$ `dl_period` (ns), using the notation presented in Section 3.1.

We remark that the structure `sched_dl_entity` is already included in mainline Linux to store scheduling entities of `SCHED_DEADLINE` jobs (i.e., H-CBSs): our implementation preserves the semantics of its members and augments them with a pointer of type `dl_rq` (the run-queue on which the virtual processor/`SCHED_DEADLINE` job is to be queued) and with a pointer of type `rt_rq` (the local run-queue "owned" by this virtual processor; `NULL` for a `SCHED_DEADLINE` job). In particular, the members `runtime` and `deadline` represents the "current budget" and the "absolute deadline" of the H-CBS server, respectively; also, a timer (`dl_timer`) is "started" when the server "exhausts its budget" (we say that the server is being *throttled*) and set "to fire" at the next "replenishment instant" of the server. The adoption of the same C structure (`sched_dl_entity`) to represent both virtual processors and `SCHED_DEADLINE` jobs allowed us to reuse code already available in Linux's current implementation of the `SCHED_DEADLINE` scheduling policy (`dl_sched_class`); for example, the functions `dl_runtime_exceeded`, `start_dl_timer`, `dl_timer`, `enqueue_dl_entity`, `dequeue_dl_entity` apply to virtual processor entities with minor modifications.

```
1  struct dl_bandwidth {
2          raw_spin_lock_t dl_runtime_lock;
3          u64 dl_runtime;
4          u64 dl_period;
5  };
6  
7  /* struct for virtual platforms */
8  struct task_group {
9        struct sched_dl_entity **dl_se;
10       struct rt_rq **rt_rq;
11 
12       struct dl_bandwidth dl_bandwidth;
13 
14       ...
15 };
16 
17 /* struct for virtual processors */
18 struct sched_dl_entity {
19       struct rb_node rb_node;
20 
21       u64 dl_runtime;
22       u64 dl_period;
23 
24       s64 runtime;
25       u64 deadline;
26 
27       int dl_throttled, dl_new;
28       struct hrtimer dl_timer;
29 
30       struct dl_rq *dl_rq;
31       struct rt_rq *my_q;
32 
33       ...
34 };
35 
36 #define dl_entity_is_task(dl_se) \
37       (!(dl_se)->my_q)
```

Listing 1: Main data structures.

The `sched_dl_entity`'s of both virtual processors and `SCHED_DEADLINE` jobs that are *Active* (i.e., non-throttled) are enqueued in the same per-CPU red-black trees (from which the name `rb_node`) in order of non-decreasing absolute deadline; the macro `dl_entity_is_task` (line 39) has been introduced to distinguish entities representing `SCHED_DEADLINE` jobs from entities representing virtual processors. The function `pick_next_task_dl` of the class `dl_sched_class` has been modified as displayed in Listing 2: given a (per-CPU) run-queue `rq`, we first identify the corresponding red-black tree (line 7); if there is no `SCHED_DEADLINE` or virtual processor entity in the tree, we return `NULL` (lines 14-15); if the tree is not empty, we select the leftmost entry of type `sched_dl_entity`, `dl_se`, in this tree (line 17); if this entry represents a virtual processors, we return the highest priority real-time job in the corresponding local run-queue (lines 18-27); otherwise, the entry must represent a `SCHED_DEADLINE` job and we return this job (lines 29-31). Notice that the function `pick_next_task_dl` can now (misleadingly) return a job with `SCHED_FIFO` or `SCHED_RR` policy. Moreover, as a direct consequence of the implementation of this function, Equation 6 needs to be modified to account for the "total bandwidth" allocated to `SCHED_DEADLINE` jobs, as will be detailed in Section 3.4.

```
1  /*
2   * From:
3   *
4   *   struct task_struct *
5   *   pick_next_task_dl(struct rq *rq);
6   */
7  struct dl_rq *dl_rq = &rq->dl;
8  struct sched_dl_entity *dl_se;
9  struct task_struct *p;
10
11 /*
12  * dl_nr_total = # of SCHED_DEADLINE jobs
13  *             + # of virtual processors
14  */
15 if (unlikely(!dl_rq->dl_nr_total))
16       return NULL;
17
18 dl_se = pick_next_dl_entity(rq, dl_rq);
19 if (!dl_entity_is_task(dl_se)) {
20       struct rt_rq *rt_rq = dl_se->my_q;
21       struct sched_rt_entity *rt_se;
22
23       rt_se = pick_next_rt_entity(rq, rt_rq);
24       /* rt_se != NULL */
25       p = rt_task_of(rt_se);
26       ...
27       return p;
28 }
29
30 p = dl_task_of(dl_se);
31 ...
32 return p;
```

Listing 2: "Selecting" a virtual processor.

Finally, since virtual processors do not migrate between different processors as said in Section 3.1, the Linux's pull/push functions, as presented in [6] and available in `dl_sched_class` for `SCHED_DEADLINE` jobs, do not apply to `sched_dl_entity`'s representing virtual processors.

3.3 Local Scheduler

Our implementation of the local FP scheduling algorithm is based on Linux's `rt_sched_class`: the basic C structures, `sched_rt_entity` and `rt_rq`, are mantained to implement a `SCHED_FIFO` or `SCHED_RR` scheduling policy. A major effort consisted in the modification of the Linux's pull/push mechanism; we remark that this mechanism is used to implement a global or, more generally, an arbitrary processor affinity (APA) scheduling algorithm. We limit the following discussion to the case of global scheduling, but similar considerations hold for APA scheduling ([6]).

For global scheduling, the main invariant is given by the following definition:

Definition 3 (G-FP Invariant)**.** For a virtual platform Y, let $S_Y(t)$ be the set of real-time jobs of Y which are executing on any of the m CPUs at time t, and let $m_Y(t)$ be the set of virtual processors of Y which have been selected by the root scheduler on any of the m CPUs at time t. Let $p(j)$ denote the priority of the job j. If j_r is a runnable job of Y at time t and if $j_r \notin S_Y(t)$, then

$$|S_Y(t)| = |m_Y(t)| \text{ and } \forall j \in S_Y(t)\ \ p(j) \geq p(j_r).$$

Notice that the G-FP Invariant (GFPI) property does not specify *how* the root scheduler select the virtual processors (compare with [6], where $m_Y(t)$ is constant and equals the number of CPUs, m).

In order to preserve this invariant, our implementation introduces the functions `group_pull_rt_task` and `group_push_rt_tasks`. The first is called in `pick_next_dl_entity` (line 18 in Listing 2), *after* a virtual processor entity has been selected by the root scheduler; this function tries to pull a job on the corresponding local run-queue by scanning all the run-queues in its platform. The second is called on each CPU when a scheduling decision is *completed* (see `post_schedule`); if the "previous" or the "current" job is a `SCHED_FIFO`/`SCHED_RR` job, this functions tries to push jobs from the corresponding run-queue by searching for a "better" run-queue in the platform. As in mainline Linux, a successful push triggers a *rescheduling* on the "remote" CPU.

In order to preserve the GFPI property, all the events which could lead to its violation must be considered:

- *A new platform is created, destroyed or its reservation parameters are modified*: when these events occur, the platform can not have any assigned real-time jobs;
- *A job τ is assigned/removed to/from a platform Y*: our solution calls `resched_task` in order to trigger a rescheduling on the local CPU, which, in turn, will trigger the pull/pull mechanism described above;
- *The scheduling class or the priority of a job assigned to a platform is modified*: our solution

calls check_preempt_curr that tests if a rescheduling is required;

- *A job woken up or migrated within a platform*: as in the previous case, when these events occur our solution calls check_preempt_curr;
- *A virtual processor is "preempted" or it exhausts its budget*: when these events occur our solution calls resched_task.

3.4 User Interface

Similarly to Linux's current real-time throttling infrastructure, our implementation of the virtual platform model provides an interface based on the cgroup virtual file system.[1]

A virtual platform can be created by making a sub-directory under the "cpu sub-system" directory in this file system; within each such directory, the files cpu.rt_runtime_us and cpu.rt_period_us can be used to read/write the values (in μs) of the reservation parameters Q and P, respectively, for the corresponding virtual platform (see equations 3, 5). Reservation parameters are also available for the cpu sub-system directory: if we let (Q, P) denotes the parameters corresponding to this (system) reservation and if we let (Q_a, P_a), $1 \leq a \leq N_A$, denote the reservation parameters corresponding to the virtual platforms in the system, then our implementation checks that

$$\sum_{a=1}^{N_A} \frac{Q_a}{P_a} \leq \frac{Q}{P}, \qquad (7)$$

whenever a virtual platform is created or modified. We remark that the virtual platform model considers a hierarchical scheduling framework with *two levels* (the so called "root level" and the "application level"): for this reason, our implementation prevents users from making sub-directories with depth greater than one (we consider a depth of zero for the cpu sub-system directory).

If "real-time bandwidth control" is enabled (/proc/sys/kernel/sched_rt_runtime_us >= 0), our application checks that

$$m \cdot \sum_{a=1}^{N_A} \frac{Q_a}{P_a} + \mathrm{DL} \leq m \cdot \frac{Q}{P}, \qquad (8)$$

whenever an instance of the structure sched_dl_entity (a virtual processor or a SCHED_DEADLINE job) is created or modified, where we denoted by DL the total bandwidth allocated to SCHED_DEADLINE jobs. Notice that Equation 8 expresses a necessary but, in general, *not* sufficient condition for schedulability.

[1]For more information on Linux's cgroup, see the relative documentation in the kernel source tree.

4 Evaluation

In this section, we describe some experiments aiming at validating the proposed solution. First, a runtime test is used to check the correct behaviour of our approach with respect to the mainline Linux kernel. Then, the overhead of scheduling functions is measured to confirm the real applicability of our solution.

We executed the experiments on an Intel®Core2™Q6600 quad-core machine with 4GB of RAM, running at 2.4GHz.

4.1 Runtime Validation

We considered a system composed by the virtual platforms Y_1 and Y_2 defined in Table 1, and by the real-time applications defined in Table 2. Finally, we considered the (disturbing) background workload defined in Table 3.

Virtual platform	# of virt. processors	α	Δ (ms)
Y_1	2	0.72	20
Y_2	2	0.22	20

TABLE 1: *Platforms for validation.*

Virtual platform	i	p_i	C_i (ms)	D_i (ms)	T_i (ms)
Y_1	1	13	10	60	60
	2	12	140	270	270
	3	11	90	520	520
Y_2	4	15	40	270	270
	5	14	40	520	520

TABLE 2: *Applications for validation.*

i	p_i	C_i (ms)	D_i (ms)	T_i (ms)
6	18	25	100	100
7	17	50	200	200
8	16	100	400	400

TABLE 3: *Background workload.*

We used rt-app[2] to generate this workload and to count the number of deadline misses for the corresponding jobs over a time-window of 120 seconds. We ran this experiment 20 times against both our implementation and mainline Linux. In agreement with the theoretical results from Section 2.2, no misses is detected when using virtual platforms (the applications are both schedulable, as can be verified by applying Theorem 1). Table 4 reports the results when using Linux's throttling mechanism.

i	Throttling (no background)	Throttling (with background)
1	6 ± 1	210 ± 9
2	0 ± 0	222 ± 6
3	0 ± 0	5 ± 1
4	0 ± 0	0 ± 0
5	0 ± 0	0 ± 0

TABLE 4: *Average number of deadline misses for the applications of Table 2 over 20 runs, when using Linux's throttling.*

As it emerges from Table 4, the Linux's throttling mechanism is not able to guarantee the real-time constraints of the applications. Notice that this is true even when no background workload is present.

4.2 Overhead Measurement

Turing the experiment described in Section 4.1 we collected the overhead measurements of Linux's scheduling functions, obtained using ftrace[3] Table 5 and Table 6 show a report of the measurements when using virtual platforms and throttling, respectively, and the measured kernel functions are:

(a) `pick_next_task_dl`,

(b) `post_schedule`,

(c) `enqueue_task_rt`,

(d) `pick_next_task_rt`,

(e) `task_tick_rt`.

Function	Hits ($\times 10^3$)	Duration (μs)
(a)	155	1.1 ± 148.4
(b)	155	0.8 ± 65.3
(c)	147	0.25 ± 9.1
(d)	18	1.6 ± 83.7
(e)	29	0.4 ± 3.5

TABLE 5: *Overhead measurements of kernel functions for the applications of Table 2, when using virtual platforms.*

Function	Hits ($\times 10^3$)	Average (μs)
(a)	2251	0.1 ± 17.2
(b)	2251	0.7 ± 51.4
(c)	8	0.7 ± 1.8
(d)	2251	0.2 ± 22.1
(e)	29	0.4 ± 2.2

TABLE 6: *Overhead measurements of kernel functions for the applications of Table 2, when using Linux's throttling.*

As it emerges from Table 5 and Table 6, the overhead of virtual platforms is comparable with that of Linux's throttling mechanism. Notice that virtual platforms result in a lower number of scheduling events w.r.t. the Linux's throttling mechanism.

5 Conclusions

In this paper, we presented an implementation within the Linux kernel of a multiprocessor bandwidth reservation mechanism for control groups implementing the BDM interface and based on SCHED_DEADLINE. First results showed agreement with theoretical analysis and overheads comparable with the cgroups throttling mechanism available in mainline Linux.

As a future work, we want to better characterize the computational costs and the introduced overheads. From the theoretical side, the next step that needs to be addressed is the analysis of shared resources access; a promising approach concerning this problem is the extension of the multiprocessor bandwidth inheritance (M-BWI) protocol proposed in [11] and implemented in [12], in order to support reservations for groups of tasks.

[2]https://github.com/scheduler-tools/rt-app

[3]See `Documentation/trace/ftrace.txt` in the kernel source tree.

References

[1] *Virtual multiprocessor platforms: specification and use*, Enrico Bini, Marko Bertogna and Sanjoy Baruah, RTSS 2009.

[2] *The multi supply function abstraction for multiprocessors*, Enrico Bini, Giorgio Buttazzo and Marko Bertogna, RTCSA 2009.

[3] *A framework for hierarchical scheduling on multiprocessors: from application requirements to runtime allocation*, Giuseppe Lipari and Enrico Bini, RTSS 2010.

[4] *Hard constant bandwidth server: comprehensive formulation and critical scenarios*, Alessandro Biondi, Alessandra Melani and Marko Bertogna, SIES 2014.

[5] *Resource reservation in dynamic real-time systems*, Luca Abeni and Giorgio Buttazzo, RTSJ 2004.

[6] *Schedulability analysis of the Linux push and pull scheduler with arbitrary processor affinities*, Arpan Gujarati, Felipe Cerqueira and Björn Brandenburg, ECRTS 2013.

[7] *A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees*, Hennadiy Leontyev and James H. Anderson, ECRTS 2008.

[8] *Hierarchical scheduling framework for virtual clustering multiprocessors*, Insik Shin, Arvind Easwaran and Insup Lee, ECRTS 2008.

[9] *Schedulability analysis for a real-time multiprocessor system based on service contracts and resource partitioning*, Yang Chang, Robert Davis and Andy Wellings, University of York, Tech. Rep. 2008.

[10] *Compositional Multiprocessor Scheduling: the GMPR interface*, Artem Burmyakov, Enrico Bini and Eduardo Tovar, Real-Time Systems, No. 50(3), May 2014.

[11] *Analysis and implementation of the multiprocessor bandwidth inheritance protocol*, Dario Faggioli, Giuseppe Lipari and Tommaso Cucinotta, Real-Time Systems, No. 48, 2012.

[12] *An implementation of the Multiprocessor Bandwidth Inheritance Protocol on Linux*, Andrea Parri, Juri Lelli, Mauro Marinoni and Giuseppe Lipari, RTLWS 2013.