# Non-Preemptive Access to Shared Resources in Hierarchical Real-Time Systems[*]

Marko Bertogna, Fabio Checconi, Dario Faggioli[†]

November 4, 2008

## Abstract

This paper presents a new strategy to arbitrate the access to globally shared resources in hierarchical EDF scheduled real-time systems, without needing any information on the duration of each critical section. Previous works addressing this problem assumed each task worst-case critical section length be known in advance. However, this assumption is valid only in restricted system domains, and is definitely inadequate for general purpose real-time operating systems. To sidestep this problem, we will instead measure at run-time the amount of time for which a task keeps a resource locked, assuring that there is enough bandwidth to tolerate the interferences associated to such measured blocking times. The protocol will execute each critical section non-preemptively, exploiting a previously proposed server that performs a budget check before each locking operation. Two methods with different complexities will be derived to compute upper-bounds on the maximum time for which a critical section may be executed non-preemptively in a given hierarchical system.

## 1 Introduction

One of the main problems in composing different applications on a dedicated processor is related to the existing inter-dependencies due to the concurrent access to shared resources. When the same resource is accessed by two tasks standing at different hierarchy levels, particular care should be taken in designing a proper shared resource protocol that could avoid the so-called "budget exhaustion" problem. This problem arises when a server finishes its budget while a task is still inside a critical section, and is preempted by other servers. If some of these servers may share the same locked resource, no progress can be made in the system without losing consistency, until the preempted server resumes its execution.

In order to avoid complex protocols to arbitrate the access to shared resources, a good programming practice is to keep the length of every critical section short [17]. If this is the case, preemptions may be disabled while a task is holding a lock, without incurring significant schedulability penalties. In Section 3, we will show how to derive safe upper bounds on the maximum time for which a critical section may be executed non-preemptively. In Section 4, we will then explain how to efficiently use such bounds.

General purpose operating systems, like Linux, complicate, sometimes making it impossible, the task of providing tight estimations of the worst-case critical section lengths for the general workloads they support. These systems often execute a small number of real-time tasks in parallel with many other best effort tasks, and there may be shared resources among the two classes of applications; thus, a shared resource protocol has to be transparent as much as possible, to avoid changes to existing applications.

Our approach tries to deal with such situations, using a runtime estimate for the critical section lengths and adapting the admission parameters to the behavior shown by the already admitted tasks. When an estimate is proven to be too optimistic (i.e., when a task executes inside a critical

[*]This work has been supported in part by the European Commission under the ACTORS (FP7/2008/ICT/216586) and PREDATOR (FP7/2008/ICT/216008) ICT projects.

[†]All authors are with the Scuola Superiore S.Anna, Pisa, Italy. Mail: marko@sssup.it, fabio@gandalf.sssup.it, faggioli@gandalf.sssup.it.

section for more than what it has been estimated when admitting it) there may be deadline misses. The system will nevertheless react to the overload condition by taking appropriate scheduling decisions and accordingly updating the estimations on the critical section lengths.

The computational complexity of the adopted mechanism is a key factor. For the admission control to be really useful in a highly dynamic system, as general purpose operating systems usually are, it has to be implemented inline and it must be able to support a very large number of tasks.

## 2 Related Work

Earlier works addressing the access to shared resources in highly dynamic open environments exist, but they often rely on some simplified task model. For example the solutions presented in [15, 10, 8] assume that the computational demand of each application may be aggregated and represented as a single periodic task, excluding the possibility to address hierarchical systems.

Davis and Burns presented in [9, 5] schedulability tests for open environments using a generalized version of the Stack Resource Policy [2] for hierarchical systems. Even if their analysis is accurate and robust to overload conditions, it is not appropriate for the target of this paper due to two reasons: (i) it is necessary to a priori know the maximum critical section length of each scheduled entity, and (ii) the test has a computational complexity that is not suitable for highly varying workloads in which entities can dynamically join and leave the system.

Another hierarchical-aware mechanism has been proposed by Behnam *et al.* in [5]. Their idea is to suspend a task that does not have enough remaining budget to serve its whole critical section until the next recharging time. A similar approach has been presented by Fisher *et al.* in [11, 12], where a modification of the Constant Bandwidth Server (CBS) of Abeni and Buttazzo [1], called BROE, is presented. Nevertheless, in both approaches, it is again necessary to know in advance the length of the critical sections.

The Bandwidth Inheritance protocol described in [13] is one of the first attempts to avoid such a need. However, it is limited to single task servers, and is therefore not suitable for hierarchical systems.

Remarkable work in this topic has also been carried out Block et al. [7] by means of the Flexible Multiprocessor Locking Protocol (FMLP), which supports resource protection either via spin-based or suspension-based locks, depending whether a resource is defined as short or long. It is an effective solution to handle resource sharing in multiprocessor systems, but it does not yet support hierarchical systems as well.

## 3 Scheduling analysis

Assume that each individual application may be characterized as a collection of sporadic tasks [14, 4], having deadlines equal to periods. Tasks may share resources that are *local* to an application (i.e., only shared within the application) or *global* (i.e., may be shared among different applications). Each application is scheduled in a dedicated server, which can potentially include other servers, forming a hierarchy of "entities". The term entity is introduced to unify the analysis of tasks and servers, according to the following definition.

**Definition 1** (Entity). *We recursively define an* entity $\eta_i$ *as either a sporadic task, or a server in which a set of entities is scheduled. The period $T_i$ of an entity is accordingly identified either by the minimum interarrival time of the sporadic task, or by the period of the server. Similarly, the budget $C_i$ of an entity is either the worst-case computation time of the task, or the server budget.*

An example scenario is depicted in Figure 1, with $\eta_0$ being the root server, $\eta_1$ and $\eta_2$ its child entities, and $\eta_3$ being the only child entity of $\eta_1$. The leaf entities $\eta_2$ and $\eta_3$ are both tasks, while $\eta_1$ is a non-root server, so it acts both as an entity scheduled by its parent server (i.e., $\eta_0$), and a server itself, scheduling its own child $\eta_3$.

To avoid the implementation of complex shared resource protocols, as well as to limit the budget exhaustion problem, we decided to execute each critical section *with system preemptions disabled.* However, to preserve the system schedulability, we will compute for each entity the total amount of time for which preemptions may be safely disabled, exploiting this information during the admission control. If the admission of a new entity would leave enough bandwidth to execute each critical section

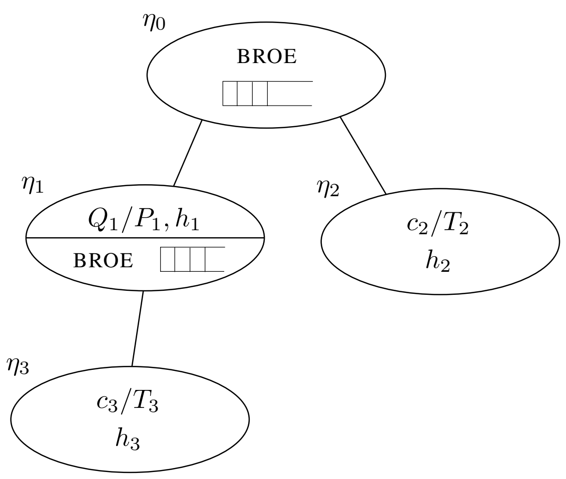

Figure 1: Tree of entities.

non-preemptively, the entity is admitted. Otherwise it is rejected.

As previously mentioned, one of the main problems in designing an efficient shared resource protocol for a hierarchical system is given by the difficulties in deriving tight upper bounds on the time spent in each critical section. Since we do not want to charge the user with the burden of providing such upper bounds, we developed an alternative strategy that is able to efficiently solve this problem.

- First of all, we define a parameter $h_i$ specifying the maximal length for which an entity $\eta_i$ can execute non-preemptively without missing any deadline.
- We then compute a safe lower bound on $h_i$ for each admitted entity.
- We measure the amount of time spent inside a critical section by each executing entity, storing the worst-case critical section length of each entity.
- New entities are admitted at a particular hierarchy level as long as each resulting non-preemptive chunk length is not lower than the corresponding maximal critical section length measured, as we will better describe in Section 4.

We hereafter provide a method to compute a safe lower bound on the maximal non-preemptive chunk length of an entity.

Assume a set $\eta$ of entities $\eta_1, \eta_2, \ldots, \eta_n$ at a particular hierarchy level is scheduled with EDF on a CBS-like server with budget $Q$ and period $P$. Entities are indexed in increasing period order, with $T_i \leq T_{i+1}$. The utilization of an entity is defined as $U_k = \frac{C_k}{T_k}$. Let $T_{min}$ be the minimum period among all entities, and $U_{tot}$ be the sum of the utilizations of all entities.

When entities may share common resources with other tasks or groups, it is important to avoid the situation in which the budget is exhausted while an entity is still inside a critical section. To avoid this problem, a budget check is performed before each locking operation, as in the BROE server described in [11]. If the remaining budget is not sufficient to serve at least the maximum non-preemptive chunk length, the server is suspended, and reactivated as soon as the server capacity can be safely recharged[1]. Otherwise, the critical section is served with the current server parameters.

Before stating our first schedulability result, we need to impose two constraints on each non-preemptive chunk length $h_k$, in order to avoid interfering with other entities in the system. In particular,

- *(i) $h_k$ should not exceed the maximum budget $Q$,*
- *(ii) the computed maximum non-preemptive chunk length at a given hierarchy level should not exceed the same parameter at the parent level.*

Both constraints are needed to avoid the budget exhaustion problem while holding a lock, as well as to preserve the bandwidth isolation properties of the server-based open environment.

The following theorem presents a method to compute a safe bound on the maximum time length $h_k$ for which an entity $\eta_k$ may execute non-preemptively, preserving system schedulability.

**Theorem 1** ($O(n)$)**.** *A set of entities that is schedulable with preemptive* EDF *on a server with budget $Q$ and period $P$ remains schedulable if every entity executes non-preemptively for at most $h_k$ time units, where $h_k$ is defined*

[1]For more details on the recharging mechanisms of the BROE server, please refer to [11]

*as follows, with $h_0 = \infty$:*

$$h_k = \min \left\{ h_{k-1}, \left( \frac{Q}{P} - \sum_{i=1}^{k} U_i \right) T_k - 2(P-Q) \right\}. \tag{1}$$

*Proof.* The proof is by contradiction. Assume a set of entities $\eta$ misses a deadline when scheduled with EDF on a server having budget $Q$ and period $P$, executing every entity $\eta_k$ non-preemptively for at most $h_k$ time-units, with $h_k$ as defined by Equation (1). Let $t_2$ be the first missed deadline. Let $t_1$ be the latest time before $t_2$ in which there is no pending entity with deadline $\leq t_2$. Consider interval $[t_1, t_2]$. Since at start time there are no active entities, the interval is correctly defined, and the processor is never idled in $[t_1, t_2]$. Due to the adopted policy, at most one job with deadline $> t_2$ might execute in the considered interval: this happens if such job is executing in non-preemptive mode at time $t_1$. Let $\eta_{\text{np}}$ be the entity which such job, if any, belongs to. The demand of $\eta_{\text{np}}$ in $[t_1, t_2]$ is bounded by $h_{\text{np}}$. Moreover, $T_{\text{np}} > t_2 - t_1$. Every other entity executing in $[t_1, t_2]$ has instead $T_i \leq (t_2 - t_1)$. Let $\eta_k$ be the entity with the largest period among such entities. Then, $T_k \leq t_2 - t_1 < T_{\text{np}}$, and $h_k \geq h_{\text{np}}$.

Since there is a deadline miss, the total demand in interval $[t_1, t_2]$ must exceed the capacity supplied by the server throughout the same interval. Such capacity[2] cannot be lower than $\frac{Q}{P}(t_2 - t_1) - 2(P - Q)$. Then,

$$h_{\text{np}} + \sum_{i=1}^{k} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i > \frac{Q}{P}(t_2 - t_1) - 2(P-Q).$$

Using $x \geq \lfloor x \rfloor$ and $h_k \geq h_{\text{np}}$, we get

$$h_k + (t_2 - t_1) \sum_{i=1}^{k} U_i > \frac{Q}{P}(t_2 - t_1) - 2(P-Q), \tag{2}$$

$$h_k > (\frac{Q}{P} - \sum_{i=1}^{k} U_i)(t_2 - t_1) - 2(P-Q). \tag{3}$$

And, since $t_2 - t_1 \geq T_k$,

$$h_k > \left( \frac{Q}{P} - \sum_{i=1}^{k} U_i \right) T_k - 2(P-Q),$$

reaching a contradiction. □

[2]The capacity supplied by the server can be bounded from below by a function that is null for $2(P - Q)$ time-units and than increases with slope $\frac{Q}{P}$, as shown in [16].

The above theorem provides a method to compute an upper bound on the time for which an entity can be executed non-preemptively, with a complexity that is linear in the number of entities at the same hierarchy level. This result can be used for the arbitration of the access to shared resources in a hierarchical system: it is possible to execute each critical section non-preemptively, as long as no critical section is longer than the derived bound.

A weaker upper bound on the available non-preemptive chunk length can be computed with a reduced (constant) complexity, as shown by the following corollary.

**Corollary 1** ($O(1)$). *A set of entities that is schedulable with preemptive* EDF *on a server with budget $Q$ and period $P$ remains schedulable if every entity executes non-preemptively for at most*

$$\left( \frac{Q}{P} - U_{tot} \right) T_{\min} - 2(P-Q)$$

*time units.*

The above theorem allows computing a single value $h$ for the allowed maximum non-preemptive chunk length *of all entities* at a given hierarchy level, in a constant time. This lighter tests is a valid option whenever it is important to limit the overhead imposed on the system.

In the following sections, we will compare the solutions given by Theorem 1 and Corollary 1, in terms of schedulability performances and system overhead. Other more complex methods may be used to derive tighter values for the allowed lengths of non-preemptive chunks (see, for instance, the work presented by Baruah in [3]); nevertheless, we chose not to implement such methods due to their larger (pseudo-polynomial) complexity. Having a fast, $O(n)$ method to calculate a global value for $h$ is, in our opinion, really important, as in a highly dynamical system, with thousands of tasks, as Linux can be, it allows our method to be used without significant overhead. Using a global value for $h$ simplifies the implementation and reduces the runtime overhead of the enforcing mechanism, that has not to keep track of the per-task values.

It is worth noting that more sophisticate shared resource protocols like the Stack Resource Policy (SRP) [2] are not so suitable for the target architecture, since they are based on the concept of ceiling of a resource. To properly compute such parameter, it would be necessary to know a priori which task will lock each resource and, in a real operating system, this is definitely not a viable approach from a system design point of view.

## 4 Admission Control

One of the key points of our approach is that there is no need for the user to specify a safe upper bound on the worst-case length of each critical section, something that is very problematic in non-trivial architectures. The system will use all the available bandwidth left by the admitted entities to serve critical sections, automatically detecting the length of each executed critical section, by means of a dedicated timer. If some entity holds a lock for more than the current corresponding non-preemptive chunk length, it means that some deadline may be missed, and the system is overloaded. In this case, some decision should be taken to reduce the system load.

There are many possible heuristics that can be used to remove some entities from the system to solve the overload condition, the choice of which depends on the particular application. For instance, the system may reject entities with heavier utilizations or longer critical sections, leaving enough bandwidth for the admission of lighter entities; it can penalize less critical entities, if such information is available, or the most recently admitted one; or it can simply ask the user what to do. We chose to reject the entity with the largest critical section length, which is the one that triggered such scheduling decision executing for more than the current non-preemptive chunk length.

The system keeps track of the largest critical section at each hierarchy level. The maximum critical section length of a group is recursively defined as the maximum critical section length among all entities belonging to that group. For all deadlines to be met, this value should always be lower than the corresponding non-preemptive chunk length.

The admission control algorithm changes depending on the complexity of the adopted method to compute the time for which a task may execute with preemptions disabled. We distinguish into two cases: (i) using for all entities in the same hierarchy level a single value $h$ given by Corollary 1; or (ii) using for each entity $\eta_i$ a different value $h_i$ given by Theorem 1.

In the first case the system keeps track of the largest critical section among all entities belonging to the same group: $R_{\max} = \max_{\text{group}}\{R_i\}$. For all deadlines to be met, this value should always be lower than the non-preemptive chunk length at the corresponding level:

$$R_{\max} \leq h. \qquad (4)$$

When an entity $\eta_k$ asks to be admitted into the system at a particular hierarchy level, the following operations are performed:

- the new allowed non-preemptive chunk length $h'$ for the considered group of entities after the insertion of the new element is computed using Theorem 1.
- If such value is lower than the maximum critical section length $R_{\max}$ among the entities already admitted in the group, the candidate entity is rejected, since it means that there would not be enough space available to allocate the blocking time of some entity.
- Otherwise, $\eta_k$ is admitted into the system, updating $h$ to $h'$. Note that $R_{\max}$ does not need to be updated, since there is no available estimation of the maximum critical section length of $\eta_k$ (initially, $R_k = 0$).

When instead an entity $\eta_k$ leaves the system, the new (larger) value of $h$ is computed and accordingly updated for the considered group of entities. Moreover, if $R_k = R_{\max}$, $R_{\max}$ may as well be updated (decreased).

The slightly more complex case in which different non-preemptive chunk values $h_i$ are used for each entity $\eta_i$, we will instead proceed as follows. In order to guarantee all deadlines be met, we will check that every entity $\eta_i$ has a non-preemptive chunk length $h_i$ sufficiently large to accommodate the maximum critical section of that entity:

$$\forall i, \;\; R_i \leq h_i. \qquad (5)$$

When an entity $\eta_k$ asks to be admitted into the system at a particular hierarchy level, the following operations are performed within that level:

- using Theorem 1, we compute the allowed non-preemptive chunk length $h'_i$ after the insertion of the new entity, for all entities $\eta_i$ having a period at least as large as $\eta_k$'s: $T_i \geq T_k$.

- If there is at least one value $h'_i$ that is lower than the maximum critical section length of the corresponding entity $\eta_i$ — i.e., $h'_i < R_i$ — the candidate entity $\eta_k$ is rejected.
- Otherwise, $\eta_k$ is admitted into the system, updating each $h_i$ to $h'_i$.

When an entity $\eta_k$ leaves the system, we simply recompute the $h_i$ values of the entities with period greater than $T_k$.

Independently from the adopted strategy, the system will check if an invariant condition (given by Equation (4) or Equation (5)) is maintained. When it is not, some decision should be taken to solve the overload condition.

In a certain sense, we can say that an entity is *conditionally* admitted into the system, and it will remain so as long as it does not show any critical section that is longer than the maximum non-preemptive chunk length allowed, in which case the task is rejected from the system. As we previously mentioned, alternative strategies may instead trigger different scheduling decision when $R_{\max}$ exceeds $h$, for instance creating room for an entity with a long critical section by rejecting different entities.

One last question needs to be answered: what to do if some entity holds a lock for more than the available non-preemptive chunk length. It is worth noting that there is no way to preventively reject an entity that will hold a lock for more than the allowed non-preemptive chunk length, since there is no way to know in advance for how long each lock will be held. We may notice that an entity is executing a critical section for longer than the allowed non-preemptive time interval only at run-time, in which case we can decide to (i) *suspend the entity*, (ii) *abort it*, or (iii) *continue executing it until the lock is released.* Each one of these methods has its advantages and drawbacks. Suspending an entity may increase the blocking time on other entities that share the same locked resource. Aborting a task while inside a critical section may leave the system in an inconsistent state. Continuing to execute the overloading entity non-preemptively may delay other tasks, leading to a deadline miss.

The choice of the adopted methods depends on the addressed application and on the characteristics of the shared resources that are accessed. When no information is available, we chose to continue executing the entity with system preemptions disabled until the lock is released (case (i)). Of course the compositional guarantees will be temporarily violated, but this appears to be the minimum price to pay for the indeterminism of the considered system model. We believe it is better to miss some deadline executing an overloaded entity, rather than leave the shared resource in an unpredictable state (as in case (ii)), or than stall the system due to the budget exhaustion problem (as in case (iii)). This consideration appears to be particularly true when considering nested resources, since a crossed access of nested resources by two different entities could lead to a deadlock condition when an entity is preempted before releasing a lock (case (iii)). Particular measures to avoid deadlocks need therefore to be taken when choosing this method.

## 5 Experimental Results

To evaluate the effectiveness of the proposed method, we performed a set of experiments with various different kinds of randomly generated task sets. We observed the values obtained for the maximum non-preemptiveness intervals given by our tests, trying to understand if they can be employed in real-world scenarios.

### 5.1 Experiment Setup

We focused on a single node of the entity hierarchy, and we analyzed the behavior of the tests when changing the entity's server parameters and the child entities generation parameters. With respect to the server, we considered different values for its $P$ and $Q$, ranging from low utilization over short periods (e.g., 5ms every 100ms), to medium utilization over long periods (e.g., 500ms every 1s). The child entity set parameters we considered were the total utilization $U$, given as a fraction of the parent's $Q/P$ ratio, the number of child entities, and for each entity, the period $T_k$ and the computation time $C_k$.

We considered entity sets composed by 3 and 10 elements, and we varied the entity set utilization from 0.1 to 0.9, in steps of size 0.1. For each configuration we generated the entities periods $T_k$ from a uniform distribution over intervals $[10P, 100P]$ or $[20P, 500P]$, and the entities utilizations $U_k$ using the method described in [6]. The values for the entities computation times where then derived as $C_k = U_k * T_k$.

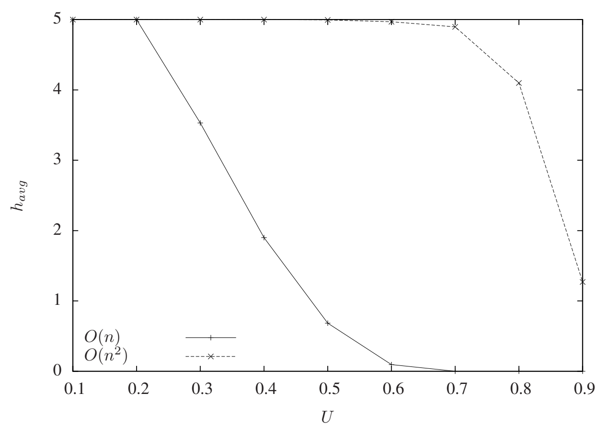

Figure 2: Large periods, small utilization.

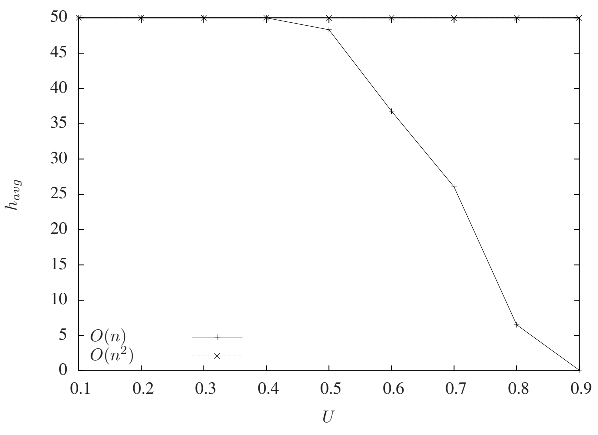

Figure 3: Small periods, high utilization.

## 5.2 Evaluation of Experiments

First of all, here we report only a fraction of the results, as the conclusions that may be obtained from the ones we show are confirmed by the remaining ones.

We report the results only for entity sets of 10 elements; with a lower number of entities, results are better, since $T_{min}$ tends to be larger, resulting in larger values of $h$.

Figure 2 shows the average $h$ value obtained generating child entities with large periods (in the $[20P, 500P]$ range), inside a server with small utilization, with $Q = 5$ and $P = 100$. For the $O(n)$ test the plotted average is taken on the minimum $h_k$ obtained.

From the figure it is clear that the $O(1)$ test starts degrading very early when utilization increases, giving pretty soon small values of $h$. On the other hand, the $O(n)$ test shows a good behavior in this scenario, giving a worst-case $h_k$ that is close to the upper bound $Q$ for medium-to-high utilizations (up to $0.7$).

Figure 3 shows a scenario with a server having a large utilization, with $Q = 50$ and $P = 100$; as we could expect, increasing the server budget produces higher values for $h$, with both tests. It is worth noting that the above results were obtained with small values for $T_{min}$, as $T_k$ were generated according to a uniform distribution in $[5P, 50P]$; bigger values for $T_{min}$ resulted in even bigger values for $h$.

In this scenario the $h_k$ values obtained with the $O(n^2)$ test are always close to the $Q$ value, and even the $O(n)$ test gives large $h$ values until $U = 0.6$.

## 6 Conclusions

In this paper we presented a combined method to arbitrate the access to shared resources in a hierarchical system composed by EDF scheduled sporadic tasks. The advantage of our approach are manyfold. There is no need for the user to specify the length of the critical sections that are accessed. The protocol for the arbitration of the access to shared resources is very simple, executing each critical section non-preemptively and limiting the overhead to a few bookkeeping operation. A simple CBS-like server is used to solve the budget exhaustion problem, limiting the interference imposed on each entity, improving the system schedulability. We presented two methods with different complexities for the computation of the available non-preemptive chunk length to accommodate the non-preemptive execution of the critical sections. The reported simulations show the effectiveness of both methods.

## References

[1] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the Real-Time Systems Symposium*,

Madrid, Spain, December 1998. IEEE Computer Society Press.

[2] Theodore P. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems: The International Journal of Time-Critical Computing*, 3, 1991.

[3] Sanjoy Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 137–144, Palma de Mallorca, Balearic Islands, Spain, July 2005. IEEE Computer Society Press.

[4] Sanjoy Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th Real-Time Systems Symposium*, Orlando, Florida, 1990. IEEE Computer Society Press.

[5] Moris Behnam, Insik Shin, Thomas Nolte, and Mikael Nolin. Sirap: a synchronization protocol for hierarchical resource sharingin real-time open systems. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, 2007.

[6] Enrico Bini and Giorgio C. Buttazzo. Biasing effects in schedulability measures. In *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems*, Catania, Italy, July 2004.

[7] A. Block, H. Leontyev, B. Brandenburg, and James Anderson. A flexible real-time locking protocol for multiprocessors. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, August 2007.

[8] Marco Caccamo and Lui Sha. Aperiodic servers with resource constraints. In *Proceedings of the IEEE Real-Time Systems Symposium*, London, UK, December 2001. IEEE Computer Society Press.

[9] Robert I. Davis and Alan Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proceedings of the IEEE Real-time Systems Symposium*, Rio de Janeiro, 2006. IEEE Computer Society.

[10] D. de Niz, L. Abeni, S. Saewong, and R. Rajkumar. Resource sharing in reservation-based systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, London, December 2001. IEEE Computer Society Press.

[11] Nathan Fisher, Marko Bertogna, and Sanjoy Baruah. The design of an EDF-scheduled resource-sharing open environment. In *28th IEEE Real-Time Systems Symposium (RTSS)*, Tucson, Arizona (USA), 2007.

[12] Nathan Fisher, Marko Bertogna, and Sanjoy Baruah. The design of an EDF-scheduled resource-sharing open environment. Technical report, Department of Computer Science, The University of North Carolina at Chapel Hill, 2007. Available at `http://www.cs.unc.edu/~fishern/pubs.html`.

[13] Giuseppe Lipari, Gerardo Lamastra, and Luca Abeni. Task synchronization in reservation-based real-time systems. *IEEE Trans. Computers*, 53(12):1591–1601, 2004.

[14] Aloysius K. Mok. *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983. Available as Technical Report No. MIT/LCS/TR-297.

[15] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource kernels: a resource-centric approach to real-time and multimedia systems. pages 476–490, 2001.

[16] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE Computer Society, 2003.

[17] Victor Yodaiken. Against priority inheritance. Technical report, Finite State Machine Labs, June 2002.