Task Synchronization in Reservation-Based Real-Time Systems

Giuseppe Lipari, *Member*, *IEEE*, Gerardo Lamastra, *Member*, *IEEE*, and Luca Abeni, *Member*, *IEEE*

Abstract—In this paper, we present the BandWidth Inheritance (BWI) protocol, a new strategy for scheduling real-time tasks in dynamic systems, which extends the resource reservation framework to systems where tasks can interact through shared resources. The proposed protocol provides temporal isolation between independent groups of tasks and enables a schedulability analysis for guaranteeing the performance of hard real-time tasks. We show that BWI is the natural extension of the well-known Priority Inheritance Protocol to dynamic reservation systems. A formal analysis of the protocol is presented and a guarantee test for hard real-time tasks is proposed that takes into account the case in which hard real-time tasks interact with soft real-time tasks.

Index Terms—Real-time scheduling, dynamic system, resource reservation, priority inheritance, constant bandwidth server.

1 Introduction

The main goal of a real-time scheduler is to provide temporal guarantees. In a hard real-time system, the scheduler must guarantee that, under certain worst-case assumptions, the temporal constraints of all tasks are respected. When mixing hard, soft, and non-real-time tasks, providing such temporal guarantees becomes a complex problem. In *dynamic real-time systems*, tasks can be activated dynamically and the system has no a priori knowledge about their run-time behavior. Classical real-time schedulability analyses are not appropriate for dynamic real-time systems because they require a priori knowledge of the characteristics of all the tasks to guarantee that every hard real-time task will meet its deadline.

The resource reservation framework is a class of techniques that have been proven very effective in jointly scheduling hard real-time (HRT) and soft real-time (SRT) tasks. In particular, these approaches provide 1) *temporal isolation* between tasks and 2) *schedulability analysis* for HRT tasks. However, tasks are assumed to be independent. This is a severe limitation that hinders their utilization in real operating systems. The extension of the underlying model to cope with tasks that access shared resources through mutually exclusive (mutex) semaphores has only recently been addressed [1], [2], [3].

In this paper, a new protocol, BandWidth Inheritance (BWI), is presented. It extends the Constant Bandwidth Server (CBS) algorithm [4] to real-time tasks that can access shared resources via critical sections, by using a technique derived from the Priority Inheritance Protocol (PIP) [5].

- *G. Lipari is with the Retis Lab, Scuola Superiore Sant'Anna, piazza Martiri della Libertà 33, 56100, Pisa (PI), Italy. E-mail: lipari@sssup.it.*
- *G. Lamastra is with Telecom Italia Lab, G. Reiss Romoli 274, 10148 Torino (TO), Italy. E-mail: Gerardo.Lamastra@tilab.com.*
- *L. Abeni is with Broadsat S.r.l., viale Garibaldi 54, 51017, Pescia (PT), Italy. E-mail: lucabe72@email.it.*

Manuscript received 11 June 2002; revised 17 Mar. 2003; accepted 27 May 2004.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 116736.

In the remainder of this paper, we describe the BWI protocol and its properties. Then, we provide a schedulability analysis for HRT tasks. This protocol does not require any a priori knowledge of each task's behavior. Only the analysis is based on the knowledge of the worst-case behavior of the HRT tasks. Hence, the BWI protocol is suitable for a real operating system.

2 Related Work

Deng and Liu [6] proposed a model of dynamic real-time systems called the *open system* model. In their model, an application is a set of tasks and a *server* (i.e., an algorithm of the class of the aperiodic server algorithms [7], [8], [9]) is used to handle each different application. A customized scheduler can be associated with each server in a hierarchical way. However, the algorithms proposed by Deng and Liu require a priori knowledge of the tasks' execution times even for SRT tasks.

A better approach is to provide *temporal isolation* between tasks such that each task is protected from the misbehaviors of the other tasks. This property is also called *Bandwidth Isolation* (BIP) [4], [10]. The net effect is that each task executes as though it were executing on a slower dedicated processor. The BIP can be implemented using the *resource reservation framework* [11], [12]. In this framework, when a task arrives in the system requiring a certain level of Quality of Service (QoS), an *admission test* is run. If, given the current system load, the required level of QoS can be guaranteed, the task is accepted in the system and assigned a certain fraction of the system resources. Many scheduling algorithms based on the resource reservation framework have been proposed. The temporal isolation and the resource reservation concepts were formally introduced for the first time in the RT-Mach operating system [11].

The Resource Kernel (RK) [12] is the most complete work to explicitly address the temporal isolation problem. An RK allows each task to reserve a hardware resource for a certain amount of time C every interval of time of length T. The RK

0018-9340/04/$20.00 © 2004 IEEE Published by the IEEE Computer Society

technology is particularly interesting since it seamlessly provides QoS guarantees even to non-real-time legacy applications and it has been recently ported to Linux [13].

Resource-reservation techniques, originally based on a fixed-priority scheme, have also been applied to dynamic priority schemes like the Earliest Deadline First (EDF) scheduler [14], [4]. One of the advantages of using EDF is that it is an optimal scheduler and permits very high processor utilization [15]. The Constant Bandwidth Server (CBS) [4] is based on EDF and uses a deadline postponing mechanism to efficiently provide the BIP. In particular, the CBS algorithm enables a schedulability analysis for HRT tasks and probabilistic analysis for SRT tasks.

All the algorithms cited so far support only independent tasks. This is a major limitation for their implementation in an operating system. In the real world, many real-time tasks communicate by means of shared memory protected by mutex semaphores.

Many researchers addressed the problem of providing guarantees to hard real-time tasks accessing shared resources. It has been proven that, if classical mutex semaphores are used, a particular problem arises, known as *priority inversion*. This problem was first described by Sha et al. [5], who proposed two solutions, the *Priority Inheritance Protocol* (PIP) and the *Priority Ceiling Protocol* (PCP). Similar protocols have been proposed for dynamic priority schedulers [16], [17], [18].

Applying these techniques to resource reservations is not trivial. For example, the CBS algorithm assumes a very simple task model, where no task can suspend itself or block on a critical section. If tasks are allowed to suspend, the properties of the CBS algorithm are not valid anymore (see Section 4 for more details).

Kuo and Li [19] propose an extension to the open system model presented in [6] that accounts for shared resources. In [1], it is shown how to combine the Stack Resource Policy (SRP) [17] with aperiodic servers, in order to share resources between aperiodic tasks and hard real-time tasks. This approach has been recently applied to the CBS algorithm [2]. However, these solutions require a priori knowledge of the maximum resource usage time for each task; otherwise, the protocol may not work properly. This information may not be available in dynamic real-time systems.

The problem of priority inversion in reservation-based systems was also considered by De Niz et al. [3]. The reserve-inheritance approach proposed therein resembles the approach proposed in this paper. However, while their methodology is based on fixed priority scheduling, our protocol is based on dynamic priority scheduling. In addition, in this paper, we propose a new sufficient schedulability analysis for HRT tasks, even when they interact with SRT tasks.

3 System Model

3.1 Definitions

A *real-time task* τ_i is a stream of *jobs*, or *instances*, $J_{i,j}$ (where $J_{i,j}$ is the jth job of task τ_i); each job is a request for execution on a shared processor and is characterized by an *arrival time* $a_{i,j}$, an *execution time* $c_{i,j}$, and an *absolute deadline* $d_{i,j}$. A real-time task is also assigned a *relative deadline* D_i and $d_{i,j} = a_{i,j} + D_i$. We denote the *finishing time* of job $J_{i,j}$ with $f_{i,j}$.

Real-time tasks can be *hard* or *soft*. A task τ_i is said to be a hard real-time (HRT) task if all its jobs must complete within their deadline ($\forall j \;\; f_{i,j} \leq d_{i,j}$); otherwise, a critical failure may occur in the system. HRT tasks need to be guaranteed a priori. In order to perform a schedulability test, every HRT task τ_i is characterized by a *minimum interarrival time* $T_i = \min_j\{a_{i,j+1} - a_{i,j}\}$ and a *worst-case execution time* (WCET) $C_i = \max_j\{c_{i,j}\}$.

A task is said to be a soft real-time (SRT) task if it can tolerate an occasional deadline miss. Usually, it is difficult to compute the WCET for an SRT task. Moreover, in most cases, the WCET of an SRT task is much higher than its average-case execution time. Therefore, to allocate the processor to an SRT based on its WCET is often considered a waste of resources. At the price of an occasional deadline miss, SRT tasks are allocated less bandwidth than the worst case.

A task may access shared resources using mutex semaphores. If task τ_i accesses a resource R_k, we say that τ_i uses R_k. Without loss of generality, in the remainder of this paper, we will denote the resource and the corresponding mutex semaphore with the same symbol R_k. A critical section on resource R_k is a section of code delimited by *wait* and *signal* operations on the corresponding semaphore, denoted by $P(R_k)$ and $V(R_k)$, respectively. Critical sections can be *nested*, i.e., it is possible to access resource R_j while holding the lock on resource R_k. We assume *properly nested* critical sections, i.e., a sequence of code like $P(R_1), \ldots, P(R_2), \ldots, V(R_2), \ldots, V(R_1)$ is valid, whereas a sequence like $P(R_1), \ldots, P(R_2), \ldots, V(R_1), \ldots, V(R_2)$ is not valid. *Internal* critical sections are nested inside other critical sections, whereas *external* critical sections are not. We denote the worst-case execution time of the longest critical section of task τ_i on resource R_k as $\xi_i(R_k)$. Note that $\xi_i(R_k)$ comprises the execution time of all the nested critical sections. We also assume that, if a job performs a wait operation on semaphore R_k, it performs the corresponding signal operation before its completion.

In this paper, we propose to schedule each task through a *server*. A server is an abstraction used by the scheduler to store the scheduling parameters of a task. It is characterized by a priority and it can be inserted in the system ready queue. Each task is associated with a server and, when the server is selected by the scheduler, the associated task is dispatched. Therefore, the server can be seen as a *wrapper* for a task. For the sake of simplicity, in the remainder of the paper, we will write "server S_i is blocked," meaning that the task served by S_i is blocked, and "server S_i is executing," meaning that the task served by S_i is executing. Each server is characterized by a budget Q_i and a period P_i, with the interpretation being that each task is allowed to execute for Q_i out of every P_i time units. The fraction $U_i = \frac{Q_i}{P_i}$ represents the *share* (or *bandwidth*) of the processor reserved for server S_i. Our attention is restricted to systems in which all servers execute on a

single shared processor and the sum of the processor shares of all the servers is no more than one:

$$\left(\sum_{i=1}^{n}\frac{Q_i}{P_i}\right) \leq 1. \quad (1)$$

3.2 Requirements

In this paper, we present a novel scheduling policy and the schedulability analysis for HRT tasks. As a general rule, the scheduling algorithm should not be based on the knowledge of the temporal behavior of the tasks. However, it must allow a schedulability analysis. This separation of concerns between scheduling algorithm and schedulability analysis is very useful in dynamic real-time systems. Our goal is to find a scheduling algorithm that is able to provide the BIP *without making any assumption on the temporal behavior of the tasks*. If it is possible to exactly characterize a priori the temporal behavior of a task, then the proposed schedulability analysis can be used to compute the server's budget and period that guarantee the task's deadlines. However, if our analysis is not correct, the BIP guarantees that the other tasks in the system will not be affected.

Therefore, our scheduling algorithm must fulfill the following requirements:

- The arrival times of the jobs (the $a_{i,j}$s) are not known a priori, but are only revealed online during system execution. Hence, our scheduling strategy cannot require knowledge of future arrival times. For example, we do not require the tasks to be periodic.
- The exact execution requirements $c_{i,j}$ are also not known beforehand. They can only be determined by actually executing $J_{i,j}$ to completion. Nor do we require an a priori upper bound (a "worst-case execution time") on the value of $c_{i,j}$.
- The scheduling algorithm has no a priori knowledge of which resources a task will access; it can only be known online when the task tries to lock a resource. Nor do we require any a priori upper bound on the worst-case execution time $\xi_{i,j}$ of a critical section.

The last two assumptions are important because they rule out the use of protocols like the PCP. See Section 6.2 for a discussion on the PCP and the deadlock problem.

The following information is needed only for performing a schedulability analysis on an HRT task τ_i:

- the worst-case computation time C_i;
- the period T_i;
- the type (HRT or SRT) of every task that (directly or indirectly) interacts with τ_i (see Section 6.1 for a definition of interaction);
- for each interacting task τ_j and for each shared resource R_k, the worst-case execution time $\xi_j(R_k)$ of the longest critical section of τ_j on R_k.

4 The Constant Bandwidth Server

In this section, a brief overview of the Constant Bandwidth Server (CBS) algorithm is given. A fully detailed description can be found in [4]. A server S_i is described by two parameters: *the server maximum budget* Q_i, and the *server period* P_i. The *server bandwidth* $U_i = \frac{Q_i}{P_i}$ is the fraction of the CPU bandwidth assigned to S_i. The algorithm dynamically updates two variables (q_i, δ_i) for each server S_i: q_i is the server's *current budget* and keeps track of the consumed bandwidth; δ_i is the server's *current scheduling deadline*. Initially, q_i is set to the maximum budget Q_i and δ_i is set to 0. A server is *active* if the corresponding task has a pending instance.

All tasks in the system are assumed to be *independent* and no task is allowed to suspend itself waiting for a shared resource or a synchronisation event. The system consists of n servers and a global scheduler based on the Earliest Deadline First (EDF) priority assignment. At each instant, the active server with the earliest scheduling deadline δ_i is selected and the corresponding task is dispatched to execute.

The CBS algorithm updates its variables as follows:

- **Rule A:** When job $J_{i,j}$ of task τ_i arrives at time $a_{i,j}$, the server checks the following condition:

  $$q_i \leq Q_i \frac{\delta_i - a_{i,j}}{P_i}.$$

  If the condition is verified, the current pair (q_i, δ_i) is used. Otherwise, a new pair (q_i, δ_i) is computed as $q_i \leftarrow Q_i$ and $\delta_i \leftarrow a_{i,j} + P_i$.
- **Rule B:** If server S_i executes for Δt units of time, the budget is decreased accordingly: $q_i \leftarrow q_i - \Delta t$.
- **Rule C:** Server S_i is allowed to execute while $q_i > 0$. When the budget is exhausted ($q_i = 0$) and the served job has not finished yet, a new pair (q_i, δ_i) is computed: The scheduling deadline is postponed to $\delta_i \leftarrow \delta_i + P_i$ and the budget is recharged to $q_i \leftarrow Q_i$. Since the scheduling deadline has changed, the EDF queue may be reordered and a preemption may occur.

If Rule C is never applied to server S_i (i.e., it is never the case that $q_i = 0$ and the task has not yet finished), we say that server S_i *never postpones its scheduling deadline*.

The CBS algorithm has two important properties: the *bandwidth isolation property* (BIP) and the *hard schedulability property* (HSP). The BIP ensures that each server S_i will contribute to the total system utilization for no more than U_i. The HSP ensures that it is possible to independently verify the schedulability of each HRT task. For proofs of the following theorems, please refer to [20] and [10].

Theorem 1 (Bandwidth Isolation Property). *Given a system of n servers, with $\sum_{i=1}^{n} U_i \leq 1$, no server misses its scheduling deadline, regardless of the behavior of the tasks.*

Theorem 2 (Hard Schedulability Property). *If an HRT task τ_i is served by a server $S_i = (Q_i, P_i)$, with maximum budget $Q_i \geq C_i$ and period $P_i \leq T_i$, then server S_i never postpones its scheduling deadline and each job will complete before its absolute deadline.*

The underlying assumption used to prove the previous theorems is that the executing server is the one with the earliest deadline. If a task is blocked on a semaphore, then this assumption is violated and the previous properties do not hold anymore.

5 The Priority Inheritance Protocol

The Priority Inheritance Protocol (PIP) was first presented in [5] to solve the priority inversion problem. According to this protocol, when a high-priority job J_H wants to access a critical section that is already held by a low-priority job J_L, the latter *inherits* the priority of J_H. When J_L unlocks the resource, it returns to the priority it had at the time it acquired the lock.

Even though the PIP was developed in the context of fixed priority scheduling, it can be applied in the context of dynamic priority scheduling as well. The following basic properties hold:

- *A job J_H can be blocked by a lower priority job J_L for at most the worst-case execution time of one critical section of J_L, regardless of the number of semaphores shared by J_H and J_L.*
- *A job J_i can encounter blocking by at most one critical section for each semaphore that it tries to lock.*

Using these properties, it is possible to give a sufficient condition for the schedulability of a set of n HRT periodic tasks. Suppose that the tasks are ordered by nondecreasing periods ($T_i < T_j \Rightarrow i < j$). The schedulability condition is the following [21], [22]:

$$\forall i \quad 1 \le i \le n \quad \sum_{j=1}^{i} \frac{C_j}{T_j} + \frac{B_i}{T_i} \le 1, \tag{2}$$

where B_i is the worst-case blocking time of task τ_i.

5.1 Using the Priority Inheritance Protocol with the CBS

When applying the PIP to the CBS, it is not clear how to account for the blocking time. One possible way would be to consider the blocking time using the following admission test:

$$\forall i \quad 1 \le i \le n \quad \sum_{j=1}^{i} \frac{Q_j}{P_j} + \frac{B_i}{P_i} \le 1, \tag{3}$$

where B_i represents the maximum blocking time experienced by each server.

However, this solution is not suitable for a dynamic system. In fact, to compute the maximum blocking time of each server, when a task is created we must "declare" the worst-case execution time of the critical sections on each accessed resource. This is in contrast with the goal of a scheduler that must be independent of the actual requirements of the tasks. In addition, if an SRT task holds a critical section for longer than declared, *any server* can miss its deadline.

Example 1. To highlight this problem, consider the example shown in Fig. 1. In this example, there are three servers, $S_1 = (2, 6)$, $S_2 = (2, 6)$ and $S_3 = (6, 18)$. Server S_1 is assigned task τ_1, which accesses a resource R for the entire duration of its jobs (i.e., two units of time). Server S_2 is assigned task τ_2, which does not use any resource. Server S_3 is assigned task τ_3, which has an execution time of six units of time and accesses resource R for five units of time. Now, suppose that τ_3 is an SRT task that claims to use resource R for only two units of time. The system computes a maximum blocking time $B_1 = B_2 = 2$ for servers S_1 and S_2. According to (3), the system is schedulable and all servers can be admitted.

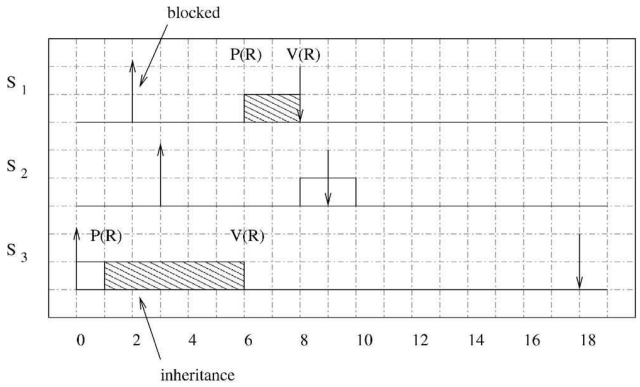


Fig. 1. In this example, blocking times are not correctly accounted for.

In the configuration of arrival times shown in Fig. 1, server S_1 arrives at time t_2 and tries to access R. Since it is locked, server S_3 inherits a deadline $\delta'_3 = 8$ and continues executing. If no enforcement is put on the worst-case execution time of the critical section of task τ_3 on resource R, server S_2 misses its deadline, as is shown in Fig. 1. The simple fact that τ_3 executes more than expected inside the critical section invalidates the BIP and task τ_2, which does not use any resource, misses its deadline.

Another problem that must be considered is the depletion of the server budget while the task is in a critical section and has inherited the deadline of another server. In the original CBS formulation, the server deadline is postponed and the server budget is immediately recharged. When the PIP is applied, it is not clear which deadline has to be postponed.

To solve the problems mentioned above, we combine the PIP and the CBS in a single protocol called BandWidth Inheritance (BWI). The basic idea is that, when a task that executes inside a low-priority server blocks a high-priority server, it inherits the pair (q, δ) of the blocked server.

6 The Bandwidth Inheritance Protocol

Before starting with the description of the Bandwidth Inheritance protocol, we need to understand the meaning of *temporal isolation* when considering interacting tasks. In the original CBS paper [4], tasks are assumed to be independent and, hence, do not interact in any way. When tasks access shared resources, they cannot be considered completely independent anymore. What does *isolation* mean in such a scenario?

Consider again the example shown in Fig. 1. Server S_1 and server S_3 share a resource. It is easy to see that if S_3 holds the lock for longer than declared, some task will probably miss its deadline. Our goal is to prevent task τ_1 and τ_3 from interfering with τ_2. In fact, since τ_1 and τ_3 both access the same resource, it is impossible to provide isolation among them.

6.1 Bandwidth Isolation in the Presence of Shared Resources

In this section, we define more precisely the concept of *interaction* between tasks. Intuitively, a task τ_i can be affected by a task τ_j if it can be directly or indirectly blocked by τ_j. This relation is formalized by the following definition:

Definition 1. *A sequence* $H_i = (\tau_1, R_1, \tau_2, R_2, \ldots, R_{z-1}, \tau_z)$, *with* $z \geq 2$, *is a* blocking chain on task τ_i *if:*

- $\tau_i = \tau_1$;
- $\forall k = 1, \ldots, z-1$, τ_k *and* τ_{k+1} *both use* R_k; *and*
- *If* $z > 2$ $\forall k = 2, \ldots, z-1$, τ_k *accesses* R_k *with a critical section that is nested inside a critical section on* R_{k-1}.

If $z = 2$, then τ_i and τ_z directly share a resource. If $z > 2$, then τ_i and τ_z interact through nested critical sections.

As an example, consider the blocking chain $H_1 = (\tau_1, R_1, \tau_2, R_2, \tau_3)$:

- Task τ_3 uses resource R_2;
- Task τ_2 uses R_2 with a critical section that is nested inside the critical section on R_1; and
- Task τ_1 uses R_1.

Notice that, in the above example, τ_1 can be blocked by τ_2 and by τ_3, but τ_3 cannot be blocked by τ_1. Hence, a blocking chain defines an antisymmetric relation $\models$ between τ_i and τ_z: $\tau_i \models \tau_z$ but not vice versa.

In general, there can be more than one chain between two tasks τ_i and τ_j because they can directly or indirectly share more than one resource. Let us enumerate the chains starting from task τ_i in any order. H_i^h denote the hth blocking chain on τ_i. Without loss of generality, in the remainder of the paper, we will sometimes drop the superscript on the chain.

Let $\Gamma(H_i)$ denote the set of tasks $\tau_2, \ldots, \tau_z$ in the sequence H_i (τ_i excluded), and $R(H_i)$ the set of resources $R_1, \ldots, R_{z-1}$ in the sequence H_i, respectively.

Definition 2. *The set* Γ_i *of tasks that may interact with* τ_i *is defined as follows:*

$$\Gamma_i = \bigcup_h \Gamma(H_i^h).$$

Set Γ_i comprises all tasks that may directly or indirectly block τ_i.

Given these definitions, we can state more precisely the goals of our scheduling strategy. Whether task τ_i meets its deadlines should depend only on the timing requirements of τ_i and on the worst-case execution time of the critical sections of the tasks in Γ_i. Therefore, in order to guarantee an HRT task τ_i, it is only necessary to know the behavior of the tasks in Γ_i.

6.2 The Priority Ceiling Protocol and the Problem of Deadlock

If we allow nested critical section, the problem of deadlock must be taken into account. Deadlock can be avoided by means of static or dynamic policies. One possibility is to use a protocol, like the PCP, that prevents deadlock from occurring.

However, the PCP uses a priori information on the task parameters. For each resource, it requires the knowledge of the priorities of all the accessing tasks in order to compute the *resource ceiling*. The resource ceilings are then used by the protocol to regulate access to resources. Therefore, the requirements of the PCP are in contrast with the initial requirements we made for our scheduling discipline. In addition, if the ceilings are not computed correctly, *not only the analysis but also the resulting schedule is incorrect*. An example can be easily built in which, due to an SRT task that fails to declare access to one resource, the ceilings are not computed correctly and a deadlock can occur.

On the other hand, the PIP does not require any a priori information. Hence, in this paper, we select the PIP as the basic resource access protocol.

To solve the deadlock problem, we consider another static policy. We assume that resources are totally ordered and each task respects the ordering in accessing nested critical sections. Thus, if $i < j$, then task τ can access a resource R_j with a critical section that is nested inside another critical section on resource R_i. When such an order is defined, the sequence of resources in any blocking chain is naturally ordered. For a deadlock to be possible, a blocking chain must exist in which there is a circular relationship like $H = (\ldots, R_i, \ldots, R_j, \ldots R_i, \ldots)$. Therefore, if the resources are ordered a priori, a deadlock cannot occur.

If the total order is not respected when accessing nested critical sections, a deadlock can still occur. As we will see in the next section, our scheduler is able to detect it during runtime, but the action to be taken depends on the kind of resources. In the remainder of the paper, we shall assume that resources are ordered.

6.3 Description of the Bandwidth Inheritance Protocol

The BWI protocol allows tasks to be executed on more than one server. Every server S_i maintains a list of served tasks. During runtime, it can happen that a task τ_i is in the list of more than one server. Let $e(i,t)$ be the index of the earliest deadline server among all the servers that have τ_i in their list at time t. Initially, each server S_i has only its own task τ_i in the list, hence, $\forall i\ e(i,0) = i$.[1] We call server S_i the *default server* for task τ_i.

As long as no task is blocked, BWI follows the same rules as Algorithm CBS. In addition, BWI introduces the following rules:

- **Rule D:** If task τ_i is blocked when accessing a resource R that is locked by task τ_j, then τ_j is added to the list of server $S_{e(i,t)}$. If, in turn, τ_j is currently blocked on some other resource, then the chain of blocked tasks is followed and server $S_{e(i,t)}$ adds all the tasks in the chain to its list until it finds a ready task.[2] In this way, each server can have more than one task to serve, but only one of these tasks is not blocked.

1. Note that index i denotes the task's index when it is the argument of function $e()$ and the server's index when it is the value of $e()$.

2. If, by following the chain, the protocol finds a task that is already in the list, a deadlock is detected and an exception is raised.

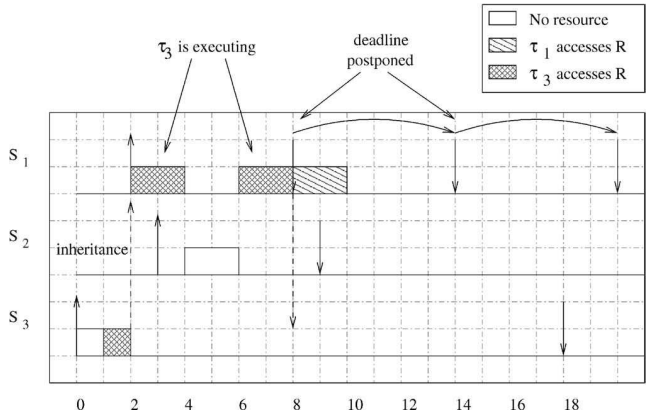


Fig. 2. BWI is applied to the example of Fig. 1.

- **Rule E:** When task τ_j releases resource R, if there is any task blocked on R, then τ_j was executing inside a server $S_{e(j,t)} \neq S_j$. Server $S_{e(j,t)}$ must now discard τ_j from its own list and the first blocked task in the list is now unblocked, let it be τ_i. All the servers that added τ_j to their list while τ_j was holding R must discard τ_j and add τ_i.

BWI is an inheritance protocol: When a high-priority task τ_i is blocked by a lower-priority task τ_j, τ_j *inherits* server $S_{e(i,t)}$ and the execution time of τ_j is then charged to $S_{e(i,t)}$. Therefore, $S_{e(j,t)} = S_{e(i,t)}$. When the budget of $S_{e(i,t)}$ is exhausted, $S_{e(i,t)}$'s deadline is postponed and τ_j can continue to execute on server $S_{e(j,t)}$ that may now be different from $S_{e(i,t)}$.

Example 2. The behavior of BWI is demonstrated by applying the protocol to the example of Fig. 1. The resulting schedule is depicted in Fig. 2.

- At time $t = 1$, task τ_3, which is initially served by S_3, locks resource R.
- At time $t = 2$, server S_1 becomes the earliest deadline server and dispatches task τ_1, which immediately tries to lock resource R. However, the resource is already locked by τ_3. According to Rule D, τ_3 is added to the list of S_1 and τ_1 is blocked. Now, there are two task in S_1's list, but only τ_3 is ready. Hence, $S_{e(3,2)} = S_1$ and S_1 dispatches task τ_3.
- At time $t = 3$, server S_2 is activated, but it is not the earliest deadline server. Thus, τ_3 continues to execute.
- At time $t = 4$, the budget of server S_1 has been depleted. According to Rule C (see Section 4), the server deadline is postponed to $\delta_1 \leftarrow \delta_1 + P_1 = 14$ and the budget is recharged to $q_1 \leftarrow Q_1 = 2$. Since S_1 is no longer the earliest deadline server, S_2 is selected and task τ_2 is dispatched.
- At time $t = 6$, S_1 is again the earliest deadline server; hence, task τ_3 is dispatched.
- At time $t = 8$, τ_3 releases the lock on R. According to Rule E, τ_1 is unblocked and τ_3 is discarded from the list of server S_1. Now, S_1's list contains only task τ_1 and $S_{e(3,2)} = S_3$. Server S_1 is still the earliest deadline server, but its budget has been depleted. According to Rule C, $\delta_1 \leftarrow \delta_1 + P_1 = 20$ and $q_1 \leftarrow Q_1 = 2$.

Note that, in this case, task τ_2 is not influenced by the misbehavior of τ_3 and completes before its deadline.

6.4 Properties of BWI

In this section, the bandwidth isolation property and the hard schedulability property are extended to consider interacting tasks and they are proven for the BWI protocol. We also derive sufficient conditions for assigning server parameters that guarantee HRT tasks. First, we present some preliminary results.

Lemma 1. *Each active server always has exactly one ready task in its list.*

Proof. Initially, no task is blocked and the lemma is true. Suppose that the lemma holds just before time t_b, when task τ_i is blocked on resource R by task τ_j. After applying Rule D, both servers S_j and S_i have task τ_j in their list and task τ_i is blocked. By definition of $e(j, t_b)$, $S_{e(j,t_b)} = S_i$. Moreover, if τ_j is also blocked on another resource, the blocking chain is followed and all the blocked tasks are added to S_i until the first nonblocked task is reached. The lists of all the other servers remain unchanged, thus the lemma is true.

Now, suppose that the lemma is true just before time t_r. At this time, task τ_j releases the lock on resource R. If no other task was blocked on R, then the lists of all the servers remain unchanged. Otherwise, suppose that task τ_i was blocked on R and is now unblocked: Server S_i has τ_j and τ_i in its list and, by applying Rule E, discards τ_j. The lists of all the other servers remain unchanged, and the lemma holds. □

Theorem 3. *Consider a system consisting of n servers with $\sum_{i=1}^{n} U_i \leq 1$, which uses the BWI protocol for accessing shared resources. No server in the system misses its scheduling deadline.*

Proof. Lemma 1 implies that, at all times, the earliest deadline server has one and only one ready task in its list. As explained in [10], the resulting schedule can be regarded, from the viewpoint of the global scheduler, as a sequence of real-time jobs whose deadlines are equal to the deadlines of the servers (also referred as *chunks* in [20] and [4]). As the earliest deadline server never blocks, the computation times and the deadlines of the chunks generated by the server do not depend on the presence of shared resources. In [20], [10], it was proven that, in every interval of time, the bandwidth *demanded* by the chunks produced by server S_i never exceeds $\frac{Q_i}{P_i}$, regardless of the behavior of the served tasks. Since Lemma 1 states that each active server always has one nonblocked task in its list, the previous result is also valid for BWI. Hence, from the optimality of EDF and from $\sum_{i=1}^{n} \frac{Q_i}{P_i} \leq 1$, it follows that none of these chunks misses its deadline. □

Note that the previous theorem states that no *scheduling deadline* (that is, we recall, the server's deadline used to schedule the servers with EDF) will be missed, but it does

not say anything about a task's deadlines. Recall that the goal of a real-time scheduling algorithm is to meet the HRT deadlines. At this point, we need a way to relate the tasks' deadlines to the server deadlines (and, hence, to the server parameters) so that it is possible to provide guarantees to HRT tasks.

Definition 3. *Given a task τ_i, served by a server S_i with the BWI protocol, the* interference time *I_i is defined as the maximum time that all other tasks can execute inside server S_i for each job of τ_i.*

Theorem 4. *If an HRT task τ_i is served by a server S_i with the BWI protocol, with parameters $Q_i = C_i + I_i$ and $P_i = T_i$, where C_i is the WCET, T_i is the minimum interarrival time, and I_i is the maximum interference time for S_i, then task τ_i will meet all its deadlines, regardless of the behavior of the other noninteracting tasks in the system.*

Proof. According to Theorem 2, the CBS algorithm guarantees that each server S_i receives up to Q_i units of execution every P_i units of time. Hence, if each instance of τ_i consumes less than Q_i and instances are separated by P_i or more, server S_i never postpones its scheduling deadline. From Theorem 1, $f_{i,j} \leq \delta_i$.

Theorem 3 extends the result of Theorem 2 to BWI. However, when considering the BWI protocol, other tasks can execute inside server S_i, consuming its budget (and, hence, postponing the deadline of server S_i even if $C_i \leq Q_i$). In order to ensure that server S_i will never postpone its scheduling deadline, we have to consider the interference time from those tasks. If I_i is the maximum time that other tasks can execute inside S_i, it follows that τ_i can execute for $Q_i - I_i$ units of time before exhausting the server budget. Hence, the theorem follows. □

6.4.1 Considerations

When our system consists only of HRT tasks, BWI is not the best protocol to use. In fact, substituting Q_i and P_i into (1), we obtain:

$$\sum_{i=1}^{n} \frac{C_i + I_i}{T_i} \leq 1,$$

which may result in a lower utilization than (2) because all the interference times are summed together. Hence, if we are dealing with a hard real-time system, it is better to use other scheduling strategies like the PCP [5] or the SRP [17], [2].

The BWI protocol is more suitable for dynamic real-time systems, where hard, soft, and non-real-time tasks can coexist and it is impossible to perform an offline analysis for the entire system. Of course, this comes at the cost of a lower utilization for HRT tasks.

7 Interference Time Computation

In the general case, an exact computation of I_i is a complex problem. In this section, we restrict our attention to the computation of the interference time for HRT tasks. At first glance, the problem may seem similar to the problem of computing the blocking times B_i for the PIP. However, computing the interference time is much more difficult because we have to consider the interference of the SRT tasks. Their unpredictable execution times may cause the associated servers to exhaust their budgets and postpone their deadlines.

In many cases, it is desirable to guarantee an HRT task τ_i even if it interacts with SRT tasks. In fact, sometimes it is possible to know indirectly the worst-case execution time of the critical sections of an SRT task. For example, consider an HRT task and an SRT task that access the same resource by using common library functions. If the critical sections are implemented as library functions with bounded execution time, then we can still determine the amount of time that a soft task can steal from the server's budget of an HRT task. Indeed, this is a very common case in a real operating system.

Therefore, we will now consider the problem of computing I_i for a server S_i that is the default server of an HRT task. We start by providing an important definition that simplifies the discussion.

Definition 4. *Let S_i be a server that never postpones its deadline (i.e., S_i's budget is never exhausted while there is a job that has not yet finished). We call S_i an* HRT server. *If the server deadline of S_i can be postponed (i.e., a time t exists in which $q_i = 0$ and the served job has not yet finished), we call S_i an* SRT server.

The distinction between HRT and SRT servers depends only on the kind of tasks they serve. Both HRT and SRT servers follow the same rules and have the same characteristics. However, it may be impossible to know the WCET of an SRT task, so the corresponding default SRT server can *decrease its priority* while executing. The presence of SRT servers that interact with HRT servers complicates the computation of the interference time.

The following examples show how one or more SRT tasks can contribute (directly or indirectly) to the interference time of an HRT task.

Example 3. Consider an HRT task τ_i, served by server S_i and an SRT task τ_j, served by a server S_j with period $P_j < P_i$. We do not know the WCET of task τ_j. Therefore, we assign the budget of S_j according to some rule of thumb. Server S_j is an SRT server as it may postpone its deadline. If τ_j executes less than its server budget and the server deadline is not postponed, S_i cannot preempt S_j. If, instead, τ_j executes for more than its server budget, the server's deadline is postponed. The corresponding situation is shown in Fig. 3a. S_j can be preempted by S_i while inside a critical section, and block τ_i, contributing to its interference time I_i.

Example 4. Consider three tasks, τ_i, τ_j, and τ_k, served by servers S_i, S_j, and S_k, respectively, with $P_j < P_i < P_k$. Servers S_i and S_k are HRT servers, while S_j is an SRT server. All tasks access resource R. Task τ_i accesses resource R twice with two different critical sections. One possible blocking situation is shown in Fig. 3b. The first time, τ_i can be blocked by task τ_k on the first critical section. Then, it can be preempted by task τ_j, which first locks R, and then, before releasing the resource, depletes the server budget and postpones its deadline. Thus,

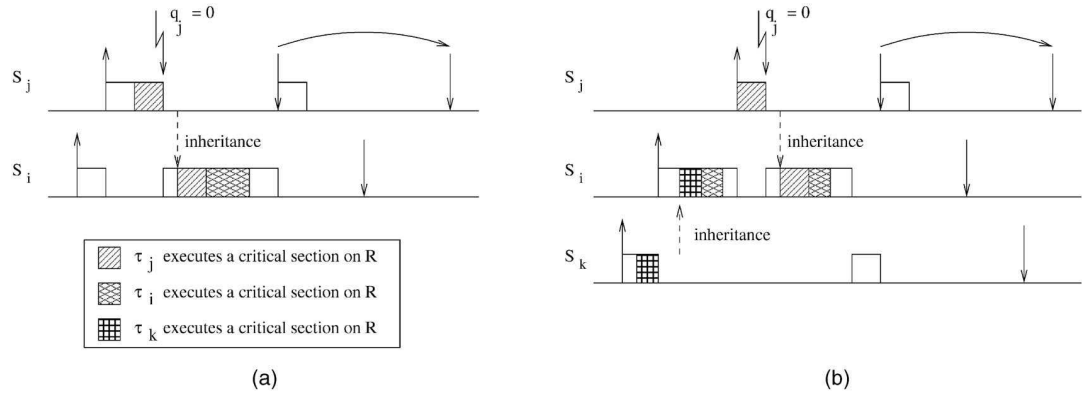


Fig. 3. Example of blocking situations with SRT tasks: (a) An SRT task with a short period blocks an HRT task with a long period. (b) An HRT task is blocked twice on resource R.

when τ_i executes, it can be blocked again on the second critical section on R. Note that both τ_j and τ_k belong to Γ_i.

Example 5. As the last example, we show one case in which, even if all tasks in Γ_i are HRT tasks, it may happen that τ_j interferes with S_i with two different critical sections. Consider three tasks, τ_i, τ_j, and τ_k. Task τ_i accesses only resource R_2 with two critical sections. Task τ_j accesses two resources R_1 and R_2 and R_2 is accessed twice with two critical sections, both nested inside the critical section on R_1. Task τ_k accesses only R_1 with one critical section. The only blocking chain starting from task τ_i is $H_i = (\tau_i, R_2, \tau_j)$. Hence, $\Gamma_i = \{\tau_j\}$. Note that task τ_k cannot interfere with task τ_i.

Tasks τ_i, τ_j, and τ_k are assigned servers S_i, S_j, and S_k, respectively, with $P_k < P_i < P_j$. Tasks τ_i and τ_j are both HRT tasks and we know their WCETs and periods. Task τ_k is an SRT task and we do not know its WCET. Finally, we assume knowing the duration of all critical sections (for example, because resources are accessed through shared libraries that we are able to analyze).

We assign budgets and periods of servers S_i and S_j so that they are HRT servers (to compute their interference time, we use the algorithm described in Fig. 5, which will be presented later). We assign the budget of server S_k according to some rule of thumb. We do not know if S_k will exhaust its budget while executing. Therefore, we consider server S_k as an SRT server.

One possible blocking situation is shown in Fig. 4. Task τ_j locks resource R_1 and then resource R_2. At time t_1, it is preempted by task τ_i that tries to lock resource R_2 and it is blocked. As a consequence, task τ_j inherits server S_i and interferes with it for the duration of the first critical section on R_2. When τ_j releases R_2, it returns inside its server S_j and τ_i executes completing its critical section on R_2. Then, server S_k is activated and τ_k starts executing and tries to lock resource R_1. Since R_1 is still locked by τ_j, τ_k is blocked and τ_j inherits server S_k. While τ_j executes inside S_k, it again locks resource R_2. Before releasing R_2, server S_k exhausts its budget and postpones its deadline. Now, the earliest deadline server is S_i that continues to execute and tries to again lock R_2 at time t_2. As a consequence, τ_j inherits S_i and interferes with it for the second time.

From the examples shown above, it is clear that there are many possible situations in which a task can interfere with a server. In the next section, we formally present a set of lemmas that identify the conditions under which a task can interfere with an HRT server.

7.1 Conditions for Interference

We start by defining the set of servers that can be inherited by a task.

Definition 5. *Set Ψ_j is the set of all servers that can be "inherited" by task τ_j, S_j included:*

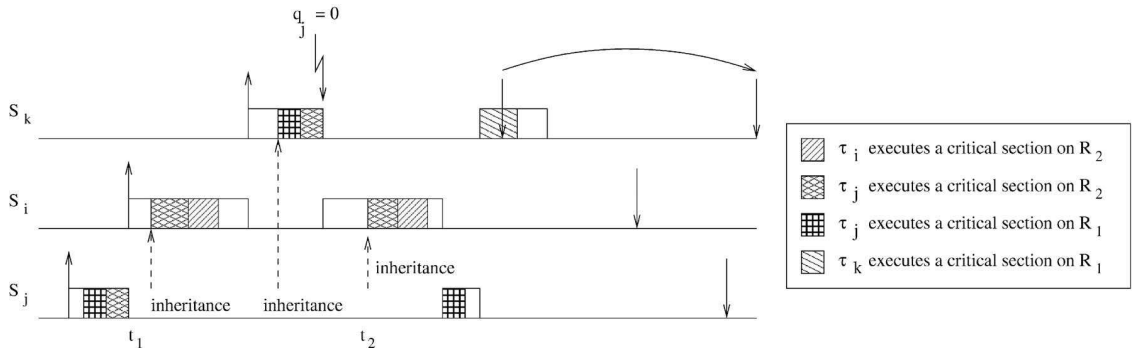


Fig. 4. Example of blocking situation: Task τ_j can interfere twice with S_i even if S_i and S_j are both HRT servers.

```
1: int interference(int k, set T, set R)
2: {
3:   int ret = 0;
4:   if (k > cs_i) return 0;
5:   ret = interference(k+1, T, R);
6:   foreach (H_i ∈ CS_i(k)) {
7:     if (Γ̄(H_i) ⊆ T and R̄(H_i) ⊆ R) {
8:       T' = T \ Γ̄(H_i);
9:       R' = R \ R̄(H_i);
10:      ret = max(ret, ξ(H_i) + interference(k+1, T', R'));
11:    }
12:  }
13:  return ret;
14:}
```

Fig. 5. Algorithm for computing the interference time for server S_i.

$$\Psi_j = \{S_i | \exists H_i, \ \tau_j \in H_i\} \cup \{S_j\}.$$

A task τ_j can only inherit tasks in Ψ_j, hence $\forall t \ \ S_{e(j,t)} \in \Psi_j$.

Definition 6. *Set $\Psi_j^{SRT}(i)$ is the set of SRT servers that can be "inherited" by task τ_j and interfere with server S_i:*

$$\Psi_j^{SRT}(i) = \{S_k | S_k \text{ is an SRT server} \wedge \exists H_k = (\tau_k, \ldots, \tau_j, \ldots, \tau_i)\}.$$

If S_j is an SRT server, it is also included in $\Psi_j^{SRT}(i)$.

Consider again Example 5. There is one chain from τ_k to τ_i: $H_k = (\tau_k, R_1, \tau_j, R_2, \tau_i)$. Therefore, $S_k \in \Psi_j^{SRT}(i)$. Set $\Psi_j^{SRT}(i)$ is important in our analysis because it identifies whether a task τ_j can inherit an SRT server before interfering with the server S_i under analysis. In Example 5, task τ_j can inherit the SRT server S_k, which may later postpone its deadline.

Now, we prove some important properties of the BWI protocol.

Lemma 2. *If server S_i is HRT, then $\forall t : \delta_{e(i,t)} \leq \delta_i$.*

Proof. When τ_i inherits a server S_j, this server must have a scheduling deadline shorter than δ_i. Recall that, by definition, $e(i,t)$ is the index of the server with the shortest scheduling deadline among all servers inherited by τ_i at time t. Hence, $\delta_{e(i,t)} = \delta_j < \delta_i$. If S_j postpones its deadline before the time in which τ_i releases the resource, τ_i continues to execute inside the server with the shortest deadline among the inherited servers. Since S_i never postpones its deadline, the lemma is proven. □

Lemma 3. *Given a task τ_i, only tasks in Γ_i can be added to server S_i and contribute to I_i.*

Proof. It follows directly from Rule D and from the definition of Γ_i. □

Lemma 4. *Let S_i be an HRT server. Task τ_j with default server S_j cannot interfere with server S_i if:*

$$P_j \leq P_i \quad \wedge \quad \forall S_k \in \Psi_j^{SRT}(i) : P_k > P_i.$$

Proof. By contradiction. For τ_j to interfere with S_i, it must happen that, at a certain time t_1, τ_j locks a resource R; it is then preempted by server S_i at time t_2, which blocks on some resource; τ_j inherits S_i as a consequence of this blocking. Therefore, τ_j must start executing inside its default server before S_i arrives and executes in a server $S_{e(j,t_2)}$ with deadline $\delta_{e(j,t_2)} > \delta_i$ when it is preempted. By hypothesis, $P_j \leq P_i \Rightarrow \delta_j < \delta_i$. Hence, τ_j executes in a server $S_{e(j,t_1)}$ with $\delta_{e(j,t_1)} > \delta_i$. However, it follows from the hypothesis that server S_j never postpones its deadline ($S_j \notin \Psi_j^{SRT}(i)$) and, from Lemma 2, $\delta_{e(j,t_1)} \leq \delta_j < \delta_i$. This is a contradiction, hence the lemma is proven. □

Now, we are ready to define more precisely which tasks can interfere with our server S_i.

Definition 7. *A* proper blocking chain *H_i is a blocking chain that contains only tasks that can interfere with S_i:*

$$\forall \tau_j \in H_i : P_j > P_i \quad \vee \quad \exists S_k \in \Psi_j^{SRT}(i) : P_k \leq P_i.$$

Later, we will present an algorithm for computing the interference time for server S_i which explores all proper blocking chains starting from τ_i. However, in some cases, we have to consider multiple interference times from the same task and multiple interference times on the same resources. The following lemmas restrict the number of possible interference situations.

Lemma 5. *Let S_i be an HRT server and τ_j a task belonging to a proper blocking chain H_i. If the following condition holds:*

$$P_j > P_i \quad \wedge \quad \forall S_k \in \Psi_j^{SRT}(i) : P_k \geq P_i,$$

then τ_j can interfere with server S_i for at most the worst-case execution of one critical section for each job.

Proof. Suppose that τ_j interferes with S_i in two different intervals. The first time it interferes in interval $[t_1, t_2)$, the second time in interval $[t_3, t_4)$. Therefore, at time t_2, $\delta_{e(j,t_2)} > \delta_i$. If τ_j does not lock any resource in $[t_2, t_3)$, then, at time t_3, server S_i blocks on some resource R that was locked by τ_j before t_1 and that it has not yet released. Therefore, τ_j interferes with S_i for the duration of the critical section on R, which includes the duration of the first critical section ($\xi(R) \geq (t_4 - t_3) + (t_2 - t_1)$) and the lemma is proven.

Now, suppose that τ_j executes in interval $[t_2, t_3)$ and locks another resource R_1. It follows that it inherits a server S_k that preempts S_i with $\delta_k < \delta_i$. Hence, $P_k < P_i$. From the hypothesis, S_k is an HRT server and δ_k is not

postponed before τ_j releases resource R_1. Hence, τ_j cannot inherit S_i while it is inside S_k and we fall back to the previous case. □

Lemma 6. *Let S_i be an HRT server and R a resource. If the following condition holds:*

$$\forall H_i^h, H_i^h = (\ldots, R, \tau_j, \ldots), \quad \forall S_k \in \Psi_j^{SRT}(i) : P_k \geq P_i,$$

then at most one critical section on R can contribute to interference time I_i.

Proof. The proof of this lemma is very similar to the proof of Lemma 5. By contradiction. Suppose that two critical sections on the same resource R contribute to I_i. The first time, task τ_p inherits server S_i at time t_1 while it is holding the lock on R. The second time, task τ_j inherits server S_i at time $t_2 > t_1$ while it is holding the lock on R. It follows that:

- The lock on R was released between t_1 and t_2;
- τ_j arrives before t_2 and executes on a server $S_{e(j)}$ with $\delta_{e(j)} < \delta_i$;
- τ_j acquires the lock on R at $t_a < t_2$;
- just before t_2, $\delta_i < \delta_{e(j),t_2}$.

Hence, at time t_a, τ_j is executing in an inherited server $S_k \in \Psi_j^{SRT}(i)$ that postpones its deadline before τ_j releases the lock on R. S_k must arrive after S_i with deadline $\delta_k < \delta_i$ and later postpone its deadline. This contradicts the hypothesis that $\forall S_k \in \Psi_j^{SRT}(i) : P_k \geq P_i$. Hence, the lemma is proven. □

The previous lemmas restrict the number of combinations that we must analyze when computing the interference time. In particular, Lemma 5 identifies the conditions under which a task can interfere with server S_i for at most one critical section; Lemma 6 identifies the conditions under which a certain resource can interfere with server S_i at most one time.

Now, we need to quantify the interference time due to each blocking chain.

Lemma 7. *The worst-case interference time for server S_i due to a proper blocking chain $H_i = (\tau_1, R_1, \ldots, R_{z-1}, \tau_z)$ is:*

$$\xi(H_i) = \sum_{k=2}^{z} \xi_k(R_{k-1}). \tag{4}$$

Proof. It simply follows from the definition of proper blocking chain. □

Given a proper blocking chain, we need to distinguish the tasks that can interfere with S_i for at most the duration of one critical section (i.e., that verify the hypothesis of Lemma 5) from the task that can interfere with S_i multiple times.

Definition 8. *Given a proper blocking chain H_i^h, let $\overline{\Gamma}(H_i^h)$ be the set of tasks in H_i^h that verify the hypothesis of Lemma 5.*

$$\overline{\Gamma}(H_i^h) = \Big\{\tau_j | \tau_j \in H_i^h \wedge P_j > P_i \wedge (\forall S_k \in \Psi_j^{SRT}(i) : P_k \geq P_i)\Big\}.$$

We do the same thing for the resources.

Definition 9. *Given a proper blocking chain H_i^h, let $\overline{R}(H_i^h)$ be the set of resources in H_i^h that verify the hypothesis of Lemma 6.*

$$\overline{R}(H_i^h) = \Big\{R_j | R_j \in H_i^h \wedge P_{j+1} > P_i \wedge (\forall S_k \in \Psi_{j+1}^{S} RT(i) : P_k \geq P_i)\Big\}.$$

7.2 Algorithm for Computing the Interference Time

We are now ready to present the pseudocode for Algorithm `interference()`, shown in Fig. 5. Let cs_i denote the number of critical sections for task τ_i and $CS_i(k)$ denote the set of proper blocking chains starting from the kth critical. More formally, $CS_i(k)$ is the set of proper blocking chains of the form $H_i^h = (\tau_i, R, \ldots)$, where R is the resource accessed in the kth critical section of τ_i.

Function `interference`$(\mathtt{k}, \mathcal{T}, \mathcal{R})$ is first called with $k = 1$, $\mathcal{T} = \Gamma_i$, and with $\mathcal{R}$ equal to the set of all possible resources. At line 5, we consider the case in which τ_i is not blocked on the kth critical section. In this case, the function is recursively called for the $(k+1)$th critical section.

At lines 6-12, we consider the case in which τ_i is blocked on the kth critical section. For each proper blocking chain H_i in $CS_i(k)$, the algorithm checks if it is a legal blocking chain, i.e., the resources in $\overline{R}(H_i^k)$ and the tasks in $\overline{\Gamma}(H_i^k)$ have not yet been considered in the interference time computation. If so, function `interference()` is recursively called with $k' = k + 1$, $\mathcal{T}' = \mathcal{T} \setminus \overline{\Gamma}(H_i)$, and $\mathcal{R}' = \mathcal{R} \setminus \overline{R}(H_i)$ (lines 8-10). Otherwise, it selects another chain from $CS_i(k)$. The recursion stops when $k > cs_i$ (line 4).

The algorithm has exponential complexity since it explores all possible interference situations for server S_i. We conjecture that the problem of finding the interference time in the general case is NP-Hard. However, we leave the proof of this claim as future work.

7.3 Interference Time Computation for SRT Tasks

It is very difficult to compute the interference time for an SRT task. The problem is that an SRT task can deplete the capacity of its server, postponing the server deadline, which causes the server's priority to decrease.

As a consequence, an SRT task can potentially be blocked many times by every other task and several times on each resource. The worst-case interference time can be very high, whereas the average interference time will probably be much lower than the worst case. Moreover, since an SRT task can tolerate occasional deadline misses, there is no advantage to selecting the server's budget based on the worst-case computation time and worst-case interference time.

In order to assign the server parameters and then adjust them for minimizing the number of deadline misses, it is better to dynamically adjust the budget by using an online feedback mechanism like the one proposed by Abeni and Buttazzo [23] or the elastic model proposed by Buttazzo et al. [24].

8 Conclusions and Future Work

In this paper, we presented the Bandwidth Inheritance protocol, a novel scheduling discipline that allows the sharing of resources between real-time tasks in dynamic real-time systems. Coherently with the Resource Reservation approach, BWI provides some temporal isolation properties without requiring any a priori knowledge about the structure and the temporal behavior of the tasks.

In addition, a schedulability analysis for HRT tasks has been developed and presented in this paper. This analysis is based on formal properties of the BWI protocol that are presented and proven in the paper.

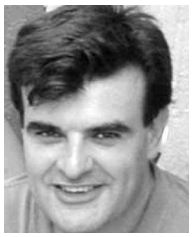
As future work, we are currently analyzing new strategies for coping with SRT tasks. Roughly speaking, an SRT task that *borrows* interference time from another SRT task should *give it back* after some time. The idea is that every SRT task should be given a fair share of the processor over long intervals. Moreover, we are planning to combine feedback scheduling techniques [25], [24], [23] with BWI in order to adjust the parameters of the servers and minimize the number of missed SRT deadlines.

ACKNOWLEDGMENTS

This work was carried out while G. Lamastra and L. Abeni were PhD students at the Scuola Superiore Sant'Anna.

REFERENCES

[1] T. Ghazalie and T. Baker, "Aperiodic Servers in a Deadline Scheduling Environment," *J. Real-Time Systems,* vol. 9, 1995.
[2] M. Caccamo and L. Sha, "Aperiodic Servers with Resource Constraints," *Proc. IEEE Real-Time Systems Symp.,* Dec. 2001.
[3] D. de Niz, L. Abeni, S. Saewong, and R. Rajkumar, "Resource Sharing in Reservation-Based Systems," *Proc. IEEE Real-Time Systems Symp.,* Dec. 2001.
[4] L. Abeni and G. Buttazzo, "Integrating Multimedia Applications in Hard Real-Time Systems," *Proc. 19th IEEE Real-Time Systems Symp.,* Dec. 1998.
[5] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. Computers,* vol. 39, no. 9, Sept. 1990.
[6] Z. Deng and J.W.S. Liu, "Scheduling Real-Time Applications in Open Environment," *Proc. IEEE Real-Time Systems Symp.,* Dec. 1997.
[7] M. Spuri and G.C. Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems," *J. Real-Time Systems,* vol. 10, no. 2, 1996.
[8] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic Task Scheduling for Hard-Real-Time Systems," *J. Real-Time Systems,* vol. 1, July 1989.
[9] J.K. Strosnider, J.P. Lehoczky, and L. Sha, "The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard-Real-Time Environments," *IEEE Trans. Computers,* vol. 4, no. 1, Jan. 1995.
[10] G. Lipari, "Resource Reservation in Real-Time Systems," PhD dissertation, Scuola Superiore S. Anna, 2000.
[11] C.W. Mercer, S. Savage, and H. Tokuda, "Processor Capacity Reserves for Multimedia Operating Systems," *Proc. IEEE Int'l Conf. Multimedia Computing and Systems,* May 1994.
[12] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems," *Proc. Fourth Real-Time Computing Systems and Applications Workshop,* Nov. 1997.
[13] R. Rajkumar, L. Abeni, D.D. Niz, S. Gosh, A. Miyoshi, and S. Saewong, "Recent Developments with Linux/RK," *Proc. Real Time Linux Workshop,* Dec. 2000.
[14] H. Chu and K. Nahrstedt, "CPU Service Classes for Multimedia Applications," *Proc. IEEE Int'l Conf. Mutimedia Computing and Systems,* June 1999.
[15] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM,* vol. 20, no. 1, 1973.
[16] M. Chen and K. Lin, "Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems," *J. Real-Time Systems,* vol. 2, 1990.
[17] T.P. Baker, "A Stack-Based Allocation Policy for Realtime Processes," *Proc. IEEE Real-Time Systems Symp.,* Dec. 1990.
[18] K. Jeffay, "Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems," *Proc. 13th IEEE Real-Time Systems Symp.,* pp. 89-99, Dec. 1992.
[19] T.-W. Kuo and C.-H. Li, "Fixed-Priority-Driven Open Environment for Real-Time Applications," *Proc. IEEE Real Systems Symp.,* Dec. 1999.
[20] L. Abeni, "Server Mechanisms for Multimedia Applications," Technical Report RETIS TR98-01, Scuola Superiore S. Anna, 1998.
[21] G. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Kluwer Academic, 1997.
[22] J.A. Stankovic, M. Spuri, K. Ramamritham, and G.C. Buttazzo, *Deadline Scheduling for Real-Time Systems—EDF and Related Algorithms.* Kluwer Academic, 1998.
[23] L. Abeni and G. Buttazzo, "Adaptive Bandwidth Reservation for Multimedia Computing," *Proc. IEEE Real Time Computing Systems and Applications,* Dec. 1999.
[24] G. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni, "Elastic Scheduling for Flexible Workload Management," *IEEE Trans. Computers,* vol. 51, no. 3, pp. 289-302 Mar. 2002.
[25] C. Lu, J.A. Stankovic, G. Tao, and S.H. Son, "Design and Evaluation of a Feedback Control EDF Scheduling Algorithm," *Proc. IEEE Real Time Systems Symp.,* Dec. 1999.



Giuseppe Lipari graduated in computer engineering from the University of Pisa in 1996 and received the PhD degree in computer engineering from the Scuola Superiore Sant'Anna in 2000. During 1999, he was a visiting student at the University of North Carolina at Chapel Hill, collaborating with Professor S.K. Baruah and Professor K. Jeffay on real-time scheduling. Currently, he is an assistant professor of operating systems with the Scuola Superiore Sant'Anna. His main research activities are in real-time scheduling theory and its application to real-time operating systems, soft real-time systems for multimedia applications, and component-based real-time systems. He is a member of the IEEE.

Gerardo Lamastra graduated with the degree in computer engineering from the University of Pisa, Italy, in 1996. In 2000, he received the PhD degree in computer engineering and real-time systems from the Scuola Superiore Sant'Anna, Pisa. In 2000, he joined Telecom Italia Lab, where he started working in the Internet Security Group (be-secure). In 2002, he received the SANS certification for System Forensics, Investigation and Response. His interests include: operating systems architecture and design, real-time systems, intrusion detection, and forensic analysis. He is a member of the IEEE.

Luca Abeni graduated in computer engineering from the University of Pisa in 1998 and received the PhD degree in computer engineering from the Scuola Superiore Sant'Anna of Pisa, Italy, in 2002. During 2000, he was a visiting student at Carnegie Mellon University, working with Professor Ragunathan Rajkumar on resource reservation algorithms for real-time kernels. During 2002, he was a visiting student at the Oregon Graduate Institute, Portland, working on evaluating and enhancing the real-time performance of the Linux kernel. He currently works at Broadsat S.r.l., developing multimedia streaming solutions. He is a member of the IEEE.