# A Bandwidth Inheritance Algorithm for Real-Time Task Synchronization in Open Systems

Gerardo Lamastra
Telecom Italia Lab
Gerardo.Lamastra@tilab.com

Giuseppe Lipari, Luca Abeni
Scuola Superiore S.Anna
{lipari,luca}@sssup.it

## Abstract

*In this paper, we present algorithm BandWidth Inheritance (BWI), a new scheduling strategy that extends the bandwidth reservation approach to systems where tasks can interact through shared resources. The proposed algorithm provides temporal isolation between independent groups of tasks, and enables a schedulability analysis for guaranteeing the performance of real-time tasks. After showing that BWI is the natural extension of the well-known Priority Inheritance Protocol to dynamic reservation systems, a formal analysis of the algorithm is presented, and simple guarantee tests for hard real-time tasks are proposed.*

## 1. Introduction

An Open System is a general purpose computer system in which it is not possible to know a-priori how many applications need to be executed and the characteristics thereof. Typically, in an Open System applications with different levels of Quality of Service may coexists: hard real-time, multi-media and interactive non-real-time applications.

The Open System concept was originally presented in [9], where a dedicated aperiodic server is used to serve each different application, and a custom scheduler can be associated to each application in a hierarchical way. However, the algorithms proposed in [9] require a-priori knowledge of the tasks' execution times even for soft real-time tasks and this can be restrictive for a real Open System.

A better approach is to provide isolation between applications, such that an application is protected from the misbehaviors of the other applications. This property is called *Bandwidth Isolation* (BIP) [2, 11]: the net effect is that each application executes as it were on a slower dedicated processor. Based on this approach, the Resource Reservation framework [15, 18] has been proven very effective in the joint scheduling of Hard Real Time (HRT) and Soft Real Time (SRT) applications in Open Systems. In this framework, when an application arrives in the system requiring a certain level of QoS, an *admission test* is run: if, given the actual system load, the required level of QoS can be guaranteed, the application 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 Resource Reservation concepts were first introduced in RT-Mach [15], and evolved in the Resource Kernel [18]. A Resource Kernel (RK) permits to reserve hardware resources to applications, guaranteeing that each application can use a resource for a time $C$ each period $T$. The RK technology is particularly interesting, since it seamlessly provides QoS even to non real-time legacy applications, and has been recently ported to Linux [17].

Resource Reservation techniques, originally based on a fixed priorities assignment, have also been applied to dynamic priority schemes [7, 2]. One of the advantages of using dynamic priorities (and in particular an EDF scheduler), is to improve the usage of the available resources. The Constant Bandwidth Server (CBS) [2], uses a deadline postponing mechanism to efficiently provide bandwidth isolation. In particular, the CBS algorithm provides real-time guarantees to hard real-time tasks and probabilistic guarantees to soft real-time tasks. Recently, some reclaiming mechanisms [13, 5] have been added to the CBS.

However, all the algorithms presented so far support only independent tasks: a task is not allowed to block or suspend itself, otherwise certain properties, such as the BIP, cannot be guaranteed. This is a major limitation for their implementation in a real operating system: in the real world, many real-time applications communicate by means of shared memory and access mutually exclusive resource.

In the real-time system literature a great amount of work has been addressed to the problem of providing guarantees to hard real-time tasks accessing mutually exclusive resources. In fact, if classical mutex semaphores are used, a particular problem arises, usually referred as the *priority inversion problem.* This problem was first described by Sha, Rajkumar and Lehoczky in [19]: if a higher priority task is blocked on a semaphore by a lower priority task, and another medium priority task arrives, the latter can preempt the lower priority task causing an unbounded blocking delay to the higher priority task. In the same paper, the authors proposed two algorithms to overcome this problem, the *Priority Inheritance* and the *Priority Ceiling* protocols. Almost at the same time, another protocol similar to Priority Ceiling, the Stack Resource Policy, was presented by Baker in [3].

In [10], it is shown how to combine the SRP algorithm 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 in [6]. However, this kind of solutions require an a-priori knowledge of the maximum resource usage time for each task, and in an open system it is impossible to have such an a-priori knowledge.

For this reason, resource sharing protocols such as PI, PC, or SRP cannot be directly used in an Open System. For example, the CBS algorithm assumes a very simple task model (the Liu and Layland task model [14]), where no task can suspend itself or block on a critical section. If tasks are allowed to suspend, inconsistencies can arise in the CBS algorithm. In [8], similar problems have been investigated in the context of a reservation based system such as the RK; in particular, the inheritance and ceiling techniques have been modified to correctly work in a reservation system, and one of the proposed solutions (namely, the Reserve Inheritance approach) results to be similar to the one presented in this paper.

In this paper a new algorithm, the BandWidth Inheritance (BWI) algorithm, is introduced. It tries to combine the advantages of the CBS algorithm (bandwidth isolation and real-time guarantees to hard real-time tasks) and the Priority Inheritance protocol, like bounded blocking delay. The BWI algorithm does not require any a-priori information on the task, hence it is particularly suited for open systems.

## 2. System Model

A *real-time task* $\tau_i$ is a stream of *jobs*, or *instances*, $J_{i,j}$; 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 $d_{i,j} = a_{i,j} + D_i$.

Real-time tasks can be *hard* or *soft*: a task $\tau_i$ is said to be a hard real-time task (HRT) if all its jobs must complete within their deadline otherwise a critical failure may occur in the system. Every HRT task $\tau_i$ 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 task (SRT) if some occasional deadline miss is tolerated by the system. The ratio of deadline misses within a given time interval can be used as a measure of the Quality of Service experienced by SRT tasks.

A task may access shared resources in mutual exclusion using mutex semaphores. If task $\tau_i$ accesses resource $R_k$, we say that $\tau_i$ uses $R_k$. We denote the worst case execution time of the longest critical section of task $\tau_i$ on resource $R_k$ as $\xi_{i,k}$.

In this paper, we propose to schedule each task through a *dedicated server.* A server is a "scheduling entity", characterized by a scheduling priority, that is inserted in the system ready queue. Each task is associated to a server, and when the server is selected by the scheduler, the associated task is dispatched. For the sake of simplicity, in the remaining of the paper we will write "server $S_i$ is blocked", meaning that the task served by $S_i$ is blocked, and "$S_i$ is executing" meaning that the task served by $S_i$ is executing.

Each server is characterized by two parameters $Q_i$ and $P_i$, with the meaning that each task is allowed to execute $Q_i$ every $P_i$ units of time. The fraction $U_i = \frac{Q_i}{P_i}$ represents the *share* of the processor reserved to server $S_i$. Our attention is restricted to systems where all servers execute on a single shared processor and the sum of the processor shares of all the servers is no more than one: $\sum_i Q_i/P_i \leq 1$.

**Requirements.** In order to be successfully used in an open system, the scheduling discipline must not require any a-priori information about the tasks' execution patterns. In particular:

- The arrival times of the jobs (the $a_{i,j}$'s) are not known *a priori*, but are only revealed on line during system execution.
- 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 on line when the task tries to lock a resource. Nor do we require any *a priori* upper bound on the duration $\xi_{i,j}$ of a critical section.

Note that these requirements do not exclude the possibility to perform a schedulability analysis on a subset of HRT tasks. If the application designer performs an accurate timing analysis on each HRT task and is able to bound the execution times and the duration of the critical sections, then it is possible to find an assignment of parameters to the servers which guarantees the schedulability of the HRT tasks. The isolation property ensures that, even if SRT tasks are present in the system together with HRT tasks, the schedulability of the HRT tasks will not be jeopardized.

## 3. 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 [2].

### 3.1. The Server Mechanism

A CBS $S_i$ 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 reserved to $S_i$. Dynamically, each server maintains a pair $(q_i, \delta_i)$ where $q_i$ is the current budget and keeps track of the consumed bandwidth, and $\delta_i$ is the scheduling deadline. Initially, $q_i$ is set to the maximum budget $Q_i$ and $\delta_i$ is set to 0. A server is *active* if the corresponding server task has a pending instance.

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 $\delta_i$ is selected and the corresponding task is dispatched to execute.

The CBS algorithm updates its variables as follows:

**Rule A:** When a job $J_{i,j}$ of the task $\tau_i$ arrives, the server checks whether the current pair $(q_i, \delta_i)$ can be used without violating the bandwidth isolation. In the case this is not possible, a new pair $(q_i, \delta_i)$ is computed, with $q_i \leftarrow Q_i$ and $\delta_i \leftarrow a_{i,j} + P_i$, otherwise $q_i$ and $\delta_i$ are left unchanged.

**Rule B:** when server $S_i$ executes for $\Delta t$ 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 $q_i \geq 0$. When the budget is exhausted, a new pair $(q_i, \delta_i)$ 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 has to be reordered.

All the tasks in the system are considered *independent* and no task is allowed to suspend itself waiting for a shared resource or a synchronization event.

The CBS algorithm has two important properties: *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 perform a schedulability analysis on each HRT task independently from the others. For proofs of the following theorems, please refer to [1] and [11].

**Theorem 1 (Bandwidth Isolation Property)** *Given a system comprised of $n$ CBS servers, with $\sum_{i=1}^{n} U_i \leq 1$, no server misses its scheduling deadline regardless of the behavior of the tasks: in other words, it is guaranteed that at every instant $t$, for every active server $S_i$, its deadline $\delta_i$ is greater than or equal to the current time $t$.*

**Theorem 2 (Hard Schedulability Property)** *If a HRT task $\tau_i$ is served by a CBS $S_i = (Q_i, P_i)$, with maximum budget $Q_i \geq C_i$ and period $P_i \leq T_i$, then $S_i$ will never postpone its deadline and $\forall j, \quad f_{i,j} \leq \delta_i \leq d_{i,j}$.*

### 3.2. Sharing resources with CBS

Using the CBS algorithm, if tasks access critical section through classical semaphores, problems can arise. As an example, suppose that server $S_i$ is blocked on a shared resource by server $S_j$, which has a greater deadline.

- The blocked server $S_i$ must wait for the low priority server $S_j$ holding the resource; in the meanwhile, many medium priority server can execute, further delaying $S_i$. This is a form of *priority inversion*, and can be partially solved by introducing the Priority Inheritance Protocol (PIP) among the servers. The PIP is briefly presented in Section 4.

- In the admission test, blocking times are not accounted for: then, it is possible that some server $S_h$, which does not access any critical section, misses its deadline even if its served task is requiring exactly what it has declared. This is a failure of the bandwidth isolation property: in fact, $\tau_h$ is independent from $\tau_i$ and $\tau_j$.

- in order to provide a guarantees fot HRT tasks, we must account for the blocking time experienced by a server. There are different ways to account for this blocking time: in Section 4.1 we will analyze the different possibilities.

A first trivial solution, which maintains the CBS properties, is to consider a job that resumes from a blocked status as a new instance of the task. Therefore, when a job is "un–blocked", the corresponding server is assigned a new pair budget/deadline, according to the CBS rule.

This method is simple and effective for scheduling SRT tasks and guarantees the bandwidth isolation property among servers. However, it is not suitable for HRT tasks. In fact, it is difficult to bound the completion time of a task because we cannot know a-priori when the task will be un–blocked and which scheduling deadline will be assigned to its server.

## 4. The Priority Inheritance Protocol

The Priority Inheritance Protocol (PIP) was first presented in [19] to solve the priority inversion problem. It is based on the following rules:

- when an higher priority job $J_H$ wants to access a critical section and it is blocked on it, the low priority job $J_L$ which holds the lock on the resource *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.

Notice that, if more than one higher priority task is blocked on the critical section, it is possible for $J_L$ to inherit more than one priority (*multiple inheritance*).

Even though the PIP was thought in the context of fixed priority scheduling, it can be applied also in the context of dynamic priority scheduling, and the same basic properties hold:

- *A job $J_H$ can be blocked by a lower priority job $J_L$ for at most the duration 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 which 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 task are ordered by non decreasing periods ($T_i < T_j \Rightarrow i < j$). The schedulability condition is the following:

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

where $C_j$ and $T_j$ are respectively the worst case execution time and the period of task $\tau_j$ and $B_i$ is the worst case blocking time experienced by task $\tau_i$. There are two kinds of blocking times that contribute to $B_i$:

**direct blocking** – when a job $J_H$ tries to access a resource $R$ which is already locked by a lower priority job.

**push–trough blocking** – when a job $J$ is blocked because a lower priority job $J_L$ has inherited a priority higher than $J$.

The Priority Inheritance Protocol reduces the blocking delay of high priority jobs but has several limitations: first, it does not prevent deadlock; second, it causes the problem of *chained blocking* where a job can be blocked once for each critical section it tries to access. Thus, even if the blocking time is reduced, it can still be quite high. To overcome these limitations, in [19] Sha, Lehozcy and Rajkumar propose the Priority Ceiling Protocol (PCP) which prevents deadlock and chained blocking.

However, in order to use ceiling protocols like PCP or SRP, the system must know in advance which resources a task will access. This can be done only at system start–up or before a task begins the execution of the first instance. For this reason, the PCP cannot directly be used in an Open System. In the remaining of the paper, we will use the basic Priority Inheritance protocol.

### 4.1. Using the Priority Inheritance Protocol with the CBS

The simple introduction of the PIP on top of the CBS mechanism raises two questions:

- how to account for the blocking time of a task?
- what happens if some task requires more than declared?

A first solution would be to account for this blocking time in the admission test:

$$\forall i \quad 1 \leq i \leq n \quad \sum_{j=1}^{i} \frac{Q_j}{P_j} + \frac{B_i}{P_i} \leq 1 \tag{2}$$

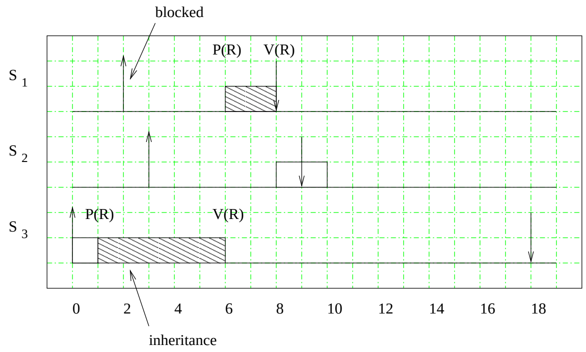


**Figure 1. A soft task blocks a hard task.**

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

Again, this solution is not suitable for an open system. In fact, in order to compute the maximum blocking time of each server, even SRT tasks must declare the maximum duration of the critical sections on each accessed resource, and this is in contrast with the goal of a scheduler which is independent from the actual requirements of the tasks.

In addition, if a SRT task fails to respect its requirements, and holds a critical section for longer than declared, some server will miss its deadline and, since we are using an EDF scheduler, we cannot know in advance which server is going to fail.

**Example.** To highlight this problem, consider the example shown in Figure 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 $\tau_1$ which access a resource $R$ for the entire duration of its jobs, that is 2 units of time. Server $S_2$ is assigned task $\tau_2$ which does not use any resource, and server $S_3$ is assigned task $\tau_3$ which has an execution time of 6 units of time and access resource $R$ for 5 units of time. However, suppose that $\tau_3$ is a SRT that declares to use the resource $R$ for only 2 units of time. The system computes a maximum blocking time $B_1 = B_2 = 2$ for servers $S_1$ and $S_2$. According to Equation (2), the system is schedulable, and all servers can be admitted.

In the configuration of arrival times shown in Figure 1, at time $t_2$ server $S_1$ arrives 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 duration of the critical section of task $\tau_3$ on resource $R$, server $S_2$ will miss its deadline, as it is shown in Figure 1. Notice that $S_2$ does not use any resource. $\Box$

Finally, there is yet another question concerning SRT tasks that need to be answered. In the original CBS formulation, when the budget of a server is exhausted, the server deadline is postponed and the server budget is immediately recharged. What happens if a server exhausts its budget while it is in a critical section and it has inherited the deadline of some other server? Which deadline has to be postponed, the server original deadline or the inherited deadline? How much should the capacity be recharged?

To answer all these questions, we shall follow a different approach. In the next section, we will fuse the PIP and the CBS in a single algorithm called BandWidth Inheritance (BWI) protocol. The basic idea is that when a lower priority server blocks a higher priority server, it inherits the pair $(q, \delta)$ of the blocked server.

## 5. The Bandwidth Inheritance Protocol

Before starting with the description of the Bandwidth Inheritance Protocol, we need to understand the meaning of “isolation” when considering interacting tasks. When tasks access shared resources, they cannot be considered completely independent anymore. What does it mean “isolation” in such scenario?

Consider again the example shown in Figure 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 let task $\tau_1$ miss its deadline instead of $\tau_2$: in fact, since $\tau_1$ and $\tau_3$ both access the same resource it is impossible to provide isolation among them. However, $\tau_2$ does not use any resource and can be considered independent from all the other tasks in the system.

### 5.1. Bandwidth isolation in the presence of shared resources

In order to formalize the concept of “interaction” between tasks, we define an equivalence relation $\models$ that divides the task set in partitions of dependent tasks. Two tasks belonging to two different partitions are independent, and can be isolated from one another. The equivalence relation is defined as follows. We say that $\tau_i$ *interacts* with $\tau_j$, and we write $\tau_i \models \tau_j$, in one of the following cases:

- $\tau_i$ and $\tau_j$ both access a resource $R$;
- a chain of tasks $\tau_1, \tau_2, \ldots, \tau_n$ exists, such that $\tau_i \equiv \tau_1$, $\tau_j \equiv \tau_n$ and for every $k = 1, \ldots, n-1$ a resource $R_k$ exists such that $\tau_k$ and $\tau_{k+1}$ both use $R_k$.

Relation $\models$ is an equivalence relation which divides the task set into disjoint subsets. Given a task $\tau_i$, let

$\Gamma_i$ be the set of tasks that interact with $\tau_i$. It is easy to see that $\tau_i \models \tau_j \Rightarrow \Gamma_i \equiv \Gamma_j$.

### 5.2. Description of the algorithm

Every server maintains a list of served tasks. Initially, each server $S_i$ has only its own task $\tau_i$ in the list. As long as no task is blocked, algorithm BWI follows the same rules of algorithm CBS. The BWI has following additional rules:

**Rule D:** if task $\tau_i$ is blocked when accessing a resource $R$ which is locked by task $\tau_j$, then $\tau_j$ is added to the list of server $S_i$. If, in turn, $\tau_j$ is currently blocked on some other resource $R'$, then the chain of blocked tasks is followed, and server $S_i$ adds to its list all the tasks in the chain, until it finds a task which is not blocked[1]. In this way, each server can have more than one task to serve, but only one of these tasks is not blocked.

**Rule E:** When task $\tau_j$ releases resource $R$, one of the tasks that were blocked on $R$ is unblocked, let it be task $\tau_i$. Server $S_i$ must discard $\tau_j$ from its own list; all the servers that added $\tau_j$ to their list while $\tau_j$ was holding $R$ must discard $\tau_j$ and add $\tau_i$.

Analyzing these rules, it is possible to see that algorithm BWI acts like an inheritance protocol: in fact, we consider algorithm BWI the natural extension of the PIP to the CBS. When a high priority task $\tau_i$ is blocked by task $\tau_j$, $\tau_j$ "inherits" server $S_i$. The execution time of $\tau_j$ is now accounted to $S_i$: when the server budget is exhausted $S_i$'s deadline is postponed and $\tau_j$ can continue to execute with the shortest between $S_i$ and $S_j$ deadlines. Algorithm BWI gives an answer to the question arisen in Section 3:

**Q:** *What happens if a server exhausts its budget while it is inside a critical section?*

**A:** As described in rule C, the deadline of the server is postponed and the budget recharged to the maximum. Then, according to the EDF algorithm, the earliest deadline server is selected to execute, and it will serve the only task in its list which is not blocked.

**Example.** The behavior of BWI is shown applying the algorithm to the example of Figure 1. The resulting schedule is depicted in Figure 2.

[1] Recall that the PIP does not prevent deadlock: hence, it may be that all tasks in the chain are currently blocked. Deadlock has to be avoided by means of some other policy: for example we can impose a total order over the set of resources and, when using nested critical sections, resources have to be accessed according to that order.

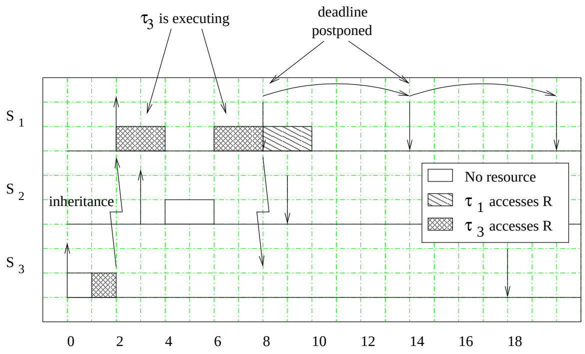


**Figure 2. Algorithm BWI is applied to the example of Figure 1.**

- At time $t = 1$, task $\tau_3$, 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 $\tau_1$ which immediately tries to lock resource $R$. However, the resource is already locked by $\tau_3$: according to rule D, $\tau_3$ is added to the list of $S_1$ and $\tau_1$ is blocked. Now there are two task in $S_1$'s list, but only $\tau_3$ is non-blocked. Hence, $S_1$ dispatches task $\tau_3$.
- At time $t = 3$, server $S_2$ is activated but it is not the earliest deadline server: thus $\tau_3$ continues to execute.
- At time $t = 4$, the budget of server $S_1$ has been depleted: according to rule C, 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 $\tau_2$ is dispatched.
- At time $t = 6$, $S_1$ is again the earliest deadline server; hence task $\tau_3$ is dispatched.
- At time $t = 8$, $\tau_3$ releases the lock on $R$. According to rule E, $\tau_1$ is unblocked, and $\tau_3$ is discarded from the list of server $S_1$. Now $S_1$'s list contains only task $\tau_1$. 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 $\tau_2$ is not influenced by the misbehavior of $\tau_3$ and meets its deadline.

### 5.3. Properties of BWI

In this section, we state the BIP and the HSP for algorithm BWI; all the proofs are omitted due to space constraints, and can be found in [12].

**Theorem 3** *Given a system consisting of $n$ CBS servers with $\sum_{i=1}^{n} U_i \leq 1$, which uses the BWI protocol for accessing shared resources, then no server misses its scheduling deadline, regardless of the parameters assigned to the server.*

Note that the previous theorem states that no *scheduling deadline* (that is, we recall, the server deadline used to schedule the servers with EDF) will be missed, but it does not say anything about task's deadlines. Recall that the goal of a real-time scheduling algorithm is not to miss the deadlines of the HRT tasks. 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 SRT and HRT tasks. In the following, we will give a method to assign the maximum budget $Q_i$ and the period $P_i$ to a server which has to serve an HRT task $\tau_i$.

**Definition 1** *Given a task $\tau_i$, served by a CBS $S_i$ with the BWI protocol, we define the interference time $I_i$ as the maximum time that all other tasks can execute inside server $S_i$ for each instance of $\tau_i$.*

**Lemma 1** *Given a task $\tau_i$, only tasks in $\Gamma_i$ can be added to server $S_i$ and contribute to $I_i$.*

**Theorem 4** *Let $\Gamma$ be a partition of the system tasks, as defined by relation $\models$. Suppose that every $\tau_i \in \Gamma$ is served by a CBS $S_i$ with parameters $Q_i = C_i + I_i$ and $P_i = T_i$, where $C_i$ is the WCET, $T_i$ is the MIT and $I_i$ is the maximum interference time for $\tau_i$. If algorithm BWI is used, then all tasks in $\Gamma$ will meet their deadlines, regardless of the behavior of the other tasks in the system.*

From the previous theorem, we can derive sufficient conditions for scheduling HRT tasks:

- HRT tasks which do not access any shared resource can be guaranteed exactly like in the CBS algorithm: a HRT task $\tau_i$ with a worst case computation time of $C_i$ and a minimum interarrival time of $T_i$ must be assigned to a server $S_i$ with parameters $Q_i \geq C_i$ and $P_i \leq T_i$.
- a HRT task $\tau_i$ which accesses shared resources can be guaranteed using a server $S_i$ with parameters $Q_i \geq C_i + I_i$ and $P_i \leq T_i$, where $I_i$ is the interference time stolen by other tasks belonging to the same partition $\Gamma_i$.

**Considerations.** It is worth to note that, in the case our system consists only of HRT tasks, BWI is not the best algorithm to use. In fact, substituting $Q_i$ and $P_i$ in the admission test, we obtain:

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

which may lead to a lower utilization with respect to Equation (1), because all the blocking times are summed together. Hence, if we are dealing with a hard real-time system, it is probably better to use other scheduling strategies like the one presented in [6].

However, algorithm BWI is more suitable for Open Systems where hard, soft and non real-time applications can coexist and it is impossible to perform an off-line analysis for the entire system. Of course, this comes at a cost of a lower utilization for hard real-time tasks.

## 6. Interference Time Computation

In the general case, an exact computation of $I_i$ is a complex problem. At first glance, the problem is similar to the computation of the blocking times $B_i$ for the PIP. However, when considering algorithm BWI the problem is slightly different. For example, with BWI there is no push–through blocking; in addition, a server can decrease its priority by postponing its deadline while in a critical section. The problem becomes more difficult when tasks are allowed to nest critical sections.

In [16], Rajkumar proved that, when using the basic Priority Inheritance Protocol, the problem of computing the blocking time of a task in the general case (allowing nested critical sections) is NP hard. We claim that the complexity of computing the interference time in the general case is also NP hard. For this reason, we will also consider a simpler case without nested critical sections, for which it is simpler to perform an exact analysis.

In the following, we will present two algorithms for computing the interference time of a task. First, we will consider the general case (with nested critical sections), and we will show that if no deadline in a partition is postponed, then an upper bound to the interference time is given by the same algorithm used for the computation of the blocking time in the basic Priority Inheritance Protocol. Hence, if all tasks in a partition are hard tasks, the classical Priority Inheritance analysis can be used (at the cost of an over-pessimistic admission test).

Second, we will consider the case in which critical sections cannot be nested, and we will develop a more precise analysis. As a result, we will present a more accurate algorithm for the computation of the interference time for a task, and we will explicitly consider interactions between soft and hard tasks.

Before proceeding we need to introduce some preliminary results that will be used in the following analysis.

**Lemma 2** *Under the hypothesis that the deadline of server $S_i$ is never postponed, each task in $\Gamma_i$ can contribute to $I_i$ for at most the duration of one critical section.*

**Lemma 3** *Under the hypothesis that the deadlines of all the servers in $\Gamma_i$ are never postponed, each instance of task $\tau_i$ can be blocked at most once for every resource it uses.*

### 6.1. Nested Critical Sections

In this subsection, we will analyze the general case in which critical sections can be nested. Since the analysis is very complex, we are going to re-use some important results from the classical Priority Inheritance analysis. In order to do that, we need to suppose that all tasks inside a partition $\Gamma_i$ are HRT tasks. In this case, the following lemma helps us:

**Lemma 4** *Under the hypothesis that the deadlines of all the servers in $\Gamma_i$ are never postponed, task $\tau_i$, served by server $S_i$ with budget $Q_i$ and period $P_i$, can be blocked only by tasks served by servers having longer periods (like in fixed priority systems).*

Looking at Lemma 2, Lemma 3, and Lemma 4, we can see that they are equivalent to the PIP properties proved in [19]. Hence, the interference time $I_i$ can be computed as the blocking time for the basic PIP case. Algorithms for computing such a blocking time have been proposed in [16] and [4].

Hence, in order not to miss any deadline in $\Gamma_i$ it is sufficient to compute the interference times using one of those algorithms, and to assign $Q_j = C_j + I_j$ to each task $\tau_j \in \Gamma_i$, according to Theorem 4.

### 6.2. Analysis with no nested critical section

As discussed above, two distinct partitions $\Gamma_i$ and $\Gamma_k$ are temporally isolated. In the general case, assigning $Q_j = C_j + I_j$ to all the tasks $\tau_j \in \Gamma_i$ (where $I_i$ is computed as $B_i$ in the simple PIP case) is a sufficient condition for not postponing any deadline in all the tasks inside $\Gamma_i$. However, it is not clear if this condition is also necessary.

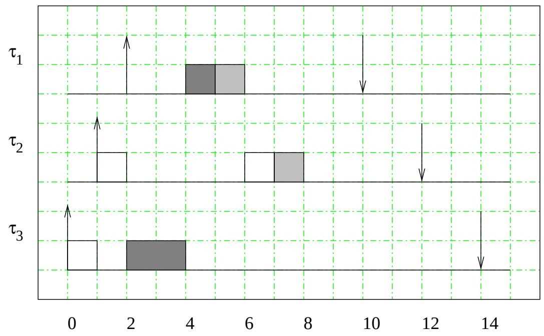


**Figure 3. Example of push-through blocking.**

Moreover, it is important to allow the cooperation of HRT and SRT tasks. This can be possible if are able to guarantee that the deadlines of a single server inside a partition are not postponed (regardless of what happens to the other servers belonging to the partition).

While it is very difficult to fulfill these requirements in the general case, it is easier in the case we disallow nested critical sections. For example, it is easy to see that the schedulability condition presented in the previous subsection is not necessary if there are not nested critical sections. In order to show this fact, let's consider the example in Figure 3: in this example we show that using algorithm BWI the phenomenon of push–through blocking do not arise.

**Example.** Three tasks, $\tau_1$, $\tau_2$, and $\tau_3$ ($T_1 < T_2 < T_3$), belong to the same partition $\Gamma$, since $\tau_1$ shares resource $R_1$ with $\tau_2$, and resource $R_2$ with $\tau_3$. However, $\tau_2$ and $\tau_3$ do not directly share any resource (they are in the same partition because they both share a resource with $\tau_1$). First, suppose that we want to schedule these three tasks using EDF and the basic PIP. If we compute the blocking time, we obtain $B_2 = \xi_{3,2}$, due to a push through blocking of $\tau_3$ on $\tau_2$. In Figure 3, $\tau_2$ incurs in this blocking time at time $t_1$: in fact, at that time, $\tau_1$ tries to access $R_2$ and is blocked by $\tau_3$ which in turn inherits its deadline (preempting $\tau_2$) for a time $\xi_{3,2}$.

Now, suppose that we want to use BWI. In this case, $\tau_3$ inherits server $S_1$, but cannot inherit server $S_2$. This means that $S_2$ do not experience any interference by $\tau_3$ and (since $T_1 < T_2$), $I_2 = 0$. Hence, we can simply assign $Q_1 = C_1 + max\{\xi_{2,1}, \xi_{3,2}\}, Q_2 = C_2, Q_3 = C_3$ and every task is guaranteed to meet its deadlines. If instead we compute the interference times as the blocking times in the PIP case, we are overestimating $I_2$.

The above example gives us an idea about how to develop a more precise computation of the interference times when critical sections are not nested. In this case, two tasks can share a server only if they directly share a resource, as shown in the following Lemma.

**Lemma 5** *If no task in partition $\Gamma_i$ uses nested critical sections, server $S_i$ can be inherited by task $\tau_j$ if and only if $\tau_i$ and $\tau_j$ directly share a resource.*

According to Lemmas 5, 2 and 3, it is possible to compute the interference time $I_i$ stolen to $S_i$ by other tasks belonging to $\Gamma_i$ when there are not nested critical sections, by computing all the possible "legal" interference times, and finding the maximum between them. An interference time is legal if it is generated according to the following properties: $\tau_i$ can be blocked at most once per each resource it uses and at most once per each low priority task sharing resources with it.

The algorithm performing such a computation is shown in Figure 4, where $\mathcal{R}_i$ denotes the set of resources accessed by task $\tau_i$ and $N$ is the total number of tasks in the system. Tasks are ordered by server period ($P_i < P_{i+1}$). Function `interference(j, ℛ)` returns the interference time caused by tasks $\tau_j, \tau_{j+1}, \ldots$ on a shorter period task $\tau_j$ accessing the resources in set $\mathcal{R}$.

```
int interference(int j, set ℛ)
{
  if (j == N + 1) return 0;
  max = interference(j + 1, ℛ)
  foreach R ∈ ℛ_j ∩ ℛ {
    res = ξ_{j,R}+ interference(j + 1, ℛ − {R});
    if (res > max) max = res;
  }
  return max;
}
```

**Figure 4. Algorithm for computing the interference time for $\tau_i$.**

The algorithm works as follows: if $j == N + 1$, it means that the recursive computation stopped, ($j == N+1$ is the end-of-recursion condition), hence the function returns 0. Otherwise, it starts to recursively compute the interference produced by task $\tau_j$ and by the longer period tasks $\tau_{j+1}, \ldots, \tau_N$, by exploring all the possible cases. The first considered case is the one in which task $\tau_j$ does not contribute to the interference $I_i$. In this case, the interference given by all the lower priority tasks is computed by `by interference(j + 1, ℛ)`; in facts, line `max = interference(j + 1, ℛ)`, stores it in `max` to check if this interference is bigger than in the other cases. All the other cases are evaluated in the `foreach` loop, that iterates on all the resources shared by $\tau_i$ and $\tau_j$. In facts, condition $R \in \mathcal{R}_j \cap \mathcal{R}$ selects **only those resources that are used both by $\tau_j$ and $\tau_i$.**

Note that when `interference()` is recursively invoked, the used resource $R$ is subtracted from $\mathcal{R}$: in this way, each resource is used only once in the computation of $I_i$.

Finally, the interference $I_i$ on task $\tau_i$ can be computed as $I_i = \texttt{interference}(i + 1, \mathcal{R}_i)$.

### 6.3. Sharing resource between HRT and SRT tasks

According to the above discussion, it is possible to reserve CPU bandwidth to the HRT tasks in a given partition so that they are all guaranteed to meet their deadlines. However, sometimes tasks in the same partition have different criticalities.

As an example, consider a HRT task and a SRT task which access the same shared resource through a common shared library. If the critical sections are implemented as library functions whose execution time can be bounded, then we can still be able to compute the interference time that the soft task "steals" to the hard task. Indeed, this is a common case in a real operating system.

For this reason, it can be desirable to guarantee a single HRT task $\tau_i$ even in the case in which there are SRT tasks in its partition. In this case, the bandwidth assigned to the SRT tasks does not influence the behavior of the HRT task, and can be allocated based on the average case. However, the server for the HRT task must always be assigned a budget and a period based on the worst case.

In this case, Lemma 2 and Lemma 3 are still valid, but Lemma 4 is not. As an example, consider a SRT task $\tau_j$ that is served by a server $S_j$ with a period $P_j < P_i$. If $\tau_j$ behaves well and the server deadline is not postponed, $S_i$ cannot preempt $S_j$. However, if $\tau_j$ executes for more than declared, it can deplete the server budget and postpone its deadline, decreasing its dynamic priority. In this case, it can later be preempted by $S_i$ while inside a critical section, and block $\tau_i$, contributing to its interference time $I_i$.

Hence, when considering SRT tasks in the same partition of a HRT task $\tau_i$, the algorithm for computing $I_i$ must consider *all the tasks* that are sharing resources with $\tau_i$ and not only those with longer periods. In practice, $I_i$ can be computed as previously, except that the `interference()` function must be iterated on all the tasks in the partition (and not only on the tasks with period $T_j > T_i$). Hence, we can compute the interference as $I_i = \texttt{interference}(1, \mathcal{R}_i)$, considering $\mathcal{R}_i = \emptyset$ in line "**foreach** $R \in \mathcal{R}_j \cap \mathcal{R}$ { " in order not to consider the interference of $\tau_i$ on itself.

The results presented in this section section are valid for partitions in which tasks do not use nested critical sections. If, instead, nested critical sections are present in the partition, only the temporal isolation (Theorem 3) and hard schedulability (Theorem 4) properties are verified. In this case, the interference times must be computed as the blocking times in the priority inheritance case. However, as shown, this condition is not tight. We are currently developing a new algorithm for computing $I_i$ in the case of nested critical sections.

## 7. Conclusions and future work

In this paper, we presented the Bandwidth Inheritance Protocol, a novel scheduling discipline that permits to share resources between applications in an open system. Coherently with the Resource Reservation approach, the BWI protocol provides some temporal isolation properties without requiring any a-priori knowledge about the the structure and on 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 the most important formal properties of the BWI algorithm, that are presented and proved in the paper.

As explained in Section 6, we are still working on a more accurate schedulability analysis for the general case considering nested critical sections. Moreover, we are planning to develop a probabilistic analysis for SRT tasks.

For what concerns the algorithm, we are currently analyzing better strategies for coping with "misbehaved" tasks that use the shared resource more that the declared $\xi$ time. Roughly speaking, these tasks can "steal" CPU bandwidth to the other tasks belonging to their partition $\Gamma_i$, compromising their guarantee. We are currently testing different mechanisms and policies that can reduce (but not eliminate) this effect, in order to penalize the misbehaved tasks inside a partition.

## References


[1] L. Abeni. Server mechanisms for multimedia applications. Technical Report RETIS TR98-01, Scuola Superiore S. Anna, 1998.

[2] L. Abeni and G.Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, december 1998. IEEE.

[3] T. P. Baker. A stack-based allocation policy for real-time processes. In *IEEE Real-Time Systems Symposium*, december 1990.

[4] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Kluwer Academic Publishers, Boston, 1997.

[5] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings of the IEEE Real-Time Systems Symposium*, Dec. 2000.

[6] M. Caccamo and L. Sha. Aperiodic servers with resource constraints. In *Proceedings of the IEEE Real-Time Systems Symposium*, London. UK, December 2001.

[7] H. Chu and K. Nahrstedt. CPU service classes for multimedia applications. In *Proceedings of the IEEE International Conference on Mutimedia Computing and Systems*, Florence, Italy, June 1999.

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

[9] Z. Deng and J. W. S. Liu. Scheduling real-time applications in open envirovment. In *IEEE Real-Time Systems Symposium*, San Francisco, December 1997.

[10] T. Ghazalie and T. Baker. Aperiodic servers in a deadline scheduling environment. *Journal of Real-Time System*, 9, 1995.

[11] G. Lipari. *Resource Reservation in Real-Time Systems.* PhD thesis, Scuola Superiore S.Anna, 2000.

[12] G. Lipari and L. Abeni. Sharing resources in a open system. Technical report, Scuola Superiore S.Anna, Retis Lab, 2001.

[13] G. Lipari and G. Buttazzo. Schedulability analysis of periodic and aperiodic tasks with resource constraints. *Journal of Systems Architecture*, 46:327–338, 2000.

[14] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1), 1973.

[15] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. In *Proceedings of IEEE international conference on Multimedia Computing and System*, may 1994.

[16] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach.* Kluwer Academic Publishers, 1991.

[17] R. Rajkumar, L. Abeni, D. D. Niz, S. Gosh, A. Miyoshi, and S. Saewong. Recent developments with linux/RK. In *Proceedings of the Real Time Linux Workshop*, Orlando, Florida, December 2000.

[18] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the 4th Real-Time Computing Systems and Application Workshop.* IEEE, November 1997.

[19] L. Sha, R. Rajkumar, and john P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE transaction on computers*, 39(9), September 1990.