An efficient implementation of the BandWidth Inheritance protocol for handling hard and soft real-time applications in the Linux kernel [*]

Dario Faggioli, Giuseppe Lipari and Tommaso Cucinotta
e-mail: d.faggioli@sssup.it, g.lipari@sssup.it, t.cucinotta@sssup.it
Scuola Superiore Sant'Anna, Pisa (Italy)

Abstract

This paper presents an improvement of the Bandwidth Inheritance Protocol (BWI), the natural extension of the well-known Priority Inheritance Protocol (PIP) to resource reservation schedulers. The modified protocol allows for a better management of nested critical section, removes unneeded overheads in the management of task block and unblock events, and introduces a run-time deadlock detection mechanism at no cost.

Also, an implementation of the new protocol on the Linux kernel is presented, along with experimental results gathered while running some synthetic application load. Presented results prove the effectiveness of the proposed solution in reducing latencies due to concurrent use of resources and in improving temporal isolation among groups of independent tasks. Also, we show that the introduced overhead is low and negligible for the applications of interest.

1 Introduction

Embedded systems are nowadays part of our everyday life. As their popularity and pervasiveness increases, these devices are required to provide more and more functionality, thus their complexity grows higher and higher. In particular, embedded systems now not only find application in the domain of self-contained, hard real-time, safety critical systems, but their applicability is undergoing a tremendous growth in the range of soft real-time applications, with various degrees of time-sensitiveness and QoS requirements.

The requirements on the real-time operating system platform on which such applications are implemented increases in parallel. The RTOS must be robust (also to timing faults), secure (also to denial of service attacks) and dependable. Finally, it must support open and dynamic applications with QoS requirements.

For these reasons, Linux is becoming the preferred choice for a certain class of embedded systems. In fact, it already provides many of the needed services: it has an open source license and an huge base of enthusiastic programmers as well as a lot of software running on it. Furthermore, it presents a standard programming interface.

Due to the increasing interest from the world of embedded applications, Linux is also being enriched with more and more real-time capabilities [13], usually proposed as separate patches to the kernel, that are progressively being integrated into the main branch. For example, a small group of developers has proposed the PREEMPT_RT patch which greatly reduces the non preemptable sections of code inside the kernel, thus reducing the worst-case latencies. The support for priority inheritance can be extended to in-kernel locking primitives and a great amount of interrupt handling code has been moved to schedulable threads. From a programming point of view, Linux now supports almost entirely the Real-Time POSIX extensions, but the mainstream kernel still lacks support for such extensions like sporadic servers or any Resource Reservation techniques, that would allow the kernel to provide temporal isolation.

Resource Reservation (RR) is an effective technique to schedule hard and soft real-time applications and to provide temporal isolation among them in open and dynamic systems. According to this technique, the resource bandwidth is partitioned between the different applications, and an overrun in one of them cannot influence the temporal behavior of the others.

In standard Resource Reservation theory, tasks are considered as independent. In practical applications, instead, tasks may interact due to the concurrent access to a shared resource, which commonly requires the use of a mutex semaphore in order to serialize those accesses. For example, a Linux application may consist of one multi-threaded process, and threads may interact and synchronize each other through pthread mutexes. In this case, it is necessary to use appropriate resource access protocols in order to bound the priority inversion phenomenon [20]. While implementations of scheduling policies for QoS guarantees on Linux exist [19, 1], they do not provide support for appropriate management of interactions among threads.

[*]This work has been partially supported supported by the European FRESCOR FP6/2005/IST/5-034026 project.

Contributions of this paper. In a previous work [14], the Bandwidth Inheritance (BWI) Protocol has been proposed as the natural extension of the Priority Inheritance Protocol [20] in the context of Resource Reservations.

In this paper, we extend that work with three important contributions: first, we propose a simplification of the BWI protocol that allows for a much more efficient implementation, for both memory and computational requirements. This is particularly relevant in the context of embedded systems. We also prove that the simplifications do not compromise the original properties of the protocol. Second, we present an efficient deadlock detection mechanism based on BWI, that does not add any overhead to the protocol itself. Third, we present a real implementation of the protocol within the Linux operating system, based on the AQuoSA Framework and the pthread mutex API, that makes the protocol widely available for soft real-time applications.

Also, we present experimental results that highlight the effectiveness of our BWI implementation in reducing latencies.

Organization of the paper The remainder of the paper is organized as follows: Sec. 2 provides some prerequisite definitions and background concepts. Sec. 3 summarizes previous and alternative approaches to the problem. Sec. 4 describes our modification to the BWI protocol, focussing on the achieved improvements. Sec. 5 focusses on details about the actual implementation of the modified protocol on Linux, while Sec. 6 reports results gathered from experimental evaluation of the described implementation. Finally, Sec. 7 draws some conclusions, and quickly discusses possible future work on the topic.

2 Background

2.1 System Model

A real-time task τ_i is a sequence of real-time jobs $J_{i,j}$, each one modeled by an arrival time $a_{i,j}$, a computation time $c_{i,j}$ and an absolute deadline $d_{i,j}$. A periodic (sporadic) task is also associated a relative deadline D_i, such that $\forall j, d_{i,j} = a_{i,j} + D_i$, and a period (minimum inter-arrival time) T_i such that $a_{i,j+1} \geq a_{i,j} + T_i$.

Given the worst case execution time (WCET) is $C_i = \max_j\{c_{i,j}\}$, the processor utilization factor U_i of τ_i is defined as $U_i = \frac{C_i}{T_i}$.

In this paper, we consider *open systems* [9], where tasks can be dynamically activated and killed. In open systems, tasks belonging to different, independently developed, applications can coexist. Therefore, it is not possible to analyze the entire system off-line.

Also, hard real-time tasks must respect all their deadlines. Soft real-time tasks can tolerate occasional violations of their timing constraints, i.e., it could happen that some job terminates after its absolute deadline. The number of missed deadlines over a given interval of time is often used as a valid measure for the QoS experienced by the user.

An effective technique to keep the number of missed deadlines under control is to use Resource Reservation [18, 2] scheduling algorithms. According to these techniques, each task is assigned a virtual resource (vres[1]), with a maximum budget Q and a period P. Resource Reservations provide the *temporal isolation* property to independent tasks: a task is guaranteed to be scheduled for at least Q time units for every period of P time units, but at the same time, in order to provide guarantees to all tasks in the system, the mechanism may not allow the task to execute for more than that amount.

Many RR algorithms have been proposed in the literature, for both fixed priority and dynamic priority scheduling, and the work presented in this paper is applicable to all of them. However, our implementation is based on a variant of the Constant Bandwidth Server.

2.2 Critical Sections

Real-time systems are often designed as a set of concurrent real-time tasks interacting through shared memory data structures. Using classical mutex semaphores in a real-time system is prone to the well known problem of unbounded priority inversion [20]. Dealing correctly with such a phenomenon is very important, since it can jeopardize the real-time guarantees and cause significant QoS degradation and waste of resources. Effective solutions have been proposed in classical real-time scheduling algorithms, such as the Priority Inheritance Protocol (PIP), the Priority Ceiling Protocol (PCP) [20] and the Stack Resource Policy [4].

In particular, PIP is very simple to implement, it can work on any priority-based scheduling algorithm (both for fixed and dynamic priority) and does not require the user to specify additional scheduling parameters. According to PIP, when a task τ_h is blocked trying to access a critical section already locked by a lower priority task τ_l, τ_h lends its priority to τ_l for the duration of the critical section. When τ_l releases the critical section, it is given back its priority. In this paper we discuss an extension of the PIP for resource reservations.

2.3 Constant Bandwidth Server

The Constant Bandwidth Server (CBS [2]) is a well-known RR scheduling algorithm working on top of Earliest Deadline First (EDF [15]). As in any RR algorithm, each task is associated a *virtual resource* with parameters Q (the maximum budget) and P (the period). Each vres can be seen as a sporadic task with worst-case execution time equal to Q and minimum inter-arrival time equal to P. The

[1]We use the term virtual resource instead of the classical term server, to avoid confusion in readers that are not expert of aperiodic servers in real-time scheduling.

EDF system scheduler uses the vres parameters to schedule the associated tasks.

The fundamental idea behind CBS is that each request for a task execution is converted into a request of the corresponding vres with associated a dynamic deadline, calculated taking into account the bandwidth of the vres. When the task tries to execute more than its associated vres budget, the vres deadline is postponed, so that its EDF priority decreases and it is slowed down.

The CBS algorithm guarantees that overruns occurring on task τ_i only affect τ_i itself, so that it can *steal* no bandwidth assigned to any other task. This form of the *temporal protection* property is also called Bandwidth Isolation Property (BIP [2]).

2.4 Critical Sections and the CBS algorithm

Unfortunately, when two tasks belonging to different vres share mutually exclusive resources, the bandwidth isolation property is broken. In fact, one assumption of every RR algorithm is that the vres with the highest priority should be executed at each instant. However, when using mutex semaphores, a task (and its corresponding vres) could be blocked on the semaphore and not able to execute. Even if we are able to bound the blocking time of each vres by using an appropriate resource access protocol like the PIP, we still have to perform a careful off-line analysis of all the critical section code, in order to be able to compute the blocking time and use it in the admission control policy. Moreover, using the PIP with resource reservations is not straightforward, since it is not clear, for example, the budget of which vres should be depleted and the deadline of which should be postponed when an inheritance is in place.

Such a limitation is hard to tolerate in modern systems, since multi-threaded applications are quite common, especially within multimedia environments, and the various tasks often need to communicate by means of shared memory data structures needing mutual exclusive access. For this reason the BandWidth Inheritance protocol (BWI [14]) has been proposed as an extension of PIP suitable for reservation based systems.

2.5 The BandWidth Inheritance Protocol

The description of BWI in this section is not meant to be exhaustive, and the interested reader is remanded to [14] for any further detail. The BWI protocol works according to the following two rules:

BWI blocking rule when a task τ_i blocks trying to access a shared resource R already owned by another task τ_j, then τ_j is added to the list of the tasks served by the vres S_i of τ_i. If τ_j is also blocked, the chain of blocked tasks is followed until one that is not blocked is found, and all the encountered tasks are added to the list of S_i;

BWI unblocking rule when task τ_j releases the lock on R and wakes up τ_i, then τ_j is discarded from the list of vres S_i. If other vres added τ_j to their list, they have to replace τ_j with τ_i.

BWI is considered the natural extension of PIP to resource reservations. In fact, when a task is blocked on a mutex, the lock-owner task inherits its entire vres. In other words, the owner task τ_j can execute on its own vres *and* in the inherited vres (the one with the highest priority), so that the blocking time of τ_i is shortened. Most importantly, the BWI protocol preserves the bandwidth isolation properties between non-interacting tasks.

A blocking chain between two tasks τ_i and τ_j is a sequence $H_{i,j} = \{\tau_1, R_1, \tau_2, \ldots, R_{n-1}, \tau_n\}$ of alternating tasks and resources, such that, the first and the last tasks in the chain are $\tau_1 = \tau_i$ and $\tau_n = \tau_j$, and they access, respectively, resources R_1 and R_{n-1}; each task τ_k (with $1 < k < n$) accesses resource R_k in a critical section nested inside a critical section on R_{k-1}. For example, the following blocking chain $H_{1,3} = \{\tau_1, R_1, \tau_2, R_2, \tau_3\}$ consists of 3 tasks: τ_3 accesses resource R_2 with a mutex m_2; τ_2 accesses R_2 with a critical section nested inside a critical section on R_1; τ_1 accesses R_1. At run-time, τ_1 can be directly blocked by τ_2 and indirectly blocked by τ_3.

Two tasks τ_i and τ_j are *interacting* if and only if there exists a *blocking chain* between τ_i and τ_j. The BWI protocol guarantees bandwidth isolation between pairs of non interacting tasks: if τ_i and τ_j are not interacting, the behavior of τ_i cannot influence the behavior of τ_j and viceversa.

3 Related Work

Many practical implementations of the RR framework have been proposed since now. In the context of general-purpose OSes, the most widely known is probably Linux/RK [19], developed as a research project at CMU and later commercialized by TimeSys. More recently, an implementation of the Pfair [5] scheduling algorithm for reserving shares of the CPU in a multi-processor environment, in the Linux kernel, has been developed by H. Anderson et al. [1] in the $LITMUS^{RT}$ project. However, to the best of our knowledge, these approaches did not provide support for mutually exclusive resource sharing.

A quite common approach [12] when dealing with critical sections and capacity-based servers is to allow a task to continue executing when the vres exhausts its budget and the task is inside a critical section. The extra budget the task has been provided is then subtracted from the next replenishments. Coupling this strategy with the SRP [4] reduces the priority inversion phenomenon to the minimum. The technique has been applied to the CBS algorithm in [6]. In general, this approach is very effective for static systems where all information on the application structure and on the tasks is known a-priori. However, it is not adequate to open and dynamic systems. In fact, in order to compute the *preemption level* of all the resources, the protocol requires that the programmer declares in advance all the tasks that will access a certain resource. Moreover, the vres parameters have to be fixed, and cannot be easily changed at run-time. Finally, this approach cannot protect the system from overruns of tasks while inside a critical section.

An approach similar to BWI protocol has been implemented in the L4 microkernel [21]. The Capacity-Reserve Donation (CREDO) is based on the idea of maintaining a *task state context* and a *scheduling context* as two, separated and independently switchable data structures. According to the authors, the technique can be applied to either PIP or Stack-Based PCP. Although being quite effective this mechanism is thoroughly biased toward message passing microkernel system architectures, and cannot be easily transposed to shared memory systems. Furthermore, to enable PIP or SRP on top of CREDO, it is necessary to carefully assign the priority of the various tasks in the system, otherwise the protocol can not work properly.

Also, in the context of micro-kernels, Mercer et al. [16] presented an implementation of a reservation mechanism on the real-time Mach OS. Interestingly, the implementation allowed for accounting the time spent within kernel services, when activated on behalf of user-level reserved tasks, to the reserves of the tasks themselves. Kernel services might be considered, in such context, as shared resources to which applications concurrently access. However, the problem of regulating application-level critical sections managed through explicit synchronization primitives, so to avoid priority inversion, is not addressed.

All this given, we say BWI is a suitable protocol for resource sharing in open, dynamic embedded systems, for the following reasons:

- BWI is completely transparent to applications. As with the PIP, the user must not specify additional scheduling parameters, as “ceiling” or “preemption-level”;
- BWI provides bandwidth isolation between non-interacting tasks, even in the case of tasks that overrun inside a critical section. Therefore, it is not necessary to implement any additional budget protection mechanism for the critical section length;
- BWI is neutral to the underlying RR algorithm and does not require any special property of the scheduler. This allows us any modification of the scheduling algorithm without the need to reimplement BWI.

4 Improvements to the protocol

In this section, we focus on the limitations of the original formulation of the BWI protocol, and propose two modifications to its rules that, without compromising the guarantees, allow for a simplification of the implementation. In what follows, we assume that each task competing for access to shared resources is served by a dedicated vres.

4.1 Nested Critical Section Improvement

In presence of nested critical sections, the two BWI rules do not correctly account for all possible situations. Consider the case of a task τ_i that blocks on another task τ_j, after having been added to some vres S_h, different from its original one (S_i), due to previous inheritance (i.e., another task τ_h is blocked waiting for τ_i to release some lock). By following the blocking rule of BWI, task τ_j is added only to S_i, but, because also τ_h is blocked waiting for τ_i, the blocking delays induced on τ_h may be reduced if τ_i would have added to S_h as well.

In general, we are saying that τ_j should be attached to all the vres to which τ_i was bound (both directly and by inheritance due to BWI) before blocking itself.

As an example, consider the situation depicted in Figure 1, where we have four tasks, τ_A, τ_B, τ_C and τ_D, each bound to its own vres. S_A has $U_{S_A} = 6/25$ utilization, S_B has $U_{S_B} = 3/20$ utilization, $U_{S_C} = 3/15$ and $U_{S_D} = 4/10$. τ_A, τ_C and τ_D use mutex m_1 and τ_A and τ_B use mutex m_2. Also notice τ_A acquires the second mutex while holding the first one (nested critical section).

In this figure, and in all the figures of Sec. 6, symbols $L(_i)$ and $U(_i)$ denote wait and signal operations on mutex m_i. Light-gray filled rectangles denote tasks executing inside a critical section, with the number in the rectangle being the acquired mutex. Vertical dashed arrows denote the time instants a task is attached (and detached) to a vres different from its own one due to BWI inheritance. White filled ones denote a task being able to execute inside a vres different from its original one thanks to BWI, with the number in the rectangle being the mutex that caused the inheritance.

At time $t = 7$ τ_C blocks on mutex m_1, owned by τ_A, and τ_A is added to the vres of τ_C. Then, at time $t = 11$, τ_D blocks on mutex m_1 too. Again, τ_A is attached to the vres of τ_D, and it can now run inside any of the three vres exhausting their budget on time instants $t = 9$ (for τ_C) and $t = 13$ (for τ_D).

Suppose that at time $t = 15$ τ_A blocks on mutex m_2: honoring the original BWI rule, τ_B is added only to the vres of τ_A, whereas τ_A remains attached to the vres of τ_D and τ_C, although being blocked. In this way, τ_B can not take advantage of the bandwidth assigned to τ_C and τ_D, delaying their own unblocking. Notice this behavior is incidental: if τ_C and τ_D would start executing *after* τ_A blocks on m_2, then τ_B would have been added to all the vres of the three tasks.

4.2 Simplified BWI blocking rule

In order to face with the just shown issue, we propose a rewrite of the BWI blocking rule as follows:

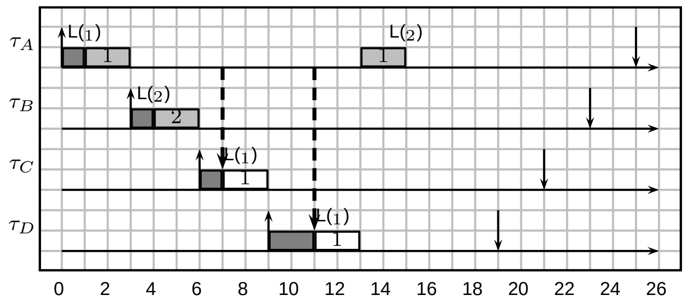

Figure 1: example of nested blocking situation not correctly handled by the original BWI protocol

new BWI blocking rule when a task τ_i blocks while trying to access a shared resource R already owned by another task, the chain of blocked tasks is followed until one that is not blocked is found, let it be τ_j. Then, τ_j is added to the vres τ_i belongs to. Furthermore, τ_i is replaced with τ_j in all vres that previously added τ_i to their task list due to this rule, if any.

Basically, the difference between the original and the new BWI formulation may be highlighted in terms of the invariant property of the algorithms. To this purpose, consider the wait-for-graph at an arbitrary point in time, where nodes represent tasks and an edge from node τ_A to node τ_B denotes that some resource is held by τ_B and requested by τ_A. Now, consider any ready-to-run task τ_j, and let G_j denote the set of tasks directly or indirectly blocked on mutexes held by τ_j. Well, the new BWI formulation has the invariant property that the running task is bound to all the vres of the tasks in G_j, so that each vres of any such task needs to bound dynamically (due to BWI) at most one task (the ready-to-run one, τ_j), in addition to the one (blocked) explicitly bound to it. Furthermore, the original BWI formulation also required each of these vres to be dynamically bound (due to BWI) to all (blocked) tasks found in the blocking chain: from the task explicitly bound to it up to τ_j.

Therefore, the following property holds for the new BWI formulation:

BWI one-blocked-task property Given a vres S_i, at each time instant it can have in its list of tasks only one other task in addition to its original task τ_i.

This is achieved because, according to the new blocking rule, every time a task τ_i blocks, if its mutex-owner is blocked too, we need to follow the blocking chain until a ready task τ_j is found, and add it to S_i. Obviously a mechanism which make it possible to traverse such a chain of blocked task has to be provided by the operating system (as the Linux kernel does). Furthermore, if τ_i was on its own already bound to other servers due to BWI, we need to replace τ_i with τ_j in those servers, keeping at 1 the number of additionally bound tasks in those servers. Since each task τ_i can, at time t, be blocked at most by only one other task τ_j, the just stated property always holds.

As an example if we have:

- task τ_A owning m_1 and running;
- task τ_B owning m_2 and blocked on m_1 (owned by τ_A);
- task τ_C owning m_3 and blocked on m_2 (owned by τ_B);

when a fourth task, task τ_D, tries to lock m_3, it blocks. According to original BWI protocol, we have to bind τ_C, τ_B and τ_A to the vres of τ_D. In the new formulation we only bind τ_A (the sole running task).

The main consequence of the new blocking rule on the implementation is lower memory occupation of the data structure needed by BWI, what is particularly relevant mainly in the context of embedded systems, especially when the task set is characterized by tight interactions and nested critical sections would cause the run-time creation of non-trivial blocking trees. In fact, even this is not necessarily of practical relevance (in a well-designed system interactions should be kept at the bare minimum), the memory overhead complexity of the new BWI formulation is *linear* in the number of interacting tasks, while in the original BWI formulation it was *quadratic*.

4.3 Correctness

The original description of the BWI protocol [14] was accompanied by a proof of correctness in the domain of hard real-time systems. The proof aimed at showing that, if the maximum interferences of the interacting tasks is properly accounted for in the admission control test, then the system scheduled with BWI allows all tasks to respect their deadlines. The following result constitutes the basis for the proof:

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

Proof. The lemma is clearly valid before any task blocks. Assume that the lemma holds true until time t, when task τ_i blocks on a resource, and, following the chain of blocked tasks, τ_j is selected. Our modified version of the BWI blocking rule only differs from the original one in stating that the running task τ_j replaces τ_i in *all* the lists of the vres where task τ_i has been bound (as opposed to only the last one). Notice that τ_i has been added to these vres because of one running task in each of them blocked on a resource directly or indirectly shared with τ_i. Thus, just before blocking, if the lemma is true, τ_i was the only ready task in all of them. Since τ_i is blocking, τ_j becomes on its turn the only runnable task in every vres, so the lemma continues to hold. ☐

Although this result may be used to prove that the new protocol is still correct, further details are omitted for the sake of brevity.

4.4 Lightweight Deadlock Detection

Finally, we improved the BWI protocol by adding the ability to detect deadlocks at run-time [2]:

Deadlock detection rule when applying the new BWI blocking rule to a task τ_i that blocks, for each task τ_k encountered while iterating the blocking chain of τ_i, if $\tau_k = \tau_i$, then a deadlock attempt is detected.

This is a lightweight yet effective approach for deadlock detection and below is a proof of correctness of it.

Theorem 1. *If there is a deadlock situation, the protocol detects it at run-time.*

Proof. As stated by Coffman et al. [7], necessary condition for deadlock is that there is a cycle in the wait-for-graph of the system. To detect a deadlock, every time a task τ_i blocks on another task τ_j, we have to add an edge from τ_i to τ_j in the graph and check if a cycle has been created. Suppose that just before time t there are no cycles, and so no deadlock is in place, and that at time t task τ_i blocks. Also consider the fact that, from any task, at most one edge can exit, directed toward the task's lock-owner. Therefore, if a cycle has been created by the blocking of task τ_i, then τ_i must be part of the cycle. Hence, following the blocking chain from τ_i, if a deadlock has been created, we will come back to τ_i itself, and so our algorithm can detect all deadlocks. □

It is noteworthy that the deadlock detection rule may be realized with practically zero overhead, adding a comparison in the search for a ready-to-run task, while we are following the chain of blocked tasks, according with the blocking rule.

5 BandWidth Inheritance Implementation

5.1 The Linux Kernel

Although not being a real-time system, the 2.6 Linux kernel includes a set of features making it particularly suitable for *soft* real-time applications. First, it is a fully-preemptable kernel, like most of the existing real-time operating systems, and a lot of effort has been spent on reducing the length of non-preemptable sections (the major source of kernel latencies). It is noteworthy that the 2.6 kernel series introduced a new scheduler with a bounded execution time, resulting in a highly decreased scheduling latency. Also, in the latest kernel series, a modular framework has been introduced that will possibly allow for an easier integration of other scheduling policies. Second, although being a general-purpose time-sharing kernel, it includes the POSIX priority-based scheduling policies `SCHED_FIFO` and `SCHED_RR`, that may result useful for real-time systems. Third, the recently introduced support in the kernel mainstream of the support for high-resolution timers is of paramount importance for the realization of high-precision customized scheduling mechanisms, and for the general performance of soft real-time applications.

Unfortunately, the Linux kernel has also some characteristics that make it impossible to realize hard real-time applications on it: the monolithic structure of the kernel and the wide variety of drivers that may be loaded within, the impossibility to keep under control all the non-preemptable sections possibly added by such drivers, the general structure of the interrupt management core framework that privileges portability with respect to latencies, and others. However, the wide availability of kernel drivers and user-space libraries for devices used in the multimedia field constitutes also a point in favor of the adoption of Linux in such application area. Furthermore, recent patches proposed by the group of Ingo Molnar to the interrupt-management core framework, aimed at encapsulating device drivers within kernel threads, are particularly relevant as such approaches would highly increase predictability of the kernel behaviour.

At the kernel level, mutual exclusive access to critical code sections is managed in Linux through classical spinlocks, RCU primitives, mutexes and rt-mutexes, a variant of mutexes with support for priority inheritance. The latter ones are particularly worth to cite, because they allow for the availability of the PIP in user-level synchronization primitives. This is not only beneficial for time-sensitive applications, since thanks to the rt-mutex run-time support, we have been able to implement the BandWidth Inheritance protocol without any modification to the kernel mutex-related logics.

At the user/application level, locking and synchronization may be achieved by means of futexes (Fast Uerspace muTEX [11]) or of standard POSIX mutexes, provided by the GNU C Library [10] and, on their turn, implemented through futexes. One remarkable peculiarity of futexes and POSIX mutexes, is that their implementation on Linux involves the kernel (thus a relatively expensive system call) only when there is a contention that requires arbitration.

5.2 The AQuoSA Framework

The CPU scheduling strategies available in the standard Linux kernel are not designed to provide temporal protection among applications, therefore they are not suitable for time-sensitive workloads. The AQuoSA framework [17] (available at http://aquosa.sourceforge.net) aims at filling this gap, enhancing a standard GNU/Linux system with scheduling strategies based on the RR techniques described in Sec. 2.3.

[2]The method proposed here differs from the method proposed in [14], as it is much more efficient.

AQuoSA is designed with a layered architecture. At the lowest level[3] there is a small patch (Generic Scheduler Patch, GSP) to the Linux kernel that allows dynamically loaded modules to customize the CPU scheduler behaviour, by intercepting and reacting to scheduling-related events such as: creation and destruction of tasks, blocking and unblocking of tasks on synchronization primitives, receive by tasks of the special `SIGSTOP` and `SIGCONT` signals). A Kernel Abstraction Layer (KAL) aims at abstracting the higher layers from the very low-level details of the Linux kernel, by providing a set of C functions and macros that abstract the needed kernel functionalities. The Resource Reservation layer (RRES) implements a variant of the CBS scheduling policy on top of an internal EDF scheduler. The QoS Manager layer allows applications to take advantage of Adaptive Reservations, and includes a set of bandwidth controllers that can be used to continuously adapt the budget of a vres according to what an application needs. An user-space library layer allows to extend standard Linux applications to use the AQuoSA functionality without any further restriction imposed on them by the architecture.

Thanks to an appropriately designed access control model [8], AQuoSA is available not only to the *root* user (as it happens for other real-time extensions to Linux), but also to non-privileged users, under a security policy that may be configured by the system administrator.

An interesting feature of the AQuoSA architecture is that it does not replace the default Linux scheduler, but coexists with it, giving to soft real-time tasks a higher priority than any non-real-time Linux task. Furthermore, the AQuoSA architecture follows a non-intrusive approach [3] by keeping at the bare minimum (the GSP patch) the modifications needed to the Linux kernel.

5.3 Bandwidth Inheritance Implementation

Design goals and choices The implementation of the BWI protocol for AQuoSA has been carried out with the following design objectives:

- to provide a full implementation of BWI;
- to allow for compile-time disabling of BWI;
- to allow the use of BWI on a per-mutex basis;
- to impact as low as possible on the AQuoSA code;
- to have as little as possible run-time overheads.

In order to achieve such goals, our implementation:

- uses the C pre-processor in order to allow compile-time inclusion or exclusion of the BWI support within AQuoSA;
- does not modify the Linux kernel patch (i.e., BWI is entirely implemented inside the kernel modules);
- does not modify the libraries and the APIs;
- does not remove or alter the core algorithms inside the framework, especially with respect to:
  - scheduling: it is not necessary to modify the implementation of various scheduling algorithms available inside AQuoSA;
  - vres queues: we do not modify the task queues handling, so that the old routines continue to work seamlessly;
  - blocking/unblocking: when BWI is not required/enabled the standard behaviour of AQuoSA is not modified by any means.

[3]For a more detailed description, the interested reader may refer to [17].

Using BWI Since BWI is the natural extension of PIP for RR-based systems, in our implementation the protocol is enforced every time two or more tasks, with scheduling guarantees provided through AQuoSA RR vres, also share a POSIX mutex that has been initialized with `PTHREAD_PRIO_INHERIT` as its protocol. This way the application is able to choose to use BWI or not on a per-mutex basis. Furthermore, all the code already using the Priority Inheritance Protocol automatically benefits of BandWidth Inheritance, if the tasks are attached to some vres.

Deadlock detection Once a deadlock situation is detected, the current implementation may be configured for realizing one of the following behaviors: 1) the system forbids the blocking task from blocking on the mutex, and returns an error (`EDEADLK`); 2) the system logs a warning message notifying that a deadlock took place.

5.4 Implementation Details

The BWI code is completely integrated inside the AQuoSA architecture and only entails very little modification to the following software components:

- the KAL layer, where the KAL API has been extended with macros and functions exploiting the in-kernel rt-mutexes functionality, for the purpose of providing, for each task blocked on an rt-mutex [4], the rt-mutex on which it is blocked on, and the task that owns such an rt-mutex;
- the RRES: where the two BWI rules are implemented.

The core of the BWI implementation is made up of a few changes to the AQuoSA data structures, and of only four main functions (plus a couple of utility ones):

1. `rres_bwi_attach()`, called when a task is attached to a vres;

[4]Note that such information is available inside the kernel only for rt-mutexes, for the purpose of implementing PIP.

2. `rres_bwi_block()`, called when a task blocks on an rt-mutex, that enforces the new BWI blocking rule (Sec. 4.1);
3. `rres_bwi_unblock()`, called when a task unblocks from an rt-mutex, that enforces the BWI unblocking rule;
4. `rres_bwi_detach()`, called when a task is detached from a vres.

The produced code can be found on the AQuoSA CVS repository, temporarily residing in a separate development branch. It will be merged soon in the very next releases of AQuoSA. It has been realized on top of the 2.6.21 kernel release and tested up to the 2.6.22 release.

In Tab. 1 the impact of the implementation on the source code of AQuoSA is briefly summarized.

	added	modified	removed
source files	2	0	0
lines of code	260	6	0

Table 1: Impact of our modification on AQuoSA sources.

6 Experimental evaluation

In this section we present some results of the experiments we ran on a real Linux system, with our modified version of AQuoSA installed and running, and with a synthetic workload provided by ad-hoc designed programs. These experiments are mainly aimed at highlighting features of the BWI protocol under particular blocking patterns, and gathering the corresponding overhead measurements.

6.1 Examples of execution

In the first example we have two tasks, τ_A and τ_B, sharing a mutex. τ_A has a computation time of $2\,msec$ and a period of $5\,msec$, and is given a reservation of $2\,msec$ every $5\,msec$. τ_B has a computation time of $6\,msec$ and a period of $15\,msec$, and is given a reservation of $2\,msec$ every $5\,msec$. In Fig. 2 we show the two schedules obtained with (bottom schedule) and without (top schedule) BWI.

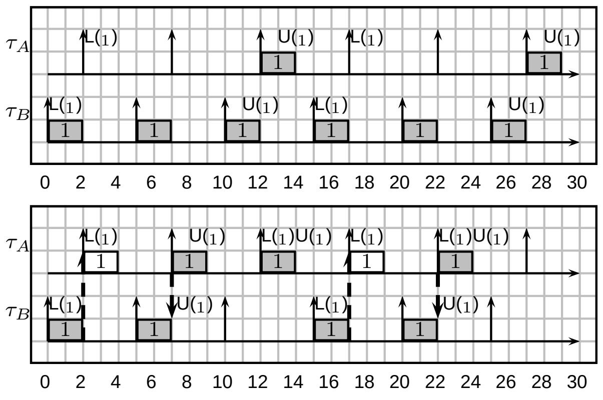

Figure 2: BWI effectiveness in reducing the tardiness

Notice that the use of BWI notably reduces the tardiness of both tasks, improving the system performance. This comes from the fact that τ_B always arrives first and grabs the lock on the shared mutex. If BWI is not used, then τ_A is able to lock the mutex and run only after τ_B ran for $2\,msec$ each $5\,msec$ time interval and completed its $6\,msec$ execution (at which instant it releases the lock). Thus, τ_A skips repeatedly the opportunity to run every two vres instances out of three: quite an outstanding waste of bandwidth. If BWI is in place, as soon as τ_A arrives and blocks on the mutex, τ_B is attached to its vres and completes execution much earlier, so that τ_A is now able to exploit two vres instances out of three, and the system does not waste any reserved bandwidth at all.

In Fig. 3 we show how the protocol is able to effectively enforce bandwidth isolation, by means of an example consisting of 5 tasks: τ_A and τ_C sharing m_1, τ_D and τ_E sharing m_2, and τ_B. Notice that τ_A does not share any mutex with τ_E. Also, τ_B has an earlier deadline than τ_C and τ_E, but a later one than τ_A, and this is a possible cause of priority inversion. When BWI is not used (top schedule), after τ_C and τ_E having locked m_1 and m_2 (respectively), they are both preempted by τ_B, and the inversion occurs. Furthermore, as a consequence of τ_E succeeding in locking m_2, since it has earlier deadline than τ_C, τ_A misses its deadline, which means bandwidth isolation is not preserved.

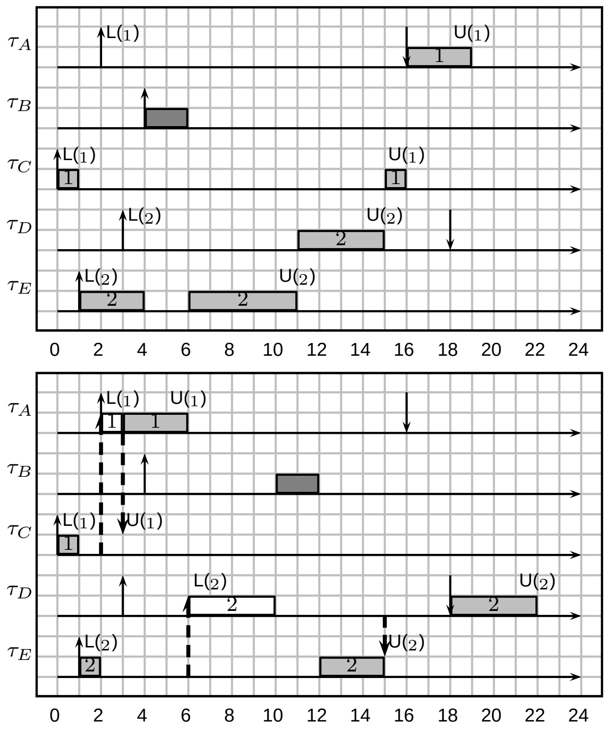

Figure 3: Example of BWI enforcing bandwidth isolation

On the other hand, when using BWI (bottom schedule), priority inversion is bounded, since τ_B is no longer able to preempt τ_C nor τ_E. Moreover, the behaviour of τ_E and τ_D can only affect themselves, between each others, and can no longer cause τ_A to miss its deadline, and this is the correct behaviour. Indeed, τ_D and τ_E are interacting tasks, and it is not possible to provide reciprocal isolation between them.

In the last example, in Fig. 5, we demonstrate the effect of BWI on bandwidth and throughput. We see (with no reclamation policy enabled) the protocol removes the waste of bandwidth due to blocking. In fact, while a task is blocked the bandwidth reserved for its vres can not be exploited by anyone else, if BWI is not in place. This is not the case if we use BWI, since when a task blocks its lock-owner is bound to such a vres and can consume the reserved bandwidth. Furthermore, thanks to our modification to the blocking rule (Sec. 4.1), this is also true in case of nested critical sections. For this example we used eight tasks, τ_A, τ_B, ..., τ_H. The mutexes are five with τ_A using m_0, m_1 and m_2; τ_B using m_2, m_3 and m_4; τ_C using m_1 and m_4; τ_D using m_1; τ_E using m_4; τ_F using m_2; τ_G using m_0; τ_H using m_0 too. Each task τ_i is bound to a vres with $U_i = 10/100$ (10% of CPU). The locking scheme is choose to be quite complex, in order to allow blocking on nested critical sections to occur. As an example of this in Fig. 4 the wait for graph at time $t = 40\,sec$, when all the tasks but τ_A are blocked, is depicted.

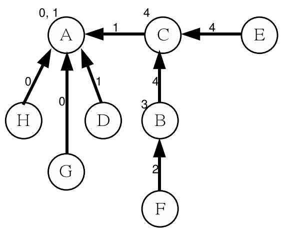

Figure 4: Wait-for graph for the example in Fig. 5. The numbers beside each task are the mutex(es) it owns. The number next to each edge is the mutex the task is waiting for.

Coming back to Fig. 5, the thick black line is the total bandwidth reserved, for each time instant t, for all the active vres. The thin black horizontal line represents the average value of the bandwidth. The thick gray line, instead, is the CPU the various running tasks are actually using and the thin gray line is its mean value. The thick black curve stepping down means a task terminated and its vres being destroyed, and so time values on the graphs are finishing times.

Comparing the two graphs it is evident that, when BWI is used (left part), 100% of the *reserved* CPU bandwidth is exploited by the running tasks, both instantaneously and on average. On the contrary, without BWI (right part) there exist many time instants during which the bandwidth that the running tasks are able to exploit is much less than what it has been reserved at that time, and the mean value is notably lower than the reserved one too. This means some reserved bandwidth is wasted. Finally, notice finishing times are are much smaller with BWI enabled.

Event	**Max. exec. (μsec)**		**Avg. exec. (μsec)**	
`blocking`	BWI unused	0	BWI unused	0.01
	used	1	used	0.169
`unblocking`	unused	0	unused	0.052
	used	3	used	0.116

Table 2: Max and mean execution times, with and without BWI

6.2 Overhead evaluation

We also evaluated the computational overhead introduced by our implementation. We ran the experiments described in the previous section on a desktop PC with 800 MHz *Intel(R) Centrino(TM)* CPU and 1GB RAM and measured mean and maximum times spent by AQuoSA in correspondence of task block and unblock event handlers, either when `PTHREAD_PRIO_INHERIT` was used and not.

BWI	**context switches #**	
unused	task τ_A	26
	task τ_B	34
used	task τ_A	25
	task τ_B	34

Table 3: context switch number with and without BWI using periodic sleeping tasks

Tab. 2 shows the difference between the measured values with respect to the ones obtained when running the original, unmodified version of AQuoSA (average values of all the different runs) [5].

BWI	**context switches #**	
unused	task τ_A	607
	task τ_B	414
	task τ_C	405
used	task τ_A	343
	task τ_B	316
	task τ_C	405

Table 4: Context switch number with and without BWI using greedy tasks.

As we can easily see, the introduced overhead is negligible for tasks not using the protocol. Anyway, also when BWI is used, the overhead is in the order of one tenth of microsecond, and this is definitely an acceptable result.

With respect to context switches, we see in Tab. 3 that the protocol has practically no effect if typical real-time tasks, with periodic behaviour, are considered.

On the contrary, if "greedy" tasks (i.e., tasks always running without periodic blocks) are used, Tab. 4 shows that the number of context switches they experience is dramatically smaller when using BWI. This is due to the bonus bandwidth each task gets thanks to the protocol.

7 Conclusions and Future Work

In this paper, we presented an improved version of the BWI protocol, an effective solution for handling critical sec-

[5]Note that, in the latter case, the use of `PTHREAD_PRIO_INHERIT` by tasks implies the use of the original PIP protocol, not the BWI one.

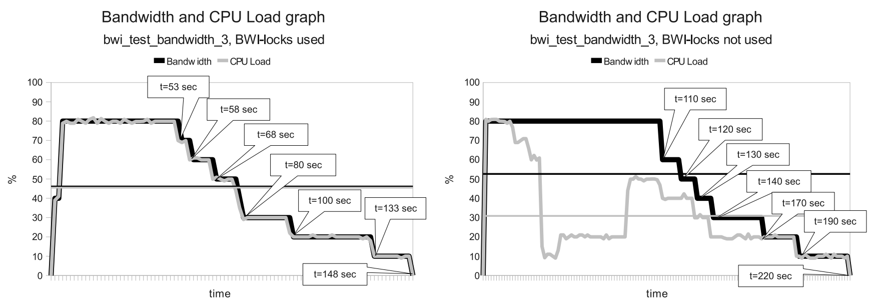

Figure 5: Resource usage with BWI

tions in resource reservation based systems. We also proposed an implementation of the protocol inside AQuoSA, a reservation framework working on Linux. Finally, we ran some experiments in order to evaluate the overhead the protocol introduces when used on such a concrete system.

Our modifications improve correctness and predictability of BWI, enable deadlock detection capabilities and enforce better handling of nested critical sections. The implementation is lean, simple and compact, with practically no need of modifying the framework core algorithms and structures. The experimental results show this implementation of BWI is effective in allowing resource sharing and task synchronization in a real reservation based system, and also has negligible overhead.

Regarding future works, we are investigating how to integrate ceiling like mechanisms inside the protocol and the implementation, in order to better deal with the problem of deadlock, so that we can prevent instead of only check for it. Work is also in progress to modify a real multimedia application so that it will use the AQuoSA framework and the BWI protocol. This way we will be able to show if our implementation is useful also inside real world applications with their own blocking schemes.

Other possible future works include the investigation of more general theoretical formulation to extend the RR methodologies and the BWI protocol to multiprocessor systems. Also, it would be interesting to adapt the AQuoSA framework to the `PREEMPT_RT` kernel source tree, so to benefit from its interesting real-time features, especially the general replacement, within the kernel, of classical mutexes with rt-enabled ones.

References

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

[2] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. IEEE Real-Time Systems Symposium*, pages 4–13, Madrid, Spain, Dec. 1998.

[3] L. Abeni and G. Lipari. Implementing resource reservations in linux. In *Real-Time Linux Workshop*, Boston (MA), Dec. 2002.

[4] T. P. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems*, (3), 1991.

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

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

[7] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971. ISSN 0360-0300.

[8] T. Cucinotta. Access control for adaptive reservations on multi-user systems. In *Proc.* 14^{th} *IEEE Real-Time and Embedded Technology and Applications Symposium (to appear)*, April 2008.

[9] Z. Deng and J. W. S. Liu. Scheduling real-time applications in open environment. In *Proc. IEEE Real-Time Systems Symposium*, Dec. 1997.

[10] U. Drepper and I. Molnar. The native posix thread library for linux. Technical report, Red Hat Inc., February 2001.

[11] H. Franke, R. Russel, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Ottawa Linux Symposium*, 2002.

[12] T. M. Ghazalie and T. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9, 1995.

[13] G. Lipari and C. Scordino. Current approaches and future opportinities. In *International Congress ANIPLA 2006*. ANIPLA, November 2006.

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

[15] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, Jan. 1973.

[16] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: An Abstraction for Managing Processor Usage. In *Proc. 4th Workshop on Workstation Operating Systems*, 1993.

[17] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. AQuoSA – adaptive quality of service architecture. *Software: Practice and Experience*, on-line early view, 2008.

[18] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems. In *Proc. Conf. on Multimedia Computing and Networking*, January 1998.

[19] R. R. Rajkumar, L. Abeni, D. de Niz, S. Ghosh, A. Miyoshi, and S. Saewong. Recent Developments with Linux/RK. In *Proc. 2nd Real-Time Linux Workshop*, Orlando, Florida, november 2000.

[20] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), September 1990.

[21] U. Steinberg, J. Wolter, and H. Hartig. Fast component interaction for real-time systems. In *Proc.* 17^{th} *Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 89–97, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2400-1.