Priority Inheritance on Condition Variables

Tommaso Cucinotta
Bell Laboratories, Alcatel-Lucent Ireland
Email: tommaso.cucinotta@alcatel-lucent.com

***Abstract*—In this paper, a mechanism is presented to deal with *priority inversion* in real-time systems when multiple threads of execution synchronize with each other by means of mutual exclusion semaphores coupled with the programming abstraction of *condition variables*. Traditional priority inheritance solutions focus on addressing priority or deadline inversion as due to the attempt to lock mutual exclusion semaphores, or deal exclusively with specific interaction patterns such as client-server ones. The mechanism proposed in this paper allows the programmer to explicitly declare to the run-time environment what tasks are able to perform a notify operation on a condition over which other tasks may be suspended through a wait operation. This enables developers of custom interaction models for real-time tasks to exploit their knowledge of the application-specific interaction so as to potentially reduce priority inversion. The paper discusses issues in the realization of the technique, and its integration with existing priority inheritance mechanisms on current operating systems. Also, the paper briefly presents the prototyping of the technique within the open-source RTSim real-time systems simulator, which is used to highlight the potential advantages of the exposed technique through a simple simulated scenario.**

I. Introduction and problem presentation

Priority inversion is a well-known problem in the literature of real-time systems occurring every time a task execution is delayed due to the interference of lower priority tasks. This problem is well-known to happen whenever a higher-priority task tries to acquire a lock on a mutual-exclusion semaphore (shortly, a *mutex*) already locked by a lower-priority task. Clearly, the lock owner task needs to release the lock before the more urgent one can proceed and this delay is unavoidable. However, if a third task with a middle priority between these two is allowed to preempt the lower-priority task holding the lock, then the release of the lock is delayed even further, adding an unnecessary delay to the execution of the higher-priority task, waiting for the lock to be released.

For example, Figure 1 shows a sequence in which three tasks, A, B and C are scheduled on the same processor by using a fixed-priority scheduler, and A and C synchronize on a mutex M for the access to some common data structure. Task C runs while no other higher-priority task is ready to run. Then, it locks the mutex (operation L(M) in the picture) but, before being able to release it (operation U(M) in the picture), it is preempted by the higher-priority task A that just woke up. Task A executes for a while, then it tries to lock the same mutex already locked by C, thus it suspends allowing C to continue execution. Unfortunately, C cannot execute for much time, because the middle-priority task B wakes up at this point, preempting C as due to its higher-priority. Even though B has a higher-priority than C, we know that C holds

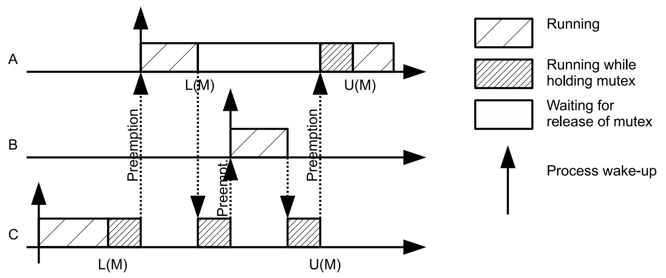


Figure 1. Sample priority inversion scenario. Task A has the highest priority, task C the lowest, and task B has a middle priority between them.

a lock for which the highest-priority task A is waiting, thus B should not be allowed to preempt C in such a case. Therefore, the time for which B keeps executing, delaying the release of the lock by C, constitutes an avoidable additional delay for the task A.

This problem has been addressed in a number of ways in the literature, for example by the well-known Basic Priority Inheritance and Priority Ceiling mechanisms [1], [2]. The related literature is discussed in Section II later.

A mutex is commonly used for synchronization of tasks in conjunction with the *condition variables* programming abstraction, a mechanism that allows a task to suspend its execution waiting for a condition to be satisfied on certain variables. The typical example is the one of a reader from a queue of messages waiting for someone to write into the queue when it is empty. When the reader tries to read an element from the queue but finds it empty, it suspends itself till the number of elements in the queue becomes greater than zero (as a consequence of a writer task pushing one element). In such a case, the reader typically blocks on a condition variable with an operation that atomically suspends the task and releases the mutex (e.g., by using the POSIX [3], [4] `pthread_cond_wait()` call) used for critical sections operating on the queue. The writer, on the other hand, after insertion of an element in the queue, notifies possible readers through a notify operation (e.g., the POSIX `pthread_cond_notify()` call).

In such cases, a form of unnecessary priority inversion may still occur. Consider for example the scenario depicted in Figure 2. Task A communicates with a set of other tasks C, D and E via a message queue Q, through which it expects to receive some message from them. Now, imagine that, for whatever reason, in the set of tasks potentially producing messages for A, there is also a task C with a priority lower than the one of A. In the shown scenario, A tries to read atomically from the queue (operation R(Q) in the picture), thus it locks the queue mutex, but releases it immediately after detecting that the queue is empty, blocking on a condition variable. Task C then executes, but, before it finishes its computations for writing into the message queue Q, it gets preempted by a third unrelated task B with a priority level higher than B but lower than A. The time for which B runs constitutes an unnecessary delay for the execution of the higher-priority task A, which is waiting for B to write something into the shared queue.

With the Priority Inheritance protocol, whenever a task blocks trying to acquire a lock on a mutex that is already locked by another task, the former task can temporarily donate its priority to the latter task, in order to let it progress quicker (avoiding unneeded preemptions) towards releasing the lock. However, for the scenario depicted in Figure 2, even if concurrent access to the queue Q is protected by a mutex exhibiting Priority Inheritance, the mechanism cannot help. Indeed, in this case Task C is not holding the lock of the Q mutex while it is progressing towards completing the computations that will lead to the production of a data item to be pushed into the message queue Q. Only as part of the atomic push and pop operations into and from the queue, does a task acquire the mutex lock protecting access to the queue. Therefore, Priority Inversion can merely fire in the short time instants of execution of the atomic operations, but not during the generally longer execution of the tasks.

Abstracting away from the specific example of shared message queues, generally speaking, consider a set of real-time tasks synchronizing through the use of mutex and condition variables. Then, if a task that needs to wait for a condition to become true may be unnecessarily delayed by lower-priority tasks, then a form of priority inversion can occur. Indeed, if the task(s) responsible for letting the mentioned condition become true run(s) at a lower-priority in the system, and a third task with a middle priority level wakes up, said third task may preempt the execution of those lower-priority tasks, thus delaying the achievement of the condition for which the higher-priority task is waiting. In this case, the traditional mechanism of Basic Priority Inheritance cannot help, because the higher-priority task waiting for the condition to become true drops the mutex lock before suspending through a wait operation on the condition variable, and the lower-priority task(s) that need to progress in their computations so as to perform the notify operation on the condition variable do not hold any mutex lock while they are computing. Furthermore, the run-time has generally no information about the (application-specific) interaction among the tasks, so it cannot infer automatically the needed task dependency information.

In some cases, real-time tasks might interact through higher-level mechanisms that allow the run-time to actually know, when a task suspends, which other tasks may actually cause the resume of the suspended task. For example, this is the case of the client-server interaction model of the Ada language run-time, in which a client task invokes explicitly an Entry of a server task, i.e., it pushes an element into the server

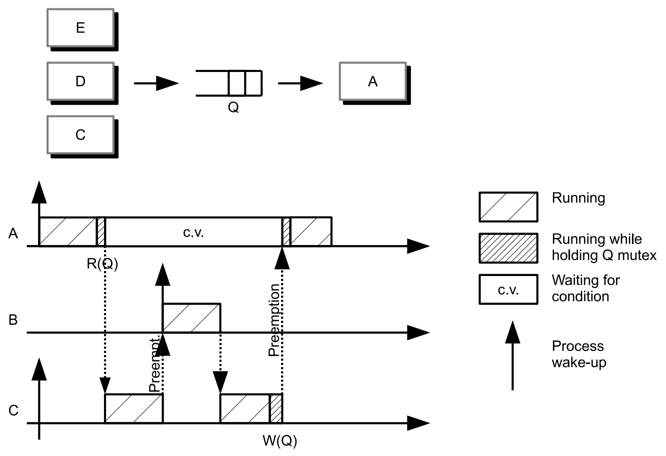


Figure 2. Priority inversion scenario with task A receiving data from a lower-priority task C through a shared message-queue Q realized with a mutex and a condition variable. Task B has middle-priority between A and C.

input queue and suspends till it receives a response. In such a case, the run-time knows which particular server has to perform work on behalf of which client(s), so it can correctly apply Priority Inheritance, as shown in the seminal work on the topic by Sha et al. [5]. However, in the general case of tasks interacting by application-specific synchronization protocols realized through mutex and condition variables, such information is not readily available to the run-time.

A. Paper contribution

In this paper, a mechanism is proposed to let the run-time be aware of the possible dependencies among tasks within a real-time system, expanding the functionality of the programming abstraction of condition variables. With the new mechanism, called PI-CV, the programmer may declare what are the tasks that may help a condition become true, over which other tasks may be waiting. Exploiting such dependency information, the run-time can trigger the necessary priority inheritance that is needed to avoid priority inversion. PI-CV alleviates the problem of priority inversion in cases in which developers code into the system custom, application-specific communication and synchronization logic through mutex and condition variables.

There are various scenarios in which the introduced mechanism may be useful and indeed improve responsiveness of real-time software components. For example, in the literature of real-time systems, it is very common to see real-time applications modeled as Directed Acyclic Graphs (DAGs) of computations which are triggered periodically or as a result of external events. Each node in the DAG can start its computations once its inputs are available (see Figure 3), which in turn are produced as output of the computations of other nodes. The mechanism is particularly useful in contexts in which producers and consumers of data share common data structures in shared memory (serializing the operations on it by means of semaphores and synchronizing among each other by means of condition variables), but at the same time they possess different priority or criticality. This situation is very common in real-time systems. For example, we can easily find co-existence of both the main real-time code, characterized by stringent timing constraints, and other side software components that are needed for monitoring or configuration purposes. Often it happens that some (often bidirectional) information flow is needed between these two worlds (e.g., the monitoring code needs to retrieve information about the status of the real-time code, and the real-time code needs to reconfigure itself according to the configuration passed by reconfiguration code).

II. Related work

The literature on the management of shared resources for real-time systems is huge. In this section, the main works related to the problem of priority inversion are shortly recalled.

During the International Workshop on Real-Time Ada Issues, back in 1987, Cornhill and Sha reported [6] various limitations of the Ada language when dealing with priority-based real-time systems. Specifically, a high-priority task could be delayed indefinitely by lower priority tasks under certain conditions. Shortly afterward, the same authors formalized [7] what are the correct interactions between client and server tasks in form of assertions on the program execution. The Ada run-time was not respecting those assertions, thus allowing tasks to undergo unnecessary priority inversion. In the same work, Priority Inheritance was informally introduced as a general mechanism for bounding priority inversion. Later, Sha et al. [2], [1] described better their idea formalizing the two well-known Basic Priority Inheritance (BPI) and Priority Ceiling (PCP) protocols. While BPI allows a task to be blocked multiple times by lower priority tasks, with PCP a task can be blocked at most once by lower-priority tasks, so priority inversion is bounded by the execution time of the longest critical-section of lower-priority tasks; also, PCP prevents deadlock. A possible realization of PCP for the Ada language has been described by Goodenough and Sha [8], and by Borger and Rajkumar [9]. Also, Locke and Goodenough discussed [10] some practical issues in applying PCP to concrete real-time systems.

Various extensions to PCP have been proposed, for example to deal with reader-writer locks [11], multi-processor systems [12], [13], [14] and dynamically recomputed priority ceilings [15]. Furthermore, Baker introduced [16] Stack Resource Policy (SRP), extending PCP so as to handle multi-unit resources, dynamic priority schemes (e.g., EDF), and task groups sharing a single stack ("featherweight" processes), treated on its own as a resource with a zero ceiling. Also, Gai et al. investigated [17] on minimizing memory utilization when sharing resources in multiprocessor systems. More recently, Lakshmanan et al. [18] further extended PCP for multi-processors grouping tasks that access a common shared resource and co-locating them on the same processor.

Schmidt et al. investigated [19] on various priority inversion issues in CORBA middleware, and proposed an architecture (TAO) not suffering of such problem. Priority inversion has also been considered by Di Natale et al. in a proposal [20] for schedulability analysis of real-time distributed applications, where, despite the use of PCP for scheduling tasks on the CPUs, non-preemptability of packet transmissions causes unavoidable priority inversion when a higher-priority packet reaches the transmission queue while a low-priority packet is being transmitted.

When scheduling under the Constant Bandwidth Server (CBS) [21], Lamastra et al. proposed [22], [23] the BandWidth Inheritance (BWI) protocol, allowing a task owning a lock on a mutex not only to inherit the (dynamic) priority of the highest priority waiting task (if higher than its own), but also to account for its execution within the reservation of the task whose priority is being inherited. This allows to keep the *temporal isolation* property ensured by the CBS, in the sense that non-interacting task groups cannot interfere on each other's ability to meet their timing constraints. Later, Faggioli et al. [24] discussed various issues and optimizations in the implementation of the protocol in the Linux kernel, and specifically as an add-on to the AQuoSA scheduler [25]. An extension of BWI to multi-processor systems has been proposed again by Faggioli et al. [26], where the implementation of the technique [27] was prototyped this time on the LITMUS-RT [28] real-time test-bed.

Block et al. proposed FMLP [29], a resource locking protocol for multi-processor systems allowing for unrestricted critical-section nesting and efficient handling of the common case of short non-nested accesses.

Guan et al. dealt [30] with real-time task sets where interactions among tasks are only known at run-time depending on which particular branches are actually executed.

Many other works exist in the literature [31], [32], [33], [34], [35], [36], [37] on variants of the above resource-sharing protocols and their analysis. An overview of them can be found in [27]. Recently, techniques to mitigate priority inversion have also been applied in the context of scheduling virtual machines communicating with each other [38]. A very interesting recent work by Abeni and Manica [39] adapts BWI to trigger priority inheritance on client-server interactions, and presents a schedulability analysis for that particular type of scenario. The mechanism being presented in this paper is more generic as it can be used with custom inter-thread communications. Though, the analysis presented by Abeni may constitute a valuable starting point for the analysis of the generic scenarios addressed by the present paper.

The above reviewed literature on resource sharing in real-time systems focuses essentially on dealing with priority inversion (and applying various types of priority inheritance mechanisms) in two main scenarios: 1) tasks interacting by the use of shared memory and critical sections, serialized through mutexes; 2) tasks interacting in a client-server fashion, where the server task executes operations on behalf of various clients.

In this paper, a general priority inheritance mechanism is presented, useful when tasks interact by using condition variables associated with mutexes. These are generally used in the implementation of custom shared data types supporting custom communication and synchronization protocols in concurrent systems. In such a case, when a task, after entering a critical section, suspends itself through a wait operation on a condition variable, it also releases the mutex associated with the critical section. At this point, some other task running in the system may be the one responsible for the notify operation on the same condition variable, waking up the task(s) suspended on it. However, without further information, the run-time cannot generally know which task(s), among the currently ready-to-run ones, may perform such a notify operation. If the interacting tasks have different priorities, then the system may undergo avoidable priority inversion. With the mechanism proposed in this paper, the run-time is informed by the tasks about which other tasks may possibly help and accelerate the wake-up of a task suspended on a condition variable, thus enabling the avoidance of this kind of priority inversion. The mechanism can also be composed with existing priority inheritance schemes for lock-based interactions. Furthermore, it can also be used for realizing priority inheritance in client-server interactions, in Ada-rendezvous style. However, it can also be used in arbitrary, application-specific interactions programmed through mutual exclusion semaphores and condition variables.

Note that Hart and Guniguntala [40] made changes to the GNU `libc` pthreads library and kernel in order to support efficient wake-up of multiple tasks waiting on a condition variable (as due to a `pthreads_cond_broadcast()`) used in connection with an rt-mutex, so as to avoid the "thundering herd" effect, and guaranteeing the correct wake-up order (considering also priority inheritance). Such changes relate to the support for priority inheritance in rt-mutexes and they are not to be confused with the mechanism being proposed in this paper.

To the best of my knowledge, there are no alternatives for dealing with the specific type of problem of priority inversion as described above, in presence of condition variables. Commonly known alternatives to using semaphores and locks at all, include recurring to lock-free data structures, and solutions based on the Transactional Memory programming paradigm [41]. Lock-free programming is well-known to be more complex and difficult to master, than traditional lock-based programming. The advantage of the presented technique is that it allows applications developers to keep designing code using traditional synchronization primitives, i.e., mutual exclusion semaphores and condition variables, but they can improve the responsiveness of their applications with the very little additional effort to sort out which are the helper tasks for the condition variables they use (or, sometimes, the helper tasks may be automatically identified in proper libraries, see Section IV-B later). On the other hand, the Transactional Memory programming paradigm is particularly useful in presence of non-blocking operations on shared data structures, i.e., operations that would not lead to the suspension of the calling task in order to wait for a condition to become true, as it happens with condition variables, thus it does not constitute an alternative to the presented technique. A thorough and detailed comparison among these communication and synchronization techniques is outside the scope of this paper.

III. Priority inheritance on condition variables

In what follows, without loss of generality, the term *task* will be used to refer to a single thread of execution in a computing system, being it either a thread in a multi-threaded application, or a process. Also, without loss of generality, the term *priority* will be used to refer to the right of execution (or "urgency" level) of a task as compared to other tasks from the CPU(s) scheduler viewpoint. This includes both the priority of tasks whenever they are scheduled according to a priority-based discipline and their deadline whenever they are scheduled according to a deadline-based discipline (and their time-stamp whenever they are scheduled according to other policies based for example on virtual times, such as the Linux CFS [42]). However, the described technique is not specifically tied to these scheduling disciplines and it can be applied in presence of other schedulers as well. Furthermore, it should be clarified that this paper deals with how to dynamically change the priorities of tasks within a system, which is orthogonal with respect to how said tasks are scheduled on the available processors. Specifically, analyzing the consequences of the introduced technique on schedulability of real-time systems in presence of multi-core and/or multi-processor systems is out of the scope of this paper.

The mechanism of priority inheritance on condition variables (PI-CV) proposed in this paper works as follows:

- it is possible (but not mandatory) to programmatically associate a condition variable with the set of tasks able to speed-up the verification of the condition; these tasks will be called *helper tasks*; the set of helper tasks associated with a condition variable can be fixed throughout the lifetime of the condition variable, or be dynamically changed at run-time, according to the application needs;
- whenever a higher-priority task executes a wait operation on a condition variable, having a non-null set of helper tasks, *it temporarily donates its priority to all the lower-priority helper tasks*, so as to "speed-up" the verification of the condition associated with the condition variable;
- *as soon as the condition variable is notified, the dynamically inherited priority is revoked*, restoring the original priority of the helper tasks;
- *the mechanism can be applied transitively*, if one or more helper tasks suspends on other condition variables;
- *the mechanism can be nicely integrated with traditional (Basic) Priority Inheritance*, resulting in priority being (transitively) inherited from a higher priority task to a lower priority one either because the former waits to acquire a lock held by the latter, or because the former suspended through a wait operation on a condition variable for which the latter is a helper task.

Whenever a higher-priority task is suspended waiting for some output produced by lower-priority tasks, PI-CV allows the lower-priority tasks to temporarily inherit the right of execution of the higher-priority task with respect to the tasks scheduler. In order for the mechanism to work, it is necessary to introduce a few interface modifications to the classical

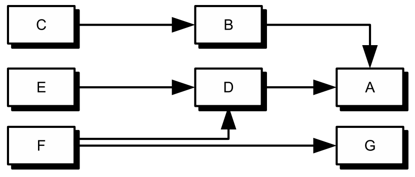


Figure 3. General interaction scenario where priority inheritance on condition variables may be applied transitively. Task F is waiting on a condition variable having tasks D and G registered as helpers.

condition variables mechanism as known in the literature, so that the run-time environment (e.g., the Operating System) knows which lower-priority tasks should inherit the priority of a higher-priority task suspending its execution waiting for a condition to become true. The interface may allow the mechanism of priority inheritance on condition variables to be enabled selectively on a case-by-case basis (per-condition variable and per-semaphore), depending on the application and system requirements (see below).

Priority inheritance may be applied transitively, when needed. For example, if Task A blocks on a condition variable donating temporarily its priority to Task B, and Task B in turn blocks on another condition variable donating temporarily its priority to Task C, then Task C should inherit the highest priority among the one associated with all the 3 tasks. Also, priority inheritance for condition variables can be integrated with traditional priority inheritance (or deadline inheritance) as available on current Operating Systems, letting the priority transitively propagate either due to an attempt of locking a locked mutex, or to a suspension on a condition variable with associated one or more helper tasks.

In other words, consider a blocking chain of tasks $(\tau_1, \tau_2, \ldots, \tau_n)$ where each task τ_i $(1 \leq i \leq n-1)$ suspended on the next one τ_{i+1} either trying to acquire a lock (enhanced with priority or deadline inheritance) already held by τ_{i+1}, or waiting on a condition variable (enhanced with PI-CV as described in this document) where τ_{i+1} is registered among the helper tasks. All the tasks in such a blocking chain are suspended, except the last one (that is eligible to run). This last task inherits the priority of any of the tasks in any blocking chain terminating on it, i.e., any task in the direct acyclic graph of blocking chains that terminate on it. For example, consider the scenario shown in Figure 3, where each arrow from a task to another means that the former is suspended on the latter due to either a blocking lock operation or a wait on a condition variable where the latter task is one of the helpers. Task A inherits the highest priority among tasks B, C, D, E, F, while G inherits the priority of F, if all of the suspensions happen through mutex semaphores enriched with priority inheritance or condition variables enriched with PI-CV. In the depicted scenario, note that F is waiting on a condition variable where both D and G are registered as helpers. This allows both of them to inherit the priority of F, until the condition is notified.

A. *Reservation-based scheduling*

Also, whenever a task is associated by the scheduler with a maximum time for which it may execute within certain time intervals, as in reservation-based scheduling [43], [44] (e.g., the POSIX Sporadic Server [3] or the CBS), the inheritance mechanism may behave in such a way that the helper task executing as a result of its priority having been boosted by the described priority inheritance mechanism, will account its execution towards the execution-time constraints of the task from which the priority was inherited (i.e., the budget of its server). For example, referring to the Bandwidth Inheritance (BWI) protocol [22], it is straightforward to think of the corresponding extension. In a BWI-CV protocol, whenever a task inherits the priority of a higher-priority task, the ready-to-run tasks at the end of the blocking chains (involving both attempts to acquire locks and wait operations on condition variables with associated other helper tasks) also execute in the server of the highest-priority task that is donating its priority to them, depleting its corresponding budget. Namely, the server to consider for budget accounting purposes should be the one associated with the highest-priority task, among the ones in the Direct Acyclic Graph (DAG) of all the blocking chains terminating on the said ready-to-run task.

B. *Multi-processor systems*

Note that PI-CV can be applied to single-processor as well as to multi-processor and multi-core systems. PI-CV merely allows the programmer to declare which are the helper tasks for each given condition variable at each time throughout the program life-time, and the run-time applies priority inheritance as described above. The specifics about how exactly tasks are scheduled in a multi-processor environment are outside the scope of this paper.

C. *Schedulability analysis*

PI-CV is presented in this paper without any particular associated schedulability analysis technique nor formal proof. As the mechanism allows for reducing priority inversion, it is expectable that the worst-case and/or average-case interference terms in schedulability analysis calculations, as coming out considering the specifics of the scheduling policy being employed on a system, have a shorter duration. This is shown by simulation in a simple scenario later in Section VI.

Similarly to the traditional Priority Inheritance mechanism available on current Operating Systems, PI-CV may reduce unneeded priority inversion in certain scenarios, leading to an improved responsiveness of the highest priority activities within a system. Also, when combined with resource reservations along the lines of BWI [22], [23], a BWI-CV mechanism should be capable of guaranteeing temporal isolation among non-interacting task groups. However, a theoretical analysis would be useful to provide a strong assessment on the (worst-case) responsiveness of the various real-time activities, including understanding whether it will be possible to meet all deadlines for higher-priority tasks that may benefit from PI-CV, as well as for lower-priority ones that may worsen their behavior, in presence of interactions based on condition variables. Further development of these concepts is left as future work.

IV. Implementation notes

From an implementation standpoint, the proposed mechanism may be made available to applications via a specialized library call that can be used by a task to declare which other tasks are the potential helpers towards the verification of the condition associated with a condition variable. For example, in an implementation leveraging the pthreads library implementation, this can be realized through the following C library calls:

```
int pthread_cond_helpers_add
  (pthread_cond_t *cond, pthread_t *helper);
int pthread_cond_helpers_del
  (pthread_cond_t *cond, pthread_t *helper);
```

These two functions add or delete the `helper` thread to the pool of threads (empty after a `pthread_cond_init()` call) that can potentially inherit the priority of any thread waiting on the condition variable `cond` by means of a `pthread_cond_wait()` or `pthread_cond_timedwait()` call. The condition variable may be associated with a list of helper threads, and a kernel-level modification needs to ensure that the highest priority among the ones of all the waiters blocked on the condition variable is dynamically inherited by the registered helper thread(s), whenever higher than their own priority (and also that this inheritance is transitively propagated across both condition variables and traditional mutex supporting Priority Inheritance). Whenever the `pthread_cond_notify()` or `pthread_cond_broadcast()` function is called, the correspondingly woken-up thread(s) will revoke donation of their own priority.

A. Message queues

In a possible usage scenario, the proposed mechanism can be associated with a message queue in shared memory protected by a mutex for guaranteeing atomic operations on the queue, and a condition variable used to wait for the queue to become non-empty (if the queue has a predetermined maximum size, then another condition variable may similarly be used to wait for the queue to become non-full). In such a scenario, whenever initializing the condition variable, a writer task will declare itself as a writer associating its `pthread_t` to the condition variable, i.e., declaring explicitly that its execution will lead to the verification of the condition associated to that condition variable (non-empty queue). This can be done with a call to the above introduced `pthread_cond_helpers_add()` function after the condition variable initialization. Therefore, whenever a reader task will suspend its execution via a `pthread_cond_wait()` call on the condition variable, the associated writer(s), if there are any of them ready for execution, will dynamically inherit the priority of the suspended reader if higher than their own priority. This will inhibit third unrelated middle-priority tasks to preempt the low-priority writers, protecting from the mentioned Priority Inversion problem.

In a possible scenario in which there is a pipeline of multiple tasks using the just mentioned PI-CV-enhanced message queue implementation, it is possible to see the transitive inheritance propagation. Consider, for example, the scenario depicted in Figure 4, where A receives data from B through a message queue Q2, and B receives data from C through another message queue Q1. In such a case, when A attempts a read from Q1 but it suspends because it finds the queue empty, its priority may be donated to B. However, if B suspends on its own because it attempts a read from Q2 but it finds it empty, then C inherits not only the priority of B, but also the one of A (i.e., C runs with the highest priority – be it priority or deadline or other type of time-stamp – among A, B and C).

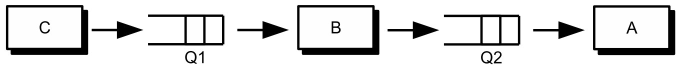


Figure 4. Pipeline interaction model.

B. Client-server interactions

The described PI-CV mechanism may be leveraged to realize client-server interactions with the correct management of priorities whenever a server executes on behalf of a client with possibly other clients waiting for its service(s). In a possible implementation, clients and servers interact through message queues, synchronized through mutexes and condition variables. A server accepts requests from clients through a single server request queue. Each client may receive the desired reply from the server through a dedicated client-server reply queue. Each client may explicitly declare the server as the helper task for the condition variable associated to the client-server reply queue being non-empty. After posting a new request in the server request queue, a client suspends on the condition variable of its dedicated client-server reply queue. This allows the OS to automatically let the server inherit the maximum priority among (its own priority and) the priorities of any client waiting for its service(s).

Also, if the mutex protecting the message queues are all enhanced with traditional priority inheritance (e.g., the POSIX `PTHREAD_PRIO_INHERIT` attribute), the two mechanisms compose with each other towards reducing priority inversion.

Note that, in such scenario, it would be easy to provide a proper programming abstraction for client-server messaging that declares *implicitly* which are the helper tasks for the condition variables of the dedicated reply message queues. When dealing with real-time scheduling and the correct set-up of scheduling parameters, it is often convenient for developers if the Operating System or middleware services exhibit self-tuning capabilities [45], [46].

Effectiveness of PI-CV in the context of client-server interactions is further explored in Section VI, reporting a few simulation results. These have been obtained by means of the implementation of PI-CV described in what follows.

V. Simulation

The described PI-CV mechanism has been prototyped within the open-source RTSim real-time systems simulator[1]. RTSim [47] allows for simulation of a multi-processor system running a set of real-time tasks. Various scheduling mechanisms are available within the framework, including fixed priority, deadline-based scheduling and resource-reservation mechanisms [43], [44] (e.g., the POSIX Sporadic Server [3] or the Constant Bandwidth Server [21]). The simulated real-time tasks can be programmed with a simple language that includes, among others, instructions for simulating:

- computations for a fixed amount of time units (`fixed()` instruction), or for a probabilistically distributed time;
- basic locking instructions (`lock(M)` and `unlock(M)`) allowing for simulations of critical sections protected by a mutex `M`, corresponding to the POSIX `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions.

The simulator also includes simulation of a few protocols for shared resources, which can be associated with the locking primitives, such as priority ceiling, traditional priority inheritance on mutexes, BWI [22] and others.

RTSim has been extended with the following modifications:

- condition variables have been supported through a new `CondVar` object type that can be referenced in two new dedicated task statements: `wait(M,CV)` and `signal(M,CV)`, which act on the mutex `M` and CondVar `CV`, with semantics corresponding to the POSIX `pthread_cond_wait()` and `pthread_cond_signal()` function calls, respectively (note that, when a task is suspended via `wait()`, the mutex `M` is released, and that `signal()` wakes up only the highest-priority task among the waiters);
- a new `Counter` object type has been added, with the associated instructions `inc()`, `dec()` and `set()`, with obvious meaning;
- as RTSim lacks of conditional statements, a new `waitc(M,CV,SZ)` instruction has been added which suspends the calling task performing a `wait()` only if the specified counter is zero;
- the support for priority inheritance has been completed for the case of arbitrarily nested critical sections;
- the PI-CV mechanism as described above has been integrated, in a way that also integrates transitive inheritance among mutex lock and condition variable wait primitives.

Helper tasks for condition variables must be set-up statically before the simulation begins. Furthermore, the PI-CV and traditional priority inheritance mechanisms can be individually enabled or disabled for the whole simulation.

With such elements, it is possible to simulate for example the synchronization among two tasks due to one of them writing onto a shared queue and the other one waiting for reception of a message, using a counter `SZ` just to keep track of the queue size. To simulate a periodic or sporadic writer that computes for 6 time units, then it pushes a message onto a queue with an atomic operation lasting for 2 time units, then it waits for the next cycle, one would use the code in Figure 5 left. To simulate a reader/waiter that computes for 2 time units, then it waits for a message to be made available onto the queue, where checking the message availability takes 1 time unit, and extracting it from the queue takes 2 time units, then it completes the cycle with further 3 time units of computation, one would use the code in Figure 5 right.

```
                      fixed(2);
fixed(6);             lock(M);
lock(M);                fixed(1);
  fixed(2);             waitc(M,CV,SZ);
  inc(SZ);              fixed(2);
unlock(M);              dec(SZ);
signal(M,CV);         unlock(M);
                      fixed(3);
```

Figure 5. Task code for send (left) and receive (right) via a shared queue.

The above scheme can be used, for example, to reproduce the scenario in Figure 2.

Additional instructions have been introduced to deal with more dynamic behaviors, as required in a real client-server interaction, in which the server cannot know in advance what client it will receive a request from, thus it cannot know in advance which client queue it will have to push the answer into. To this purpose, the following further modifications have been realized in RTSim:

- a new `Queue` type has been added, abstracting a message queue functionality, in which no messages are actually exchanged by tasks, but RTSim remembers how many messages have been posted by means of the usual `push(Q)` and `pop(Q)` operations; also, the further operations `pushptr(Q,RQ,RCV,RM)` and `popptr(Q,P_RQ,P_RCV,P_RM)` are used for more complex client-server interactions, where a client can post into the queue information on which queue the reply should be directed to (see code below); the `Queue` type is purposely non-synchronized, so as to leave freedom to specify the synchronization by composing the other primitives as needed;
- a new `waitq(M,CV,Q)` operation waits for the specified queue to be non-empty, performing a `wait()` operation on the specified CV and releasing the specified mutex, if needed;
- a new `Pointer` type has been added, capable of pointing to mutex, condition variable and queue objects; whenever RTSim expects the name of any of said objects, the name of a pointer pointing to an object of the same type can be used instead, in dereferenced notation (using a "*" prefix); namely, the operation `lock(*pM)` unlocks the mutex that is referenced by the pointer `pM`; this is useful in combination with the `popptr` and `pushptr` functions, as clarified in the example below.

As in the original RTSim code base, there are no instructions to declare mutex, condition variable, queue and pointer objects in

[1]More information is available at: http://rtsim.sourceforge.net.

```
fixed(1);
lock(ServerM);
  fixed(2);
  pushptr(ServerQ,ClientQ,
          ClientCV,ClientM);
unlock(ServerM);
signal(ServerM,ServerCV);

fixed(1);
lock(ClientQ);
  fixed(2);
  waitq(ClientCV,ClientM,
        ClientQ);
  pop(ClientQ);
unlock(ClientM);
```

```
fixed(1);
lock(ServerM);
  fixed(2);
  waitq(ServerM,ServerCV,
        ServerQ);
  popptr(ServerQ,pClientQ,
         pClientCV,pClientM);
unlock(ServerM);

fixed(5);

fixed(1);
lock(*pClientM);
  fixed(2);
  push(*pClientQ);
unlock(*pClientM);
signal(*pClientM,*pClientCV);
```

Figure 6. Task code for client (left) and server (right) using PI-CV.

the tasks code, but these have to be created by using the RTSim API before adding code to the tasks. The above elements can be used to code a client-server interaction, as shown in Figure 6.

As it can be seen, the level of detail for the simulation may be kept to a minimum, neglecting details related to the functional aspects of the simulated tasks, but catching the main behavioral aspects that may impact their response-times.

The presented modifications to the RTSim open-source simulator have been submitted for clearance to be released in public and be freely made available to other researchers. However, at this time it is not clear whether this will be possible or not.

VI. Simulated Results

Using the implementation of PI-CV within RTSim as described in the previous section, an evaluation has been done by simulating a simple scenario with 3 client tasks using the same server task and running on a single-processor platform. For example, the server task might be representative of some OS service available through proper RPC calls, realized in terms of shared in-memory data structures protected by synchronizing access through mutexes and condition variables. In the simulated scenario, each client task is periodic, it spends a fixed time processing (see Table I), then it invokes the server by pushing a message onto the server receive queue, then it waits for a response to be placed by the server onto the client own receive queue. The server, on the other hand, is not periodic. It has been given the lowest possible priority within the system. It waits for an incoming message on its receive queue, then it computes for a fixed amount of time, then it pushes a message back onto the receive queue of the caller task, and it repeats forever. Task periods have been generated randomly. The overall experiment duration has been set to 200000 simulated time units, amounting to roughly 300 activations for each task. The overall set of used parameters for the 3 clients is summarized in Table I. The parameters have been roughly chosen to create a scenario in which the advantages of the proposed technique could be easily highlighted. Other overheads such as context switch or scheduling overheads have not been simulated. A more realistic simulation, including a careful tuning of the overheads and parameters around a real platform and OS and possibly a real application, is surely valuable future work to be done.

Parameter	Client1	Client2	Client3
Task period	676	683	687
Overhead of `lock()/unlock()`	1	1	1
Overhead of `wait()/signal()`	2	2	2
Overhead of `push()/pop()`	2	2	2
Overhead of `pushptr()/popptr()`	2	2	2
Job own computation	50	50	50
Server call computation	20		
Experiment duration	200000		

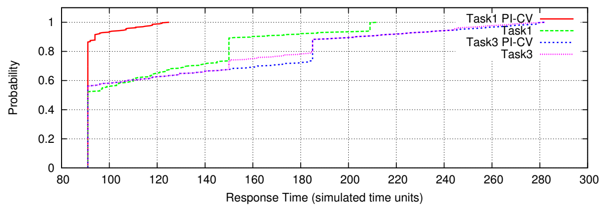
Table I
Task parameters for the simulated scenario (all values are expressed in the simulated time units).

Client-server interactions have been simulated in RTSim following the code structure exemplified in the previous section making use of the `Queue` type and of the `Pointer` type for the server. Two simulations have been done, one with only the traditional priority inheritance on all mutexes, and the other one with also the PI-CV mechanism on all condition variables (the two mechanisms acted in an integrated fashion as explained above).

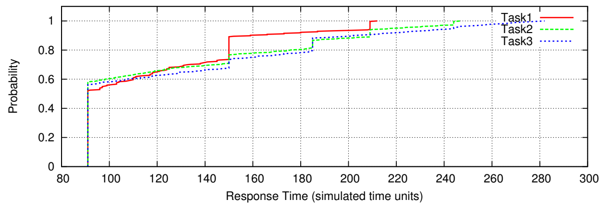
Figure 7.(a) reports the obtained Cumulative Distribution Functions (CDFs) of the response time (i.e., the difference between the job finishing and arrival times) of the highest and lowest priority clients for the experiment, in the two cases of with and without PI-CV. It is clearly visible that, when using PI-CV, the highest priority client greatly benefits of PI-CV, reducing its average and maximum response-times, at the expense of the lowest priority client for which both metrics become worse, as expected. As a result, PI-CV allows for avoiding unnecessary priority inversion. For completeness, Figures (b) and (c) report the CDFs for all the 3 clients in the two cases.

Figure 8 reports the cumulative simulated time units for which each task was assigned each priority value. Note that, in RTSim, the priority with numeric value 1 corresponds to the highest priority in the system. As it can be seen, the server task is assigned, for a significant part of the simulation, one of the clients priority levels, while it is serving requests on their behalf. Also, Client3 is assigned for a small time (note the logarithmic scale on the vertical axis) the boosted priority levels assigned to Client1 and Client2. This may be due to two factors. First, Client3 competes on the mutex protecting access to the server queue, thus whenever Client1 or Client2 wait for it to release the mutex before posting their message, the Client3 priority is correspondingly boosted to the level of Client1 or Client2 by the traditional priority inheritance mechanism. Second, whenever Client1 or Client2 submit a request to the server and start waiting for the response, but the server is still serving Client3, and no mutex is being held by any task, PI-CV boosts the priority of Client3 to the level of Client1 or Client2, depending on who is actually waiting on the condition variable.

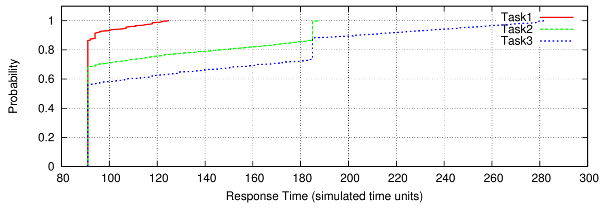
It has to be noted that the effectiveness of PI-CV and its quantitative impact on the tasks performance depends essentially on how much time a task spends wait()-ing on a



(a)



(b)



(c)

Figure 7. Response time CDFs of the response time of: (a) the highest-priority client (Task1) and the lowest-priority client (Task3) in the two cases of with and without PI-CV; (b) the 3 clients when executing without PI-CV and (c) with PI-CV.

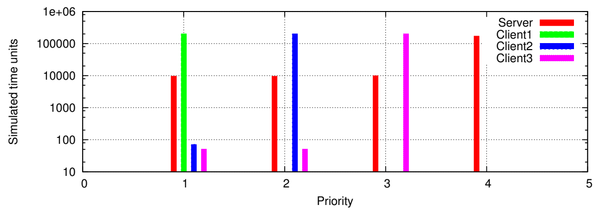


Figure 8. Cumulative simulated time units spent by each task into each priority value (1 is the highest priority, 4 is the lowest priority in the system).

condition variable for which helper tasks are defined, i.e., how much time is needed for the corresponding notify() to occur. This time is of course very application-specific. Comparing with traditional priority inheritance on mutex semaphores, in that case the effectiveness of the mechanism depends on how much time a task spends in a critical section with a mutex locked, which is *also* very application-specific. Though, the time spent with a mutex locked may be expected to be lower than the one spent wait()-ing for a notify() by some other task. Therefore, whenever it is possible to identify dependency relationships among real-time tasks, the presented PI-CV mechanism may be exploited to avoid situations of priority inversion expected to be of longer durations.

VII. Conclusions

In this paper, a mechanism has been presented for enhancing real-time systems with priority inheritance in presence of mutual exclusion semaphores and condition variables. The new mechanism, called PI-CV, alleviates the problem of priority inversion in cases in which developers code into the system custom, application-specific interaction and communication/synchronization logic by means of mutex and condition variables. With PI-CV, the programmer may declare what are the tasks that may help a condition to become true, over which other tasks may be waiting. Exploiting such dependency information, the run-time (Operating System) can correspondingly trigger the needed priority inheritance among tasks, mixing with traditional priority inheritance (e.g., as available through the `PTHREAD_PRIO_INHERIT` attribute in POSIX).

Whether or not it is meaningful that a higher-priority task suspends waiting for possible lower-priority tasks to provide some output, is something belonging to the application logic, and outside the scope of this paper. Whenever priorities of tasks can be meaningfully fine-tuned ahead of time, PI-CV might not be needed at all. However, PI-CV is applicable and useful in all those situations in which a task might need interactions with multiple tasks of different priorities (that go beyond the simple synchronization by mutual exclusion semaphores but need to recur to condition variables). This is a situation that might occur frequently in the design of real-time and embedded systems, for example for OS or middleware services that are shared across all real-time tasks within the system, as shown in the simulated scenario of Section VI.

Even though the proposed technique has been prototyped within the RTSim open-source simulator for real-time systems, possible future work includes the implementation of the proposed technique within a real OS (e.g., by extending the pthreads library and kernel functionality on Linux and integrating the technique with the SCHED_DEADLINE deadline-based scheduler [48]) showing its usefulness in concrete application contexts. For example, the Jack low-latency audio development framework allows for realizing arbitrary DAGs of inter-connected components and filters, in the audio processing pipeline. As the framework was already modified [49] for using a deadline-based scheduler, it would be interesting, in a multi-processor context, to leverage the PI-CV mechanism in order to let all the resource reservations involved in the audio processing pipeline automatically synchronize over (inherit) the common deadline of delivery of the audio frames to the speakers. Other directions for future work go of course along the direction of extending existing schedulability analysis techniques in presence of PI-CV. For example, the analysis presented in [39] might be extended and generalized for such purpose.

References

[1] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols, an approach to real-time synchronization," Tech. Rep. CMU-CS-87-181, Carnegie-Mellon University, November 1987.

[2] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *Computers, IEEE Transactions on*, vol. 39, pp. 1175 –1185, sep 1990.

[3] *IEEE Std 1003.1-1990, IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language]*, 1990.

[4] U. Drepper and I. Molnar, "The Native POSIX Thread Library for Linux," tech. rep., Red Hat Inc., February 2001.

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

[6] D. Cornhilll, L. Sha, and J. P. Lehoczky, "Limitations of Ada for real-time scheduling," in *Proceedings of the first international workshop on Real-time Ada issues*, IRTAW '87, (New York), pp. 33–39, ACM, 1987.

[7] D. Cornhill and L. Sha, "Priority inversion in Ada," *Ada Lett.*, vol. VII, pp. 30–32, Nov. 1987.

[8] J. B. Goodenough and L. Sha, "The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High-Priority Ada Tasks," Tech. Rep. CMU/SEI-88-SR-4, Carnegie-Mellon University, March 1988.

[9] M. W. Borger and R. Rajkumar, "Implementing Priority Inheritance Algorithms in an Ada Runtime System," Tech. Rep. CMU/SEI-89-TR-015, Carnegie Mellon University, April 1989.

[10] C. D. Locke and J. B. Goodenough, "A practical application of the ceiling protocol in a real-time system," in *Proceedings of the second international workshop on Real-time Ada issues*, IRTAW '88, (NY), pp. 35–38, ACM, 1988.

[11] L. Sha, R. Rajkumar, and J. Lehoczky, "A priority driven approach to real-time concurrency control," tech. rep., CMU, July 1988.

[12] R. Rajkumar, L. Sha, and J. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *Real-Time Systems Symposium, 1988., Proceedings.*, pp. 259 –269, dec 1988.

[13] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *Proceedings of the International Conference on Distributed Computing Systems*, pp. 116–123, 1990.

[14] C.-M. Chen and S. K. Tripathi, "Multiprocessor priority ceiling based protocols," tech. rep., College Park, MD, USA, 1994.

[15] M.-I. Chen and K.-J. Lin, "Dynamic priority ceilings: a concurrency control protocol for rt systems," *RTSJ*, vol. 2, pp. 325–346, Oct. 1990.

[16] T. P. Baker, "Stack-based scheduling for realtime processes," *Real-Time Syst.*, vol. 3, pp. 67–99, Apr. 1991.

[17] P. Gai, G. Lipari, and M. D. Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip," in *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, RTSS '01, (Washington, DC, USA), pp. 73–, IEEE Computer Society, 2001.

[18] K. Lakshmanan, D. d. Niz, and R. Rajkumar, "Coordinated task scheduling, allocation and synchronization on multiprocessors," in *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, RTSS '09, (Washington, DC, USA), pp. 469–478, IEEE Computer Society, 2009.

[19] D. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Alleviating Priority Inversion and Non-Determinism in Real-Time CORBA ORB Core Architectures," in *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, RTAS '98, (Washington, DC, USA), pp. 92–, IEEE Computer Society, 1998.

[20] M. Di Natale and A. Meschi, "Guaranteeing end-to-end deadlines in distributed client-server applications," in *Real-Time Systems, 1998. Proceedings. 10th Euromicro Workshop on*, pp. 163 –171, jun 1998.

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

[22] G. Lamastra, G. Lipari, and L. Abeni, "A bandwidth inheritance algorithm for real-time task synchronization in open systems," in *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, pp. 151 – 160, dec. 2001.

[23] G. Lipari, G. Lamastra, and L. Abeni, "Task synchronization in reservation-based real-time systems," *IEEE Trans. Comput.*, vol. 53, pp. 1591–1601, Dec. 2004.

[24] D. Faggioli, G. Lipari, and T. Cucinotta, "An efficient implementation of the bandwidth inheritance protocol for handling hard and soft real-time applications in the linux kernel," in *Proceedings of the* 4^{th} *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2008)*, (Prague, Czech Republic), July 2008.

[25] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "AQuoSA – Adaptive Quality of Service Architecture," *Software: Practice and Experience*, vol. 39, no. 1, pp. 1–31, 2009.

[26] D. Faggioli, G. Lipari, and T. Cucinotta, "The multiprocessor bandwidth inheritance protocol," in *Proc. of the 22nd Euromicro Conference on Real-Time Systems (ECRTS 2010)*, pp. 90–99, 2010.

[27] D. Faggioli, G. Lipari, and T. Cucinotta, "Analysis and implementation of the multiprocessor bandwidth inheritance protocol," *Real-Time Systems*, vol. 48, pp. 789–825, 2012. 10.1007/s11241-012-9162-0.

[28] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "Litmus-rt: A testbed for empirically comparing real-time multiprocessor schedulers," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS '06, (Washington, DC, USA), pp. 111–126, IEEE Computer Society, 2006.

[29] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson, "A flexible real-time locking protocol for multiprocessors," in *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pp. 47 –56, aug. 2007.

[30] N. Guan, P. Ekberg, M. Stigge, and W. Yi, "Resource sharing protocols for real-time task graph systems," in *Proc. of the 23rd Euromicro Conference on Real-Time Systems*, (Porto, Portugal), July 2011.

[31] B. B. Brandenburg and J. H. Anderson, "Optimality results for multiprocessor real-time locking," in *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, (Washington, DC, USA), pp. 49–60, IEEE Computer Society, 2010.

[32] M. Bertogna, N. Fisher, and S. Baruah, "Resource-sharing servers for open environments," *Industrial Informatics, IEEE Transactions on*, vol. 5, pp. 202 –219, aug. 2009.

[33] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "Sirap: a synchronization protocol for hierarchical resource sharing real-time open systems," in *Proceedings of the 7th ACM and IEEE international conference on Embedded software*, 2007.

[34] R. I. Davis and A. Burns, "Resource sharing in hierarchical fixed priority pre-emptive systems," in *Proceedings of the IEEE Real-time Systems Symposium*, 2006.

[35] A. Easwaran and B. Andersson, "Resource sharing in global fixed-priority preemptive multiprocessor scheduling," in *Proceedings of IEEE Real-Time Systems Symposium*, 2009.

[36] G. Macariu, "Limited blocking resource sharing for global multiprocessor scheduling," in *Proc. of the 23rd Euromicro Conference on Real-Time Systems (ECRTS 2011)*, (Porto, Portugal), July 2011.

[37] M. M. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Dependable Resource Sharing for Compositional Real-Time Systems," in *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 153–163, IEEE, Aug. 2011.

[38] S. Xi, C. Li, C. Lu, , and C. Gill, "Limitations and solutions for real-time local inter-domain communication in xen," tech. rep., Oct 2012.

[39] L. Abeni and N. Manica, "Analysis of client/server interactions in a reservation-based system," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, (New York, NY, USA), pp. 1603–1609, ACM, 2013.

[40] D. Hart and D. Guniguntalay, "Requeue-pi: Making glibc condvars pi-aware," in *Proceedings of the Eleventh Real-Time Linux Workshop*, pp. 215–227, 2009.

[41] A. Dragojević *et al.*, "Why STM can be more than a research toy," *Commun. ACM*, vol. 54, pp. 70–77, Apr. 2011.

[42] J. Corbet, "CFS group scheduling." http://lwn.net/, July 2007.

[43] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves for multimedia operating systems," Tech. Rep. CMU-CS-93-157, Carnegie Mellon University, Pittsburgh, May 1993.

[44] 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*, Oct. 1993.

[45] T. Cucinotta, F. Checconi, L. Abeni, and L. Palopoli, "Self-tuning schedulers for legacy real-time applications," in *Proceedings of the* 5^{th} *European Conference on Computer Systems (Eurosys 2010)*, (Paris, France), European chapter of the ACM SIGOPS, April 2010.

[46] T. Cucinotta, L. Abeni, L. Palopoli, and F. Checconi, "The Wizard of OS: a Heartbeat for Legacy Multimedia Applications," in *Proceedings of the* 7^{th} *IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia 2009)*, (Grenoble, France), October 2009.

[47] L. Palopoli, G. Lipari, G. Lamastra, L. Abeni, G. Bolognini, and P. Ancilotti, "An object-oriented tool for simulating distributed real-time control systems," *Software: Practice and Experience*, vol. 32, no. 9, pp. 907–932, 2002.

[48] J. Lelli, D. Faggioli, T. Cucinotta, and G. Lipari, "An experimental comparison of different real-time schedulers on multicore systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2405 – 2416, 2012. Automated Software Evolution.

[49] T. Cucinotta, D. Faggioli, and G. Bagnoli, "Low-latency audio on linux by means of real-time scheduling," in *Proceedings of the Linux Audio Conference (LAC 2011)*, (Maynooth, Ireland), May 2011.