Real-Time Systems Journal manuscript No.
(will be inserted by the editor)

Analysis and Implementation of the Multiprocessor BandWidth Inheritance Protocol

Dario Faggioli(*) · **Giuseppe Lipari**(†) · **Tommaso Cucinotta**(§)

Received: date / Accepted: date

Abstract The Multiprocessor Bandwidth Inheritance (M-BWI) protocol is an extension of the Bandwidth Inheritance (BWI) protocol for symmetric multiprocessor systems. Similar to Priority Inheritance, M-BWI lets a task that has locked a resource execute in the resource reservations of the blocked tasks, thus reducing their blocking time. The protocol is particularly suitable for open systems where different kinds of tasks dynamically arrive and leave, because it guarantees temporal isolation among independent subsets of tasks without requiring any information on their temporal parameters. Additionally, if the temporal parameters of the interacting tasks are known, it is possible to compute an upper bound to the interference suffered by a task due to other interacting tasks. Thus, it is possible to provide timing guarantees for a subset of interacting hard real-time tasks. Finally, the M-BWI protocol is neutral to the underlying scheduling policy: it can be implemented in global, clustered and semi-partitioned scheduling.

After introducing the M-BWI protocol, in this paper we formally prove its isolation properties, and propose an algorithm to compute an upper bound to the interference suffered by a task. Then, we describe our implementation of the protocol for the $LITMUS^{RT}$ real-time testbed, and measure its overhead. Finally, we com-

The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7 under grant agreement n.248465 "S(o)OS – Service-oriented Operating Systems." and under grant agreement n. 246556, "RBUCE-UP"

(*) Real-Time Systems Laboratory, Scuola Superiore Sant'Anna
Via G. Moruzzi 1, 56124, Pisa (Italy)
e-mail: d.faggioli@sssup.it
(§) Alcatel-Lucent Bell Labs
Blanchardstown Business & Technology Park, Dublin (Ireland)
e-mail: tommaso.cucinotta@alcatel-lucent.com
(†) Laboratoire Spécification et Vérification, École Normal Supérieure Cachan,
61, Avenue du Président Wilson, 94235 Cachan,
and PRES Universud Paris
e-mail: giuseppe.lipari@lsv.ens-cachan.fr

pare M-BWI against FMLP and OMLP, two other protocols for resource sharing in multiprocessor systems.

Keywords Resource sharing · Real-Time · Multiprocessors · Resource Reservation · Priority Inheritance

1 Introduction

Multi-core platforms are being increasingly used in all areas of computing. They constitute an important step for the achievement of greater performance in the wide area of high-end servers and high-performance computing, as witnessed by the movement from the "frequency race" to the "core race". Furthermore, they constitute a promising technology for embedded and real-time systems, where providing the same computing power with multiple cores at reduced frequency may lead to advantages in terms of power consumption, something particularly important for battery-operated devices.

Therefore, an increasing effort is being dedicated in the real-time literature for multiprocessor scheduling, analysis and design methodologies. Particularly, one of the key challenges in this context is constituted by *resource synchronisation protocols*, allowing multiple threads, possibly deployed on multiple cores, to access shared resources still keeping serialisability [24] of the accesses. On symmetric shared-memory multi-core platforms, commonly used types of shared resources are in-memory shared data structures used for communication and synchronisation purposes. To avoid inconsistencies due to concurrency and parallelism, access to shared data must be protected by an appropriate access scheme.

Many different approaches have been proposed so far, including lock-based techniques, guaranteeing mutual exclusion among code sections accessing the same data, but also *wait-free* [12] and *lock-free* [2] techniques, which instead allow for true concurrent execution of the operations on the data structures, via appropriate access schemes guaranteeing consistency of the operations. Recently, the *transactional memory* (TM) programming paradigm is gaining momentum, thanks to its ability to make it easier to code certain types of interactions of parallel software.

However, most widely used techniques in the programming practice so far are based on *mutually exclusive semaphores* (a.k.a., mutexes): before accessing a shared memory area, a task must lock a semaphore and unlock it after completing the access. The mutex can be successfully locked by only one task at a time; if another task tries to lock an already locked mutex, it is *blocked*, i.e. it cannot continue its normal execution. The blocked task will be unblocked only when the mutex is unlocked by its *owner*.

In single processor systems, the blocked task is removed from its ready queue, and the scheduler chooses a new task to be executed. In multi-core systems, it may be useful to let the blocked task execute a waiting loop, until the mutex is unlocked. Such technique is often called spin-lock or *busy-wait*. The advantage of busy waiting is that the overhead of suspending and resuming the task is avoided, and this is particularly useful when the time between the lock and the unlock operations is very short.

A *resource access protocol* is the set of rules that the operating system uses to manage blocked tasks. These rules mandate whether a task is suspended or performs a busy-wait; how the queue of tasks blocked on a mutex is ordered; whether the priority of the task that owns the lock on a mutex is changed and how. When designing a resource access protocol for real-time applications, there are two important objectives: 1) at run-time, we must use scheduling schemes and resource access protocols to reduce the *blocking time* of important tasks; 2) off-line, we must be able to bound such blocking time and account for it in a schedulability analysis methodology.

In this paper, we consider *open real-time systems* where tasks can dynamically enter or leave the system at any time. Therefore, a run-time admission control scheme is needed to make sure that the new tasks do not jeopardise the schedulability of the already existing tasks. In addition, for robustness, security and safety issues, it is necessary to isolate and protect the temporal behaviour of one task from the others. In this way, it is possible to have tasks with different levels of temporal criticality coexisting in the same system.

Resource Reservations [44] were proved as effective techniques to achieve the goals of temporal isolation and real-time execution in open systems. Resource reservation techniques have initially been designed for the execution of independent tasks on single processor systems. Recently, they were extended to cope with hierarchical scheduling systems [20, 47, 31], and with tasks that interact with each other using locks [10, 21, 39]. Lamastra et al. proposed the Bandwidth Inheritance (BWI) protocol [29, 32] that combines the Constant Bandwidth Server [1] with Priority Inheritance [46] to achieve bandwidth isolation in open systems.

The Multiprocessor BWI (M-BWI) protocol described in this paper is an extension of the original BandWidth Inheritance Protocol to symmetric multiprocessor/-multicore systems. In order to reduce task waiting times in M-BWI, busy waiting techniques are combined with blocking and task migration. The protocol does not require any information on the temporal parameters of the tasks; hence, it is particularly suitable to open systems.

Nevertheless, the protocol supports hard real-time guarantees for critical tasks: if it is possible to estimate such parameters as the worst-case execution times and durations of the critical sections for the subset of tasks interacting with the task under analysis, then an upper bound to the task waiting times can be computed. Therefore, in this case it is possible to compute the reservation budget that is necessary to guarantee that the critical task will not miss its deadlines.

Finally, the M-BWI protocol is neutral to the underlying scheduling scheme, since it can be implemented in global, clustered and semi-partitioned scheduling algorithms.

1.1 Paper Contributions

The contribution of this paper is three-fold. First, M-BWI is described and its formal properties are derived and proved correct. Then, schedulability analysis for hard real-time tasks under M-BWI is presented. Finally, the implementation of M-BWI in $LITMUS^{RT}$, a well-known open-source testbed for the evaluation of real-time

scheduling algorithms[1], is also presented. An experimental evaluation of M-BWI performed on such an implementation is presented and discussed.

A preliminary version of this work appeared in [19]. In this extended paper the discussion is more complete and formal; comparison with the FMLP and OMLP protocols [6, 9] have been added; the evaluation is made through a real implementation of the proposed technique.

2 Related Work

Several solutions exist for sharing resources in multiprocessor systems. Most of these have been designed as extensions of uni-processor techniques [43, 42, 11, 33, 22, 28, 16]; fewer have been specifically conceived for multiprocessor systems [15, 6].

The Multiprocessor Priority Ceiling Protocol (MPCP) [43] and its later improvement [42] constitute an adaptation of PCP to work on fixed priority, partitioned multiprocessor scheduling algorithms. A recent variant [28] of MPCP differs from the previous ones in the fact that it introduces spin-locks to lower the blocking times of higher priority tasks, but the protocol still addresses only partitioned, fixed priority scheduling.

Chen and Tripathi [11] presented an extension of PCP to EDF. Later on, Gai et al. [22] extended the SRP for partitioned EDF. The paper deals with critical sections shared between tasks running on different processors by means of FIFO-based spin-locks, and forbids their nesting.

Concerning global scheduling algorithms, Devi et al. [15] proposed the analysis for non-preemptive execution of global critical sections and FIFO-based wait queues under EDF. Block et al. proposed FMLP [6] and validated it for different scheduling strategies (global and partitioned EDF and Pfair). FMLP employs both FIFO-based non-preemptive busy waiting and suspension blocking, depending on the critical section being declared as short or long by the user. Nesting of critical sections is permitted in FMLP, but the degree of locking parallelism is reduced by grouping the accesses to shared resources.

Brandenburg and Anderson [8, 9] discuss the definition of blocking time and priority inversion in multi-processor systems, and present the OMLP class of protocols. Currently OMLP only supports non-locked resources. Recently, Easwaran and Andersson presented the generalisation of PIP for globally scheduled multiprocessor systems [16]. They also introduced a new solution, which is a tunable adaptation of PCP with the aim of limiting the number of times a low priority task can block a higher priority one. Recently Macariu proposed Limited Blocking PCP [34] for global deadline-based schedulers, but this protocol does not support nesting of critical sections.

As it comes to sharing resources in reservation-based systems, the first proposals were made by Caccamo and Sha [10], by Niz et al. [40] and by Holman and Anderson [26]. Regarding hierarchical systems[2], Behnam et al. [3] and Fisher et al. [21] pro-

[1] More information is available at: `www.litmus-rt.org`.

[2] These, under certain assumptions and for the purposes of this paper, can be considered as a particular form of reservation-based systems

posed specific protocols to deal with shared resources. In these papers, a server that has not enough remaining budget to complete a critical section blocks before entering it, until the replenishment time. Davis and Burns [14] proposed a generalisation of the SRP for hierarchical systems, where servers that are running tasks inside critical sections are allowed to overcome the budget limit.

Furthermore, there is work ongoing by Nemati et al. [37, 36, 38] on both integrating the FMLP in hierarchical scheduling frameworks, or using a new adaptation of SRP, called MHSRP, for resource sharing in hierarchically scheduled multiprocessors.

Guan et al. recently [23] addressed resource sharing in graph-based real-time task models, proposing a new protocol called ACP which tackles the particular issue that often the actually accessed resources are determined only at run-time, depending on which branches the code actually executes.

For all these algorithms, the correctness of the scheduling algorithm depends on the correct setting of the parameters, among which there are worst-case computation times and durations of critical section. If the length of a critical section is underestimated, any task can miss a deadline. In other words, there is no isolation (or a very limited kind of isolation) and an error can propagate and cause a fault in another part of the system. For example, in [3] and [21], if the length of a critical section on a global resource is underestimated, the system could be overloaded and any task could miss its deadline.

To the best of our knowledge, the only two attempts to overcome this problem are the BandWidth Inheritance protocol by Lamastra et al. [29, 32], and the non-preemptive access to shared resources by Bertogna et al. [5, 25]. These approaches are well suited for open systems, but are limited to uni-processors. Also limited to uniprocessors was the attempt at tackling priority inheritance in deadline-based systems by Jansen et al. [27], in which a protocol similar to priority-ceiling was designed for EDF-based scheduling, and the schedulability analysis technique based on the demand-bound function for EDF was extended for such a protocol.

3 System Model

In this paper we focus on shared memory symmetric multiprocessor systems, consisting of m identical unit-capacity processors that share a common memory space.

A task τ_i is defined as a sequence of jobs $J_{i,j}$ – each job is a sequential piece of work to be executed on one processor at a time. Every job has an arrival time $a_{i,j}$ and a computation time $c_{i,j}$. A task is *sporadic* if $a_{i,j+1} \geq a_{i,j} + T_i$, and T_i is the minimum inter-arrival time. If $\forall j\, a_{i,j+1} = a_{i,j} + T_i$, then the task is *periodic* with period T_i. The worst-case execution time (WCET) of τ_i is an upper bound on the job computation time: $C_i \geq \max_j\{c_{i,j}\}$. Real-time tasks have a relative deadline D_i, and each job has an absolute deadline $d_{i,j} = a_{i,j} + D_i$, which is the absolute time by which the job has to complete.

Hard real-time tasks must respect all their deadlines. Soft real-time tasks can tolerate occasional and limited violations of their timing constraints. Non real-time tasks have no particular timing behaviour to comply with.

3.1 Critical Sections

Concurrently running tasks often need to interact through shared data structures, located in common memory areas. One way to avoid inconsistencies is to protect the shared variables with mutex semaphores (also called *locks*). In this paper we denote shared data structures protected by mutex semaphores as *software resources* or simply *resources*. In order to access a resource, a task has to first *lock* the resource semaphore; only one task at time can lock the same semaphore. From now on, the k-th mutex semaphore will simply be called *resource*, and it will be denoted by R_k.

When τ_j successfully locks a resource R_k, it is said to become the *lock owner* of R_k, and we denote this situation with $R_k \rightarrow \tau_j$. If another task τ_i tries to lock R_k while it is owned by τ_j, we say that τ_i is *blocked* on R_k: this is denoted with $\tau_i \rightarrow R_k$. In fact, τ_i cannot continue its execution until τ_j releases the resource. Typically, the operating system suspends τ_i until it can be granted access to R_k. Alternatively, τ_i can continue executing a *busy-wait*, i.e. it still occupies the processor waiting in a loop until the resource is released. When τ_j releases R_k, we say that it *unlocks* the resource; one of the blocked tasks (if any) is unblocked and becomes the new owner of R_k.

Notice that in this paper the term *blocking* refers only to a task suspension due to a lock operation on an already locked resource. Other types of suspensions (for example the end of a task job) are simply called *suspensions* or *self-suspensions*. Also, notice that our definition of *task blocking on a resource* has no relationship with the concepts of priority and priority inversion: it simply indicates that a task cannot continue execution until the resource is released. Therefore, as it will become more apparent in Section 6, the definition and results presented by Brandenburg and Anderson [8, 9] do not apply to our case.

The section of code between a lock operation and the corresponding unlock operation on the same resource is called *critical section*. A critical section of task τ_i on resource R_h can be *nested* inside another critical section on a different resource R_k if the lock on R_h is performed between the lock and the unlock on R_k. Two critical sections on R_k and R_h are *properly nested* when executed in the following order: lock on R_k, lock on R_h, unlock on R_h and unlock on R_k. We assume that critical sections are always properly nested.

In the case of nested critical sections, chained blocking is possible. A *blocking chain* from a task τ_i to a task τ_j is a sequence of alternating tasks and resources:

$$H_{i,j} = \{\tau_i \rightarrow R_{i,1} \rightarrow \tau_{i,1} \rightarrow R_{i,2} \rightarrow \ldots \rightarrow R_{i,\nu-1} \rightarrow \tau_j\}$$

such that τ_j is the lock owner on resource $R_{i,\nu-1}$ and τ_i is blocked on $R_{i,1}$[3]; each other task in the chain accesses resources with nested critical sections, being the lock owner of the preceding resource and blocking on the following resource. For example, the following blocking chain $H_{1,3} = \{\tau_1 \rightarrow R_1 \rightarrow \tau_2 \rightarrow R_2 \rightarrow \tau_3\}$ consists of 3 tasks: τ_3 that accesses R_2, τ_2 that accesses R_2 with a critical section nested inside a critical section on R_1, and τ_1 accessing R_1. This means that at run-time τ_1 can be

[3] Notice that we have re-labelled both tasks and resources in the chain to highlight the blocking sequence.

blocked by τ_2, and indirectly by τ_3. In this case τ_1 is said to be *interacting* with τ_2 and τ_3.

A blocking chain is a "snapshot" of a specific run-time situation. However, the concept of blocking chain can also be used to denote a potential situation that may happen at run-time. For example, chain $H_{1,3}$ can be built off-line by analysing the critical sections used by each task, and then at run-time it may happen or not. Therefore, in order to perform a schedulability analysis, it is possible to analyse the task code and build a set of potential blocking chains to understand the relationship between the tasks. In the previous example, τ_1 may or may not be blocked by τ_3 in a specific run. However, τ_3 cannot be blocked by τ_1, unless another blocking chain $H_{3,1}$ exists. Generally speaking τ_i can be blocked by τ_j if and only if a blocking chain $H_{i,j}$ exists.

Deadlock can be detected both off-line and on-line by computing blocking chains. If a blocking chain contains the same task or the same resource twice, then a locking cycle is possible, and a deadlock can happen at run-time. To simplify presentation, and without loss of generality, in this paper we assume that deadlock is not possible. Thus a task never appears more than once in each blocking chain, and all chains are finite sequences. However, our implementation in Section 7 can detect deadlocks at run-time.

We define the subset of tasks interacting with τ_i as follows:

$$\Psi_i = \{\tau_j | \exists H_{i,j}\}. \tag{1}$$

Two tasks τ_i and τ_h are said to be **non-interacting** if and only if $\tau_j \notin \Psi_i$ and $\tau_i \notin \Psi_j$. The set of tasks that directly or indirectly interact with a resource R_k is defined as:

$$\Gamma_k = \{\tau_j | \exists H_{j,h} = \{\tau_j \rightarrow \ldots R_k \rightarrow \tau_h\}\} \tag{2}$$

The ultimate goal of the M-BWI protocol is to provide bandwidth isolation between groups of non-interacting tasks: if $\tau_j \notin \Psi_i$, then τ_j cannot block τ_i and it cannot interfere with its execution (see Section 4).

3.2 Multiprocessor Scheduling

In multiprocessor systems, scheduling algorithms can be classified into global, partitioned and clustered. Global scheduling algorithms have only one queue for ready tasks, and the first m tasks in the queue are executed on the m available processors. As a consequence, a task can execute on any of the m processors, and can *migrate* from one processor to another even while executing a job. Global scheduling is possible on symmetric multiprocessor systems where all processors have equivalent characteristics (e.g., the same instruction set architecture).

Partitioning entails a static allocation of tasks to processors. The scheduler manages m different queues, one for each processor, and a task cannot migrate between processors. Partitioned scheduling is possible on a wide variety of hardware platform, including heterogeneous multiprocessors.

In clustered scheduling, the set of processors is divided into disjoint subsets (*clusters*) and each task is statically assigned to one cluster. Global scheduling is possible

within each cluster: there is one queue for each cluster, and a task can migrate between processors of its assigned cluster. Again, each cluster must consist of equivalent processors.

In this paper we assume that **task migration** is possible, i.e. that a task can occasionally migrate from one processor to another one. Therefore, we restrict our attention to symmetric multiprocessors platforms.

Regarding the scheduling algorithm, we do not make any specific assumption. The underlying scheduling mechanism can be global, partitioned or clustered scheduling. However, for the latter two algorithms, we assume that a task can occasionally violate the initial partitioning, and temporarily migrate from its assigned processor to another one not assigned to it for the sake of shortening the blocking time due to shared resources. For this reason, from now on we refer to these schedulers as *semi-partitioned schedulers*.

The mechanism will be explained in greater details in Section 5.

3.3 Resource Reservation

The main goal of our protocol is to guarantee timing isolation between non-interacting tasks. An effective way to provide timing isolation in real-time systems is to use the *resource reservation* paradigm [44, 1]. The idea is to *wrap* tasks inside schedulable entities called *servers* that monitor and limit the resource usage of the tasks.

A server S_i has a maximum budget Q_i and a period P_i, and *serves* one task [4]. The server is a schedulable entity: it means that the scheduler treats a server as it were a task. Therefore, depending on the specific scheduling algorithm, a server is assigned a priority (static of dynamic), and it is inserted in a ready queue. Each server then generates "jobs" which have computation times (bounded by the maximum budget) and absolute deadlines. To distinguish between the absolute deadline assigned to server jobs, and absolute deadlines assigned to real-time tasks, we call the former "scheduling deadlines".

The *scheduling deadline* is calculated by the reservation algorithm and it is used only for scheduling purposes (for example in the CBS algorithm [1], the scheduling deadline is used to order the queue of servers according to the Earliest Deadline First policy). When the server is dispatched to execute, the server task is executed instead according to the resource reservation algorithm in use. Notice that, when using resource reservations, priority (both static or dynamic) is assigned to servers, and not to tasks. A set of servers is said to be *schedulable* by a scheduling algorithm if each server job completes before its scheduling deadline. In general, schedulability of servers is not related with schedulability of the wrapped tasks. However, if the set of servers is schedulable, and there is an appropriate relationship between task parameters and server parameters, server schedulability may imply task schedulability. For example, when serving sporadic real-time tasks, if the server maximum budget is not less than the task WCET, and the server period is not larger than the task minimum

[4] Resource reservation and servers can also be used as the basis for hierarchical scheduling, in which case each server is assigned more than one task. In this paper, however, we will not take hierarchical scheduling into account.

inter-arrival time, then the task will meet its deadlines provided that the server meets its deadlines.

Many resource reservation algorithms have been proposed in the literature, both for fixed priority and for dynamic priority scheduling. They differ on the rules for updating their budget, suspending the task when the budget is depleted, reclaiming unused budget, etc. However, all of them provide some basic properties: a reserved task τ_i is guaranteed to execute at least for Q_i time units over every time interval of P_i time units; therefore, tasks are both confined (i.e., their capability of meeting their deadlines only depends on their own behaviour) and protected from each other (i.e., they always receive their reserved share of the CPU, without any interference from other tasks). The latter property is called *timing isolation*.

Two examples of resource reservation algorithms are the Constant Bandwidth Server (CBS [1]), for dynamic priority scheduling, and the Sporadic Server (SS [48]), for fixed priority scheduling. To describe a resource reservation algorithm, it is possible to use a state machine formalism. The state machine diagram of a server for a general reservation algorithm is depicted in Fig. 1. Usually, a server has a *current budget* (or simply *budget*) that is consumed while the served task is being executed, and a priority. Initially the server is in the `Idle` state. When a job of the served task is activated, the server moves to the `Active` state and it is inserted in the ready queue of the scheduler; in addition, its budget and priority are updated according to the server algorithm rules. When an active server is dispatched, it becomes `Running`, and its served task is executed; while the task executes, its budget is decreased. From there on, the server may:

- become `Active` again, if preempted by another server;
- become `Recharging`, if its budget is depleted;
- become `Idle`, if its task self-suspends (for example because of an *end of job* event).

On the way out from `Recharging` and `Idle`, the reservation algorithm checks whether the budget and the priority/deadline of the server needs to be updated. A more complete description of the state machine for algorithms like the CBS [1] can be found in [35].

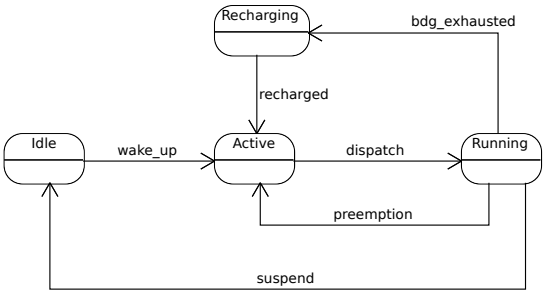


Fig. 1 State machine diagram of a resource reservation server.

4 The BandWidth Inheritance Protocol

If tasks share resources using the resource reservation paradigm, they might start interfering with each other. In fact, a special type of priority inversion is possible in such a case, due to the fact that a server may exhaust its budget while serving a task inside a critical section: the blocked tasks then need to wait for the server to recharge its budget. If the server is allowed to continue executing with a negative budget, scheduling anomalies appear that may prevent schedulability analysis, as explained for example in [32, 18].

For uni-processor systems, the Bandwidth Inheritance Protocol (BWI, see [32]) solves this issue by allowing *server inheritance*. The server of a lock-owner task can leverage not only its own budget to complete the critical section, but also the *inherited* budgets of servers possibly blocked on the lock it is owning.

This mechanism is similar to the Priority Inheritance mechanism. It helps the lock-owner to anticipate the resource release. Moreover, tasks that are not involved in the resource contention are not influenced, thus preserving timing isolation between non-interacting tasks.

A more detailed description of the BWI protocol and its properties can be found in [32]. In this paper we extend the BWI protocol to the multi-processor case.

In [45], BWI has been extended with the Clearing Fund algorithm. The idea is to *pay back* the budget that a task *steals* to other tasks by means of the bandwidth inheritance mechanism. While a similar technique can also be applied to M-BWI, for simplicity in this paper we restrict our attention to the original BWI protocol, and we leave an extension of the Clearing Fund algorithm as future work.

5 Multiprocessor Bandwidth Inheritance

When trying to adapt the BWI protocol to multiprocessor systems, the problem is to decide what to do when a task τ_A tries to lock a resource R whose lock owner τ_B is executing on a different processor. It makes no sense to execute τ_B on more than one CPU at the same time. However, just blocking τ_A and suspending the server may create problems to the resource reservation algorithm: as shown in [32], the suspended server must be treated as if its task completed its job; and the task unblocking must be considered as a new job. Whereas this strategy preserves the semantic of the resource reservation, it may be impossible to provide any timing guarantee to τ_A.

To solve this problem, M-BWI lets the blocked task τ_A perform a busy-wait inside its server. However, if the lock owner τ_B is not executing, because its server has been preempted (or exhausted its budget during the critical section) the inheritance mechanisms of BWI takes place and τ_B is executed in the server of the blocked task τ_A, thus reducing its waiting time. Therefore, it is necessary to understand what is the status of the lock owner before taking a decision on how to resolve the contention. It is also important to decide how to order the queue of tasks blocked on a locked resource.

5.1 State Machine

A server using the M-BWI protocol has some additional states. The new state machine is depicted in Figure 2 using the UML State Chart notation. In this diagram we show the old states grouped into a composite state called *Reservation*. As long as the task does not try to lock a resource, the server follows its original behaviour and stays inside the *Reservation* state.

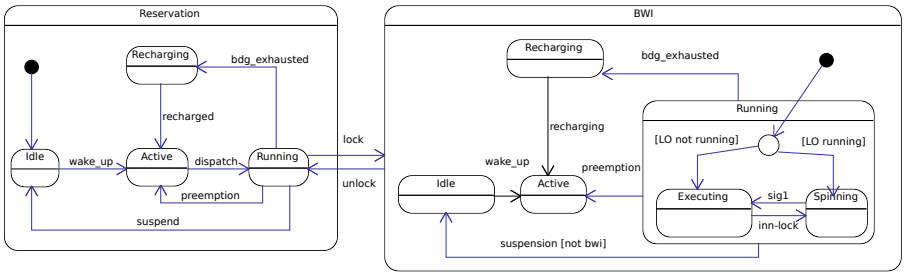


Fig. 2 State machine diagram of a resource reservation server when M-BWI is in place.

Now, let us describe the protocol rules. Let λ_j denote the set of blocked tasks waiting for τ_j to release some resource: $\lambda_j = \{\tau_k \mid \tau_k \rightarrow \ldots \rightarrow \tau_j\}$. Let ρ_k denote the set of all tasks blocked on resource R_k including the current lock-owner. Also, let Λ_j denote the set of servers currently inherited by τ_j (S_j included): $\Lambda_j = \{S_k \mid \tau_k \in \lambda_j\} \cup \{S_j\}$.

- **Locking rule**. When the task τ_i executing inside its server S_i tries to lock a resource R_k, the server moves into the `BWI` composite state, and more specifically inside the `BWI.Running` state, which is itself a state composed of two sub-states, `Executing` and `Spinning`. The set ρ_k now includes τ_i. We have two cases to consider:
  a) If the resource is free, the server simply moves into the `BWI.Running.Executing` sub-state and executes the critical section.
  b) If the resource is occupied, then the chain of blocked tasks is followed until one that is not blocked is found (this is always possible when there is no deadlock), let it be τ_j. Then, τ_j inherits server S_i, i.e. S_i is added to Λ_j. If τ_j is already executing in another server on another processor, then Server S_i moves into the `BWI.Running.Spinning` sub-state. Otherwise, it moves into `BWI.Running.Executing` and starts executing τ_j. This operation may involve a migration of task τ_j from one server to another one running on a different processor.

  Notice that in all cases S_i remains in the `BWI.Running` state, i.e. it is not suspended.
- **Preemption rule**. When server S_i is preempted while in the `BWI.Running` state, it moves to the `BWI.Active` state. We have two cases:

a) If the server was in the `BWI.Running.Spinning` sub-state, it simply moves to `BWI.Active`;
b) Suppose it was in the `BWI.Running.Executing` state, executing task τ_j. Then the list Λ_j of all servers inherited by τ_j is iterated to see if one of the servers $S_k \in \Lambda_j$ is running. This means that S_k must be in the `BWI.Running.Spinning` sub-state. Then, S_k moves to the `BWI.Running.Executing` sub-state and will now execute τ_j (transition `sig` in the figure).
If there is more than one server in Λ_j that is `BWI.Running.Spinning`, only one of them is selected and moved to `BWI.Running.Executing`, for example the one with the largest remaining budget, or the one with the earliest deadline.
This operation may involve a migration of task τ_j from server S_i into server S_k.

– **Recharging rule**. If the budget of a server in the `BWI.Running` state is exhausted, the server moves to the `BWI.Recharging` state. This rule is similar to the *Preemption rule* described above, so both cases a) and b) apply.
– **Dispatch rule**. If server S_i in the `BWI.Active` state is dispatched, it moves to the `BWI.Running` state. This rule is similar to the *locking rule* described above, and there are two cases to consider:
a) The lock-owner task is already executing in another server on another processor: then S_i moves to the `BWI.Running.Spinning` sub-state.
b) The lock-owner task is not currently executing; then S_i moves to the `BWI.Running.Executing` sub-state and starts executing the lock-owner task.
– **Inner locking**. If a task that is already the lock owner of a resource R_l tries to lock another resource R_h (this happens in case of nested critical section), then it behaves like in the *locking* rule above. In particular, if the resource is occupied, the lock owner of R_h is found and inherits S_i. If the lock-owner is already running in another server, S_i moves from the `BWI.Running.Executing` to the `BWI.Running.Spinning` sub-states (transition `inn-lock` in the figure).
– **Unlocking rule**. Suppose that a task τ_j is executing an outer critical section on resource R_k and unlocks it. Its current executing server must be in the `BWI.Running.Executing` sub-state (due to inheritance, it may or may not be S_j).
If there are blocked tasks in ρ_k, the first one (in FIFO order) is woken up, let it be τ_i. The unblocked task τ_i will inherit all servers that were inherited by τ_j, and all inherited servers are discarded from Λ_j (excluding S_j):

$$\begin{aligned} \Lambda_i &\leftarrow \Lambda_i \cup \Lambda_j \setminus S_j \\ \Lambda_j &\leftarrow S_j \end{aligned} \tag{3}$$

S_j goes out of the `BWI` composite state (transition `unlock`) and returns into the `Reservation` composite state, more precisely into its `Reservation.Running` sub-state. Notice that this operation may involve a migration (task τ_j may need to return executing into its own server on a different processor).
– **Inner unlocking rule**. If a task τ_j is executing a nested critical section on resource R_k and unlocks it, its currently executing server continues to stay in the `BWI.Running.Executing` sub-state. If there are blocked tasks in ρ_k waiting

for R_k, then the first one (according to the FIFO ordering) is woken up, let it be τ_i, and the sets are updated as follows:

$$\rho_k \leftarrow \rho_k \setminus \tau_j$$
$$\forall \tau_h \in \rho_k \quad \begin{cases} \Lambda_j \leftarrow \Lambda_j \setminus S_h \\ \Lambda_i \leftarrow \Lambda_i \cup S_h \end{cases}$$

This operation may involve a migration.

– **Suspension rule**. While holding a resource, it may happen that a task τ_j self suspends or blocks on a resource that is not under the control of the M-BWI protocol. This should not be allowed in a hard real-time application, otherwise it becomes impossible to analyse and test the schedulability. However, in a open system, where not everything is under control, it may happen that a task self-suspends while holding a M-BWI resource.
In that case, all the servers in Λ_j move to `BWI.Idle` and are removed from the scheduler ready queues until τ_j wakes up again. When waking up, all servers in Λ_j move to the `BWI.Active` state and the rules of the resource reservation algorithm are applied to update the budget and the priority of each server.

5.2 Examples

We now describe two complete examples of the M-BWI protocol. In the following figures, each time-line represents a server, and the default task of server S_A is τ_A, of server S_B is τ_B, etc. However, since with M-BWI tasks can execute in servers different from their default one, the label in the execution rectangle denotes which task is executing in the corresponding server. White rectangles are tasks executing non critical code, light grey rectangles are critical sections and dark grey rectangles correspond to servers that are busy waiting. Which critical section is being executed by which task can again be inferred by the *execution* label, thus A_1 denotes task τ_A executing a critical section on resource R_1. Finally, upside dashed arrows represent "inheritance events", i.e., tasks inheriting servers as consequences of some blocking.

The schedule for the first example is depicted in Figure 3. It consists of 3 tasks, τ_A, τ_B, τ_C, executed on 2 processors, that access only resource R_1.

At time 6, τ_B tries to lock R_1, which is already owned by τ_C, thus τ_C inherits S_B and starts executing its critical section on R_1 inside it. When τ_A tries to lock R_1 at time 9, both τ_C and τ_B inherit S_A, and both S_A and S_B can execute τ_C. Therefore, one of the two servers (S_A in this example) enters the *Spinning* state. Also, the FIFO wake-up policy is highlighted in this example: when, at time 14, τ_C releases R_1, τ_B grabs the lock because it issued the locking request before τ_A.

The second example, depicted in Figure 4, is more complicated by the presence of 5 tasks on 2 processors, two resources, and nested critical sections: the request for R_1 is issued by τ_C at time 7 when it already owns R_2.

Notice that, despite the fact that both τ_D and τ_E only use R_2, they are blocked by τ_A, which uses only R_1. This is because the behaviour of τ_C establishes the blocking chains $H_{D,A} = \{\tau_D \rightarrow R_2 \rightarrow \tau_C \rightarrow R_1 \rightarrow \tau_A\}$ and $H_{E,A} = \{\tau_E \rightarrow R_2 \rightarrow \tau_C \rightarrow$

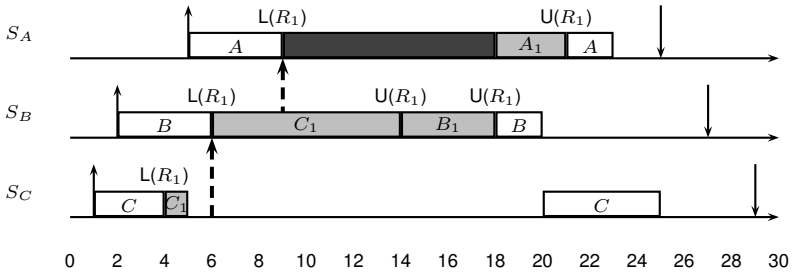


Fig. 3 First example, 3 tasks on 2 CPUs and 1 resource.

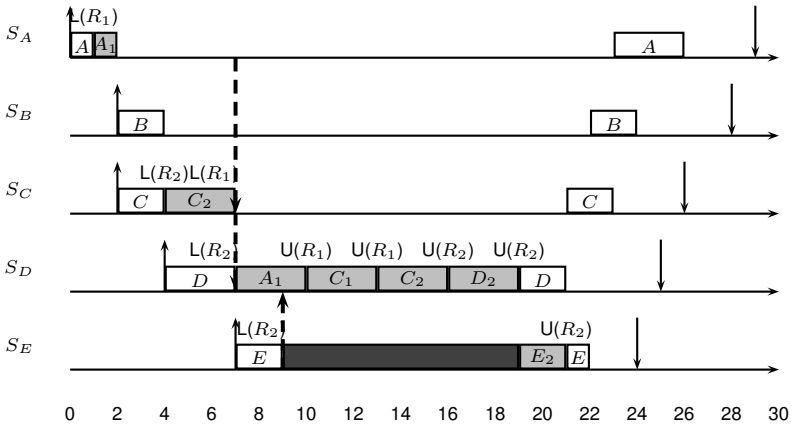


Fig. 4 Second example, 5 tasks on 2 CPUs with 2 resources — task τ_C accesses R_1 inside R_2.

$R_1 \rightarrow \tau_A\}$. For the same reason S_D and S_E are subject to interference either by busy waiting or executing τ_A until it releases R_1. This is a blocking-chain situation similar to what happens with priority inheritance in single processor systems.

5.3 Proof of Correctness

In this section, we will prove the correctness of the protocol. Let us start by defining what we mean by "correct protocol":

- First of all, we require that a task is never executed on two processors at the same time.
- Second, we require that the server is never blocked: that is, if task τ_i blocks, its server S_i will continue to execute either a busy-wait or some other task. Server S_i can suspend due to recharging, but it will never move to the `BWI.Idle` state, unless its currently executing task self-suspends.

– Finally, we require that, if a schedulability test deems the set of reservations to be schedulable when access to resources is ignored, then no server will miss its deadline at run-time when executed with the corresponding scheduling algorithm.

Notice that at this point we do not assume a specific scheduling algorithm (fixed or dynamic priority, semi-partitioned or global): we only assume a resource reservation algorithm, and an appropriate schedulability test for the admission control of reservations. The only requirement is that the set of reservations be schedulable on the selected combination of scheduling algorithm and hardware platform *when access to resources is not considered*.

Lemma 1 *If M-BWI is used as a resource access protocol, a task never executes on more than one server at the same time.*

Proof Suppose that τ_j is a lock owner that has inherited some server. For τ_j to execute in more than one server, at least two servers in Λ_j should be in the `Running.Executing` sub-state. However, the *Locking rule* specifically forbids this situation: in particular, in case b), the protocol looks at the lock owner task τ_j, ad if it already executing (i.e. if its server is in the `BWI.Running.Executing`), then the server of the new blocked task goes into `BWI.Running.Spinning` state.

Similar observations hold for the *Dispatch* and *Inner locking* rules. Hence the lemma is proved. □

Lemma 2 *Consider a set of reservations that uses the M-BWI protocol to access shared resources. Further, suppose that task τ_i and all tasks in Ψ_i never suspend inside a critical section, and never access a resource not handled by M-BWI. Then, when in the `BWI` state, server S_i always has exactly one non-blocked task to serve and never enters the `BWI.Idle` state.*

Proof The second part of the Lemma holds trivially: in fact, in order for S_i to enter the `BWI.Idle` state, it must happen that τ_i or any of the tasks from which it is blocked, self suspends while inside a critical section, against the hypothesis.

It remains to be proved that S_i has always exactly one non-blocked task to serve. In M-BWI a server can be inherited by a task due to blocking. This happens in the *Locking* and *Inner locking* rules. Also, in the *Unlocking* and *Inner unlocking* rules, a task can inherit many servers at once. Therefore, a task can execute in more than one server.

We will now prove that, when in the `BWI` state, server S_i has at most one non-blocked task to serve. By Induction. Let us denote with t_0 the first instant in which τ_i accesses a resource, entering state `BWI`. The lemma trivially holds immediately before t_0. Assume the lemma holds for all instants before time t, with $t \geq t_0$.

Suppose a task blocks at time t. In the *Locking rule* a task τ_i may block on a resource already occupied by another task τ_j. As a consequence, τ_j inherits S_i. S_i had only one non-blocked task (τ_i) before this event: hence, it has only one non-blocked task (τ_j) after the event. A similar observation is valid in the *Inner Locking rule*.

Suppose that a task τ_j releases a resource R_k at time t. In the *Unlocking rule*, τ_j wakes up one task τ_i that inherits all servers in Λ_j, except S_j. All these servers had

only one non-blocked task (τ_j) to serve before t; they still have one non-blocked task (τ_i) to serve after t. A similar observation holds for the *Inner unblocking rule*.

No other rule modifies any of the sets Λ_i. Hence the lemma is proved. □

The previous lemma implies that, under M-BWI, a server is never suspended before its task completes its job, unless the task itself (or any of its interfering tasks) self suspends inside a critical section. This is a very important property because it tells us that, from an external point of view, the behaviour of the reservation algorithm does not change. In other words, we can still view a server as a *sporadic task* with WCET equal to the maximum budget Q_i and minimum inter-arrival time equal to P_i, ignoring the fact that they access resources. Resource access, locking, blocking and busy wait have been "hidden" under the M-BWI internal mechanism. Therefore, we can continue to use the classical schedulability tests to guarantee that the servers will never miss their deadlines. This is formally proved by the following conclusive theorem.

Theorem 1 *Consider a set of reservations that is schedulable on a system when access to resources is ignored, and that uses M-BWI as a resource access protocol. Then, every server always respects its scheduling deadline.*

Proof Theorem 2 proves that a server is never blocked: a server can become idle (either `Reservation.Idle` or `BWI.Idle`) only if it self suspends or if it is blocked by a task that self suspends.

Notice in Figure 2 that the states inside `Reservation` and the states inside `BWI` were named alike with the purpose of highlighting the similarity between the two composite states. A server can move from `Reservation.Running` to `BWI.Running` and vice versa through a lock/unlock operation on a resource managed by the M-BWI protocol. Notice also that the server moves from one state to another inside each high level composite state responding to the same events: a *preemption* event moves a server from `Running` to `Active` in both composite states; a `bdg_exhausted` event moves the server from `Running` to `Recharging` in both composite states; etc. Also, the operations on the budget and priority of a reservation are identical in the two composite states, except that, while inside the `BWI` composite state, a server can execute a different task than its originally assigned one.

Therefore, from the point of view of an external observer, if we hide the presence of the two high level composite states, `Reservation` and `BWI`, and the lock and unlock events, then the behaviour of any server S_i cannot be distinguished from another server with the same budget and period that does not access any resource.

In any resource reservation algorithm, the schedulability of a set of reservations (i.e. the ability of the servers to meet their scheduling deadlines) depends only on their maximum budgets and periods. Since by hypothesis the set of reservations is schedulable on the system when ignoring resource access, it follows that the set of reservations continues to be schedulable also when resource access is considered. □

The most important consequence of Theorem 1 is that the ability of a server to meet its scheduling deadline is not influenced by the behaviour of the served tasks, but only by the global schedulability test for reservations. Therefore, regardless of

the fact that a task accesses critical sections or not, and for how long, the server will not miss its scheduling deadlines.

The first fundamental implication is that, to ensure that a task τ_i will complete before its deadline under all conditions, we must assign it a server S_i with *enough* budget and an appropriate period. If τ_i is sporadic and does not access any resource, it suffices to assign S_i a budget no less than the task's WCET, and a period no larger than the task's minimum inter-arrival time. In fact, the server will always stay inside the `Reservation` composite state and will not be influenced by the presence of other tasks in the system. We say that task τ_i is then *temporally isolated* from the rest of the system.

If τ_i does access some resource, then S_i can be inherited by other tasks due to blocking and the server budget can be consumed by other tasks. However, the set of tasks that can consume Q_i is limited to Ψ_i, i.e. the set of *interacting tasks* for τ_i. To ensure the schedulability of τ_i, we must assign S_i enough budget to cover for the task WCET and the duration of the critical sections of the interacting tasks. If a task does not belong to Ψ_i, then it cannot inherit S_i and cannot influence the schedulability of τ_i.

The conclusion is that M-BWI guarantees temporal isolation: it restricts the interference between tasks, and makes sure that only interacting tasks can interfere with each other.

6 M-BWI Interference Analysis

In the previous section we have demonstrated that M-BWI does indeed provide temporal isolation, without requiring any knowledge of the tasks temporal parameters. Also, M-BWI seamlessly integrates with existing resource reservation schedulers. Therefore, it is possible to avoid the difficult task of performing temporal analysis for soft real-time systems; for example, adaptive scheduling strategies [41, 13] may be used at run-time to appropriately dimension the budgets of the reservations.

Open systems may also include hard real-time applications, for which we must guarantee the respect of every temporal constraint. To perform an off-line analysis and provide guarantees, it is necessary to estimate the parameters (computation times, critical sections length, etc.) of the hard real-time tasks. Without isolation, however, the temporal parameters of every single task in the system must be precisely estimated. In M-BWI, this analysis can be restricted to the subset of tasks that interact with the hard real-time task under analysis. In particular, this is required to be able to compute the *interference* of interacting tasks.

The interference time I_i is defined as the maximum amount of time a server S_i is running but it is not executing its default task τ_i. In other words, I_i for S_i is the sum of two types of time interval:

- the ones when tasks other than τ_i execute inside S_i;
- the ones when τ_i is blocked and S_i busy-waits in `BWI.Running.Spinning` state.

Schedulability guarantees to hard real-time activities in the system are given by the following theorem.

Theorem 2 *Consider a set of reservations schedulable on a system when access to resources is not considered. When M-BWI is used as a resource access protocol, hard real-time task τ_i, with WCET C_i and minimum inter-arrival time T_i, attached to a server $S_i = (Q_i \geq C_i + I_i, P_i \leq T_i)$, never misses its deadline.*

Proof By contradiction. From Theorem 1, no server in the system misses its scheduling deadline. In order for τ_i to miss its deadline, the server has to go into the recharging state before τ_i has completed its instance. It follows that, from the activation of the task instance, the server has consumed all its budget by executing part of task τ_i and other interfering tasks. However, the amount of interference is upper bounded by I_i, the computation time of τ_i is upper bounded by C_i, and $Q_i \geq C_i + I_i$. Hence, the server never reaches the recharging state, and the theorem follows. □

Computing a bound on the interference for a hard real-time tasks is not easy in the general case of nested critical sections. In the following, we propose an algorithm to compute an upper bound to the interference that exhaustively checks all possible blocking conditions. The algorithm has super-exponential complexity, because it looks at all possible permutations of sequences of blocking tasks. However, consider that this algorithm is to be executed off-line; also, consider that in most practical cases, the number of resources and tasks involved in the computation is relatively small (i.e. below 10).

In the following, we also assume that the underlying scheduling algorithm is global EDF, which means that on a multiprocessor platform with m processors there is one global queue of servers, and the first m earliest deadline servers execute on the m processors. Also, we assume the Constant Bandwidth Server [1] as resource reservation algorithm.

6.1 Interference Computation

We start by observing that the two types of interference that a server can be subject to are "equivalent" from the point of view of the blocked task.

Theorem 3 *The interference for a server S_i is given by the sum of one or more instances of critical sections of tasks in Ψ_i.*

Proof Theorem 2 states that a server never blocks. It follows that when a task τ_i tries to access a resource that is already locked by another task τ_j, there are two possible cases. In one case, task τ_j is not executing because it is not the earliest deadline task, or because its server budget is 0. In this case, S_i inherits the locking tasks τ_j.

In the second case case, τ_j is executing on a different processor, so τ_i blocks and S_i spin locks, waiting for the τ_j to release the resource. It is easy to see that in both cases τ_i has to wait the same amount of time, that is the duration of the critical section of τ_j plus the possible interference time that τ_j can be subject to (due for example to nested critical sections). □

Theorems 2 and 3, combined together, tell us that there can be no indirect blocking in M-BWI.

Therefore, in order to compute the interference, we will assume that each one of the n tasks in the system executes on its own dedicated processor with a server that has maximum budget equal to its period. In this way, all active tasks are ready to execute, and the only possible type of interference is the second one (spin-lock). Also, in computing the interference for a job of task τ_i, we will assume that all other tasks will have minimal period and are always ready to interfere with τ_i. In this way we will compute a pessimistic but safe upper bound on the interference.

Once such upper bound has been computed on the dedicated virtual processors, we can go back to the original system with $m < n$ shared processors scheduled by global EDF, and in the worst-case the interference will not be larger than the one computed on the dedicated virtual processor platform.

We now compute the interference on the dedicated virtual multiprocessor platform. Let us start by modelling the critical sections.

We enumerate all critical sections of a task, and we denote by $\mathsf{cs}_i^{(j)}$ the j-th critical section of task τ_i. Also, the algorithm will make use of the following notation:

- $R(\mathsf{cs}_i^{(j)})$ is the resource accessed by the critical section
- $\mathsf{csset}_i(R_k)$ is the set of all critical sections of task τ_i that access R_k.
- $\mathsf{outer}(\mathsf{cs}_i^{(j)})$ is the set of all critical sections within which $\mathsf{cs}_i^{(j)}$ is nested. If $\mathsf{cs}_i^{(j)}$ is an outermost critical section, $\mathsf{outer}(\mathsf{cs}_i^{(j)}) = \emptyset$.
- $\mathsf{bres}(\mathsf{cs}_i^{(j)})$ is the set of all resources that have to be locked before task τ_i can access critical section $\mathsf{cs}_i^{(j)}$. In practice, it is the set of all resources accessed by the critical sections in $\mathsf{outer}(\mathsf{cs}_i^{(j)})$, and of course it can be empty if $\mathsf{cs}_i^{(j)}$ is an outermost critical section.
- $\mathsf{outmost}_i$ the set of outermost critical sections of task τ_i
- $\mathsf{inner}'(\mathsf{cs}_i^{(j)})$ is the set of critical sections that are directly nested inside $\mathsf{cs}_i^{(j)}$.
- $\mathsf{inner}(\mathsf{cs}_i^{(j)})$ is the set of all critical sections nested inside $\mathsf{cs}_i^{(j)}$, (i.e., the transitive closure of function $\mathsf{inner}'()$ on $\mathsf{cs}_i^{(j)}$)

The algorithm for the computation of the interference is reported in Figure 5. The algorithm consists of three functions: INTERFERENCE is the main function that is called by the user to compute the interference for a task τ_i. In turn, it calls function COMPUTEINTERFERENCE which performs the actual computation.

COMPUTEINTERFERENCE is a recursive function that takes 4 parameters. The first one is the task on which we want to compute the interference; the second one, CSSET, is the set of the critical sections of τ_i on which we want to compute the interference; the third parameter, BTASKS, is a set of blocking tasks (i.e. tasks that, in the enumerated scenario under analysis, have already blocked task τ_i on this or on some other critical section on another resource); the fourth parameter, BRES, is the set of locked resources, i.e. resources that have already been locked by some of the tasks in BTASKS.

The task can in principle block on each critical section in CSSET, therefore we have to sum the interference for each one of these critical sections. For each critical section, the set of tasks that can block τ_i is given by $\Theta = \Gamma(R(\mathsf{cs})) \setminus \mathsf{Btasks}$ (line 8).

The algorithm then explores all possible orderings in which the tasks in Θ block task τ_i on the current critical section (the cycle at lines 11-14). To understand why this is important, let us analyse one simple example.

Example: consider three tasks: τ_1 accesses R_2; τ_2 and τ_3 both access R_2 with a critical section nested inside another critical section on R_1. It is easy to show that τ_2 and τ_3 cannot both block τ_1 on R_2. By contradiction: since access is granted in FIFO order, the fact that τ_1 has to wait for both τ_2 and τ_3 to release R_2 implies that both tasks must have issued a request on R_2 before τ_1 issues its request on R_2. However, this means that both tasks must have successfully locked resource R_1, because both access R_2 with critical sections nested inside critical sections on R_1: this contradicts the mutual exclusion on R_1, hence only one between τ_2 and τ_3 can block τ_1 on R_2.

In the general case, we have to try all permutations of the tasks in Θ that can possibly block τ_i. For this reason, the algorithm performs a while cycle (line 11) in which it explores all possible permutations, computing the interference for each permutation by invoking function COMPUTEPERMUTATION, and selecting the maximum. Function NEXTPERMUTATION generates a new permutation of set Θ and returns *false* if no more permutations are possible.

After performing this step, function COMPUTEINTERFERENCE is called recursively on inner critical sections of CS (second parameter $\mathsf{inner}'(\mathsf{cs})$), by passing the set of blocking tasks and blocking resources (the latter one now contains resource $R(\mathsf{cs})$).

Function COMPUTEPERMUTATION performs the actual computation. It first selects the first task in the sequence (let it be task τ_j); then it selects the set of critical sections of this task on the resource R_k. Of course, τ_j can block τ_i only on one of those critical sections, hence it is necessary to see which one causes the worst-case interference. Hence, the algorithm first analyses if the set of resources in $\text{BRES}(csj)$ is free (line 27); if it is not, it means that the task cannot arrive before τ_i and lock the resource, otherwise some mutual exclusion constraint is violated. If it is possible, then we have to compute the length of the critical section plus the maximum interference that τ_j can suffer on this critical section; therefore, we recursively invoke function COMPUTEINTERFERENCE on τ_j (line 29). Among all possible critical sections of τ_j we select the one that produces the maximum interference on τ_i (lines 31-32). We perform this computations for all tasks in Θ, keeping track at each cycle of the blocking resources and of the blocking tasks (lines 23 and 37).

Example. In the previous example, consider the first permutation $\{\tau_2, \tau_3\}$. Function COMPUTEPERMUTATION will first add τ_2 to the list BTASKS (line 23); then it will look at all critical sections of τ_2 on R_2 (line 26); then it will compute $\text{BRES(CSJ)} = \{R_1\}$, and add it to the list of blocked resources TEMPBRES; then it will add the duration of its critical section on R_2 (line 29). Since it has no other inner critical section, it exits from the loop setting LOCALBRES to $\{R_1\}$. Now it goes back to line 22, adding τ_3 to BTASKS. However, it will realise that τ_3 cannot contribute to the interference because $\text{BRES}(csj) \cap \text{BRES} = \{R_1\}$ is non-empty.

Convergence and deadlock. The algorithm converges, since at each recursive step function COMPUTEINTERFERENCE is called with larger sets BTASKS and a larger BRES.

```
1: function INTERFERENCE(τ_i)
2:     return COMPUTEINTERFERENCE(τ_i, outermost_i, {τ_i}, ∅)
3: end function

4: function COMPUTEINTERFERENCE(τ_i, CSSet, BTasks, BRes)
5:     sum ← 0
6:     for all cs ∈ CSSet do
7:         LocalBRes ← BRes ∪ {R(cs)}
8:         Θ ← Γ(R(cs)) \ Btasks
9:         flag ← true
10:        partial ← 0
11:        while flag do
12:            partial ← max {partial, COMPUTEPERMUTATION(τ_i, Θ, R(cs), BTasks, LocalBRes)}.
13:            flag ← NEXTPERMUTATION(Θ)
14:        end while
15:        sum ← sum + partial + COMPUTEINTERFERENCE(τ_i, inner'(cs), BTasks, LocalBRes)
16:    end for
17:    return sum
18: end function

19: function COMPUTEPERMUTATION(τ_i, Θ, R_k, BTasks, BRes)
20:    sum ← 0
21:    LocalBRes ← BRes
22:    for all τ_j ∈ Θ do
23:        LocalBTask ← BTasks ∪ {τ}
24:        maxl ← 0
25:        MaxLocalBRes ← LocalBRes
26:        for all csj ∈ csset_j(R_k) do
27:            if bres(csj) ∩ BRes ≡ ∅ then
28:                TempBRes ← LocalBRes ∪ bres(csj)
29:                l ← length(csj)+COMPUTEINTERFERENCE(τ_j, inner'(csj), LocalBTask, TempBRes)
30:                if maxl < l then
31:                    maxl ← l
32:                    MaxLocalBRes ← TempBRes
33:                end if
34:            end if
35:        end for
36:        sum ← sum + maxl
37:        LocalBRes ← MaxLocalBRes
38:    end for
39:    return sum
40: end function
```

Fig. 5 Algorithm for computing the interference

The algorithm is correct under the assumption that there is no deadlock. We assume that deadlock is avoided by making sure that resources are totally ordered and nested critical sections access resources according to the selected order. More formally, if $\prec$ is the total order relationship between resources, we require that:

$$\forall \mathsf{cs}_i^{(j)}, \forall R_k \in \mathsf{bres}(\mathsf{cs}_i^{(j)}), R_k \prec R(\mathsf{cs}_i^{(j)})$$

Proof of correctness. We now prove the relationship between the interference computed by the algorithm in Figure 5 and the interference in the shared system.

Theorem 4 *Consider a system consisting of n tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$, each one served by a server S_i with parameters (Q_i, P_i), scheduled by Global EDF on m processors, and M-BWI as a resource access protocol.*

The interference time I'_i computed by Algorithm Interference *(Figure 5) is an upper bound on the worst case interference I_i of task τ_i.*

Proof Suppose that for some task τ_i, interference $I_i > I'_i$. From Theorem 3 it follows that some critical section contributes to I_i but not to I'_i. This means that some possible blocking chain H was not explored by the algorithm.

Now we will prove that this cannot happen, thus showing that all possible blocking chains are explored, by using induction.

Base of the induction step. Suppose that the blocking chain has length 3, i.e. $H = \{\tau_i \rightarrow R_k \rightarrow \tau_j\}$. Then, $\tau_j \in \Theta$ for task τ_i (line 8), and all feasible critical sections of τ_j on R_k are explored by function COMPUTEPERMUTATION (cycle at lines 25-35), and are therefore accounted for in I_i.

Induction hypothesis. Suppose that, for all τ_l, all blocking chains $H_{i,l}$ of length $n \geq 3$ have been accounted for in I'_i. Consider a blocking chain $H_{i,j}$ of length $n+2$ (if any): $H_{i,j} = \{\tau_i \rightarrow \ldots R_p \rightarrow \tau_l \rightarrow R_k \rightarrow \tau_j\}$. By the induction step, the sub-chain $H_{i,l} = \{\tau_i, \rightarrow \ldots \tau_l\}$ has been explored. Also, τ_l accesses R_k with a critical section nested inside a critical section on R_p. While computing the interference caused by the critical section $\mathtt{cs}_l$ of τ_l on R_p, the algorithm also takes into account the interference caused by all inner critical sections (line 29), including the one of τ_j on R_k. Finally, consider that all permutations are considered, including the one in which τ_j arrives before τ_l.

Therefore, by induction we conclude that all possible blocking chains are explored in all possible orders. □

Complexity. The algorithm is very complex. It explores all possible blocking chains starting from every outermost critical section of τ_i. Since the while cycle at line 11 is performed $O(p!)$, where $p = |\Theta|$, and function INTERFERENCE is recursive, the algorithm has super-exponential complexity.

However, it is important to highlight that the algorithm will be executed off-line; that the complexity is greatly reduced when critical sections are not nested; and that the number of interacting tasks in practical applications is usually low. In all our simulations (see Section 8) with $|\Theta| \leq 6$, we have never experienced a computation time of the algorithm superior to one second on a modern PC. With $|\Theta| = 10$ the duration of the algorithm is around 2-3 minutes; for a larger number of tasks, the algorithm becomes intractable.

The algorithm is pessimistic; we do not take into account task periods or server periods in the analysis, therefore it may be that the actual worst-case interference time is lower than the one computed by this algorithm.

On the other hand, please notice that this algorithm can be applied to a more general setting than M-BWI systems: in particular, it can be used for a system consisting of a set of tasks, partitioned onto a multi-processor platform, which access global resources with a FIFO policy.

7 Implementation in $LITMUS^{RT}$

The M-BWI protocol has been implemented on the real-time scheduling and synchronisation testbed called $LITMUS^{RT}$, developed and maintained by the UNC real-time research group. Having a real implementation of the protocol allows us to perform more complex evaluations than just simulations, and get real data about scheduling overheads and actual execution times of the real-time tasks, as well as to measure performance figures.

$LITMUS^{RT}$ was chosen as the basis for the implementation of M-BWI because it is a well-established evaluation platform (especially for scheduling and synchronisation overheads) in the real-time research community. In fact, $LITMUS^{RT}$ includes *feather-trace*, an efficient and minimally intrusive mechanism for recording timestamps and tracing overheads of kernel code paths. Moreover, it already supports a variety of scheduling and synchronisation schemes. Therefore it will be easier (in future works) to adapt M-BWI to them and compare it with other solutions. The current version of $LITMUS^{RT}$ is available as a patch against Linux 2.6.36, or via UNC `git` repository (see $LITMUS^{RT}$ web page).

$LITMUS^{RT}$ employs a "plug-in based" architecture, where different scheduling algorithms can be "plugged", activated, and changed dynamically at run-time. Consistently with the remainder of this paper, M-BWI has been implemented for global EDF, i.e., inside the plug-in called `C-EDF` (since it also supports clustered scheduling if configured accordingly). Our M-BWI patch against the development trunk (the `git` repository) version of $LITMUS^{RT}$ is available at:
`http://retis.sssup.it/people/tommaso/papers/RTSJ11/index.html`.

This section reports the principal aspects and the fundamental design choices that drove the implementation.

7.1 Implementing the Constant BandWidth Server

As the first step, the `C-EDF` plug-in has been enriched with the typical deadline postponement of the CBS algorithm, which was not included in the standard distribution of $LITMUS^{RT}$. After this modification it is possible for a task to ask for budget enforcement but, upon reaching the limit, to have it replenished and get a deadline postponement, rather than being suspended till the next period. This is done by a new parameter in the real-time API $LITMUS^{RT}$ offers to tasks, called `budget_action` that can be set to `POSTPONE_DEADLINE`.

Of course, CBS also prescribes that, when a new instance arrives, the current scheduling parameters need to be checked against the possibility of keeping using them, or calculating a new deadline and issue a budget replenishment. This was realised by instrumenting the task wake-up hook of the plug-in, i.e., `cedf_task_wake_up`.

The amount of modified code is small (8 files changed, 167 lines inserted, 33 deleted), thanks to the extensible architecture of $LITMUS^{RT}$ and to the high level of separation of concerns between tasks, jobs and budget enforcement it achieves.

7.2 Implementing Proxy Execution

The fundamental block on top of which M-BWI has been implemented is a mechanism known as *proxy execution*. This basically means that a task τ_i can be the *proxy* of some other tasks τ_j, i.e., whenever the scheduler selects τ_i, it is τ_j that is actually dispatched to run. It is a general mechanism, but it is also particularly well suited for implementing a protocol like M-BWI.

Thanks to the simple plug-in architecture of $LITMUS^{RT}$, the implementation of this mechanism was rather simple, although some additional overhead may have been introduced. In fact, it has been necessary to decouple what the scheduling algorithm thinks it is the "scheduled" task (the *proxy*), from the task that is actually sent to the CPU (the *proxied*). Also, touching the logic behind the implementation of the scheduling algorithm (global or clustered EDF, in this case) can be completely avoided, and the code responsible for priority queues management, task migration, etc., keeps functioning the same as before the introduction of proxy execution.

If tasks are allowed to block or suspend (e.g., for the purpose of accessing an I/O device) while being proxied, this has to be dealt with explicitly (it corresponds to transition from `BWI.Running` to `BWI.Idle` in the state diagram of Figure 2). In fact, when a task self-suspends, it is necessary to remove all its proxies from the ready queue. However, walking through the list of all the proxies of a task is $O(n)$ — with n number of tasks blocked on the resources the task owns when it suspends — overhead that can be easily avoided, at least for this case. In fact, the proxies of the suspending task are left in the ready queue, and it is only when one of them is picked up by the scheduler that, if the proxied task is still not runnable, they are removed from the queue and a new candidate task is selected. On the other hand, when a task that is being proxied by some other tasks wakes up, not only that task, but also all its proxies have to wake up. In this case, there is no way for achieving this than going through the list of all the waking task's proxies, during its actual wake-up, and putting all of them back to the ready queue.

In LITMUS^{RT}, self-suspension and blocking are handled by the same function `cedf_task_block`. Therefore, to implement the correct behaviour, `cedf_task_block` and `cedf_task_wake_up` have been modified. For each task, a list of tasks that are proxying it at any given time is added to the process control block (`task_struct`). The list is updated when a new proxying relationship is established or removed, and it is traversed at each self-suspension or wake-up of a proxied task. Each task is provided with a pointer to its current proxy (`proxying_for`) which is filled and updated when the proxying status of the task changes. Such field is also referenced within the scheduler code, in order to determine whether the selected task is a proxy or not.

Implementing proxy execution was more complex than just adding budget postponement (478 line additions, 74 line deletions).

As a final remark, consider that when resource reservations are being used, the budgets of the involved servers need to be properly managed while the proxy execution mechanism is triggered. The details of the budget updating are described in the next section.

7.3 Implementing Multiprocessor BandWidth Inheritance

Using a mechanism like proxy execution, implementing M-BWI is a matter of having FIFO wait queues for locks and taking care of the busy waiting of all the proxies whose proxying task is already running on some CPU.

The former is achieved by adding a new type of lock (`bwi_semaphore`) in the $LITMUS^{RT}$ kernel, backed up with a standard Linux `waitqueue`, which supports FIFO enqueue and dequeue operations. Each semaphore protects its internal data structures (mainly the `waitqueue` and a pointer to the owner of the lock itself) by concurrent access from more than one CPU at the same time by a non-interruptible spin-lock (a native Linux `spinlock_t`). Moreover, when the locking or releasing code for a lock needs to update a `proxying_for` field, it is required for it to acquire the spin-lock that serialises all the scheduling decision for the system (or for the cluster) of the $LITMUS^{RT}$ scheduler.

For the busy wait part, a special kernel thread (a native Linux `kthread`) called `pe_stub-k` is spawned for each CPU during plug-in initialisation, and it is initially in a blocked state. When a task τ_i running on CPU k needs to busy wait, this special thread is selected as the new proxy for τ_i, while the real value of `proxying_for` of τ_i is cached. Therefore, `pe_stub-k` executes in place of τ_i, depleting its budget τ_i as it runs.

The special thread checks if the real proxied task of τ_i is still running somewhere; $LITMUS^{RT}$ provides a dedicated field for that in the process control block, called `scheduled_on`. Such field is accessed and modified by the scheduler, thus holding the scheduling decision spin-lock is needed for dealing with it. However, the busy waiting done by `pe_stub-k` must be preemptive and with external interrupts enabled for CPU-k. Therefore, `pe_stub-k` performs the following loop:

1. it checks if the real proxying task of τ_i is still running somewhere by looking at `scheduled_on` *without* holding any spin-lock;
2. as soon as it reveals something changed, e.g., `scheduled_on` for the proxying task becomes `NO_CPU`, it takes the spin-lock and checks the condition again:
   - if it is still `NO_CPU` it means the proxying task has been preempted or suspended and, through a request for rescheduling, it tries to start running it;
   - if it is no longer `NO_CPU`, someone has already started executing the proxying task (recall the busy wait performed inside `pe-stub-k` is preemptable), thus it goes back to point 1.

8 Simulation Results

The algorithm for computing an upper bound on the interference time described in Section 6.1 can be used to evaluate how large is the impact of M-BWI on the schedulability of hard real-time tasks in the system. To this end, we performed an analysis of the algorithm on synthetically generated task sets, and we compared the results against two other protocols: the Flexible Multiprocessor Locking Protocol – FMLP [6], which allows nested critical sections and mixes suspension and spin-lock blocking mechanisms; and the Optimal Multiprocessor Locking family of Protocols –

Symbol	Description	Values
m	number of processors	2, 4, 6
n	number of tasks	2 or 3 times the number of processors
s	number of short resources	2, 4, 6
l	number of long resources	0, 2
t	threshold between short an long resource	from 0.05 to 0.5 in steps of 0.05 (millisec.)
g	number of resource groups	1, 2, 3
z	nesting probability	10% or 0%
u	utilisation per processor	from 0.28 to 0.72 in steps of 0.04

Table 1 Parameters of the simulation.

OMLP [9], which combines FIFO queueing with Priority-based queueing of blocked tasks in order to optimise blocking time. However, at the time of this writing, OMLP did not support nested critical sections[5].

To perform the experiments we generated tasks sets according to the parameters reported in Table 1[6].

Since the FMLP protocol distinguishes between short and long resources, we generates a number of short and long resources for the system, denoted by s and l, respectively. Critical sections on short resources have duration in $[0.05, t)$ milliseconds, where t is the threshold parameter; long critical sections have duration in $[t, 0.5]$ milliseconds.

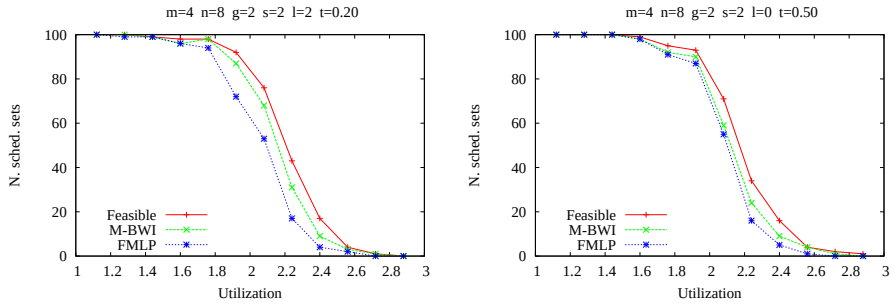
A task can have 1, 2 or 3 critical sections, with uniform probability equal to 0.9, 0.4 and 0.1, respectively. A critical section can be nested with probability equal to 0.1, and again it can have 1, 2, or 3 nested critical sections with the above probability. Long critical sections cannot be nested inside short critical sections. To avoid deadlock, we enumerated all resources, and guaranteed that a critical section on resource with a certain id cannot be nested inside a critical section on a resource with a higher id.

Task periods are generated with log-uniform distribution between 50 and 1000 milliseconds in steps of 50 milliseconds. Computation times are generated by using the randfixedsum algorithm [17], which guarantees a uniform distribution of computation times with a fixed utilisation equal to u. Execution times are inclusive of the duration of their critical sections.

Resources and tasks are divided into g groups (see Table 1). A task can only access resources from its own group. The simulation parameters have been set so that a group has at least two tasks and one resource, and no more than 6 tasks each. We kept the number of tasks per group limited because of the long execution time of the algorithm for computing the interference of M-BWI when the number of tasks accessing a given resource exceeded 6. We believe however that the results reported here are valid for a larger number of tasks per resource. Resource groups have been introduced in the simulation to highlight the isolation properties of M-BWI.

[5] Very recently, Ward and Anderson proposed an extension of OMLP to support nested critical sections [49].

[6] The test program is available on-line as open-source C++ code at `https://github.com/glipari/rtscan`. All the experiments described in this paper are available under directory `drivers/mbwi_exps/`.



(a) a) (b) b)

Fig. 6 Typical performance scenarios.

We generated 100 task sets for each combination of all these parameters. For each task set we first tested schedulability using the iterative test by Bertogna et al. [4]. If the task set was schedulable, the three protocols were analysed: we computed the interference time for M-BWI using the algorithm in Figure 5; we applied the analysis described in [6] for FMLP and the analysis described in [9] for OMLP to compute the blocking times. Then, we added the interference/blocking times to the computation times of the tasks, and we run again the schedulability test, recording the number of schedulable task sets for each protocol.

The remainder of this section shows some of the results of these simulations.

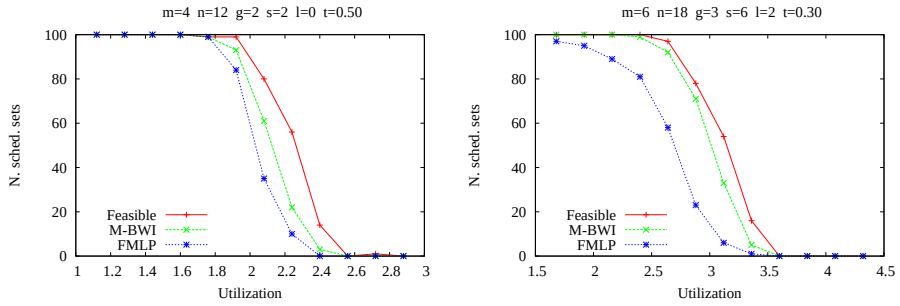
8.1 M-BWI vs. FMLP

In these experiments the nesting probability has been set equal to 0.1. All simulation experiments have shown that M-BWI and FMLP have similar performance, with M-BWI slightly better than FMLP, except in some specific combination of parameters.

In Figure 6 we report two typical performance result. At the top of the graph we report the most important parameters of the simulation; on the x axis we report the total utilisation, whereas on the y axis we report the number of schedulable task sets. The figure shows three lines: the top red line represents the number of schedulable task sets when interference and blocking is ignored; the green and blue lines show the number of schedulable task sets with M-BWI and FMLP, respectively. It is evident that in this case M-BWI is slightly better than FMLP.

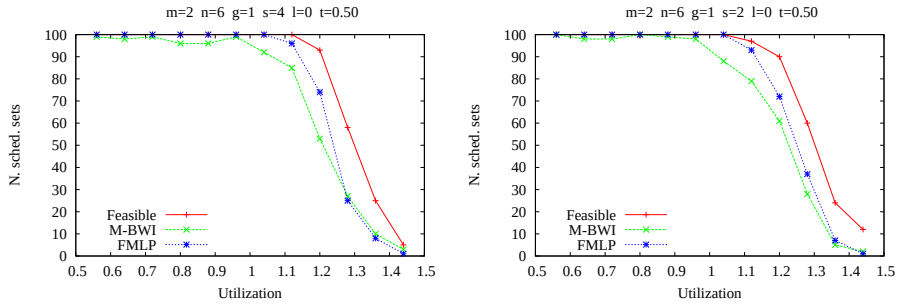
Most of the experiments with different combination of parameters follow a similar pattern. For example, in Figure 6b we show what happens when $l = 0$ (no long resources), and $t = 0.50$ (threshold set to maximum value). In this case, FMLP sees all resources as short, and always performs a busy waiting; however, in this case a "short" resource can have a duration up to 500μ sec. In this case there is no much difference between the two protocols.

The difference is more evident with other combination of parameters. In Figure 7, we see that for higher number of processors and higher number of tasks, M-BWI



(a) a) (b) b)

Fig. 7 Two example scenarios in which M-BWI outperforms FMLP.



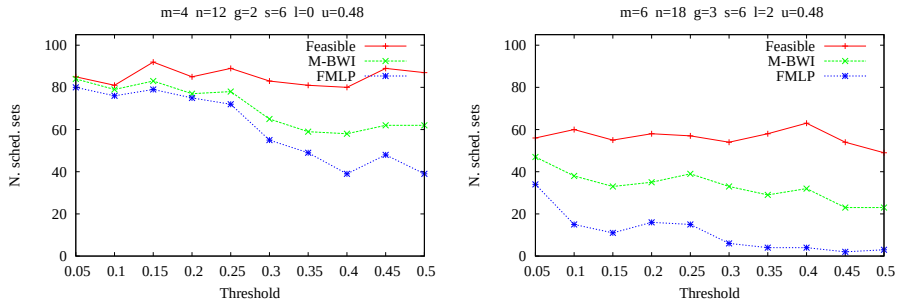
(a) a) (b) b)

Fig. 8 Two scenarios in which FMLP is better than M-BWI.

outperforms FMLP, with or without long resources. Also, in general the number of resource groups has a slight beneficial influence on the performance of M-BWI, thanks to the isolation properties of this protocol.

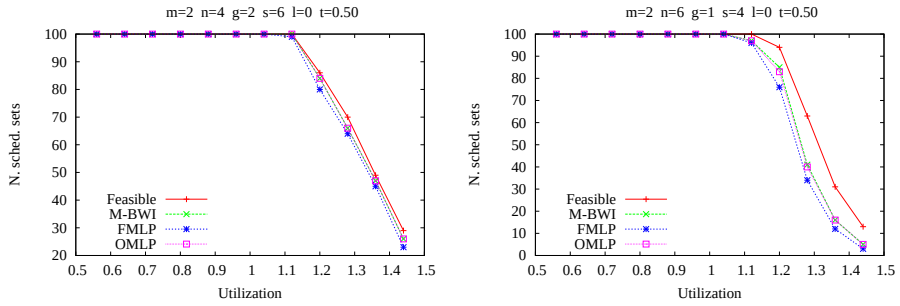
One combination of parameters favours FMLP. In Figure 8 we see that for $m = 2$ and $n = 6$, and only short resources, FMLP actually performs better than M-BWI. This is probably due to the fact that FMLP uses one single lock for all nested critical sections. In this way concurrency is reduced, but it avoids the problem of multiple interference on the external and internal critical section, a problem to which M-BWI is subject. In all our simulation, the combination of $m = 2, n = 6, g = 1, l = 0$ is the only one in which FMLP performs substantially better than M-BWI. However, this means than none of the two protocols dominates the other. Also, this opens a new line of research: to investigate when it is advantageous to group together locks for nested critical sections. We defer such investigation to a future paper.

In Figure 9, we show the impact of the threshold on the performance of the two protocols. The threshold sets an upper bound to the duration of short critical sections; in case long resources are present, it also sets a minimum upper bound on the duration of long critical sections. In Figure 9 we can see that the threshold has a minor effect



(a) a) (b) b)

Fig. 9 Two scenarios in which the performance of FMLP decreases by increasing the threshold.



(a) a) (b) b)

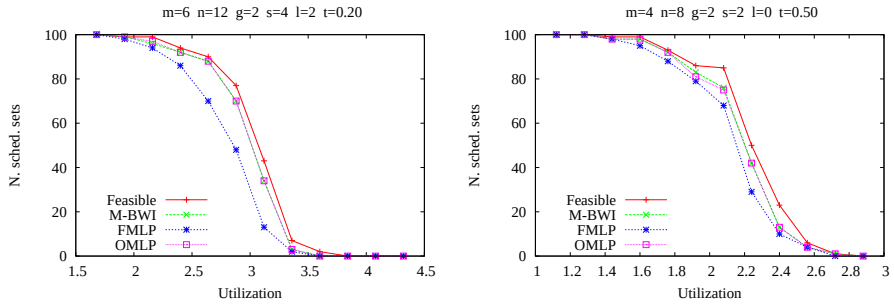
Fig. 10 When $m = 2$ and $n = 6$ and critical sections are not nested, all protocols behave similarly (see Figure 8).

mainly on FMLP. In general, the threshold has a minor impact on schedulability with respect to other important parameters, like number of tasks and number of processors.

8.2 No nested critical sections

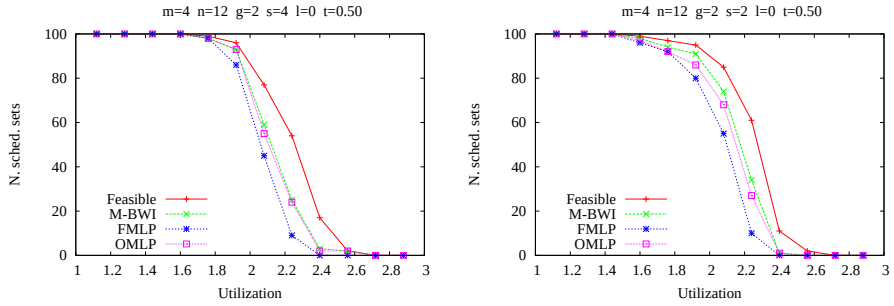
If we exclude the possibility of nested critical sections, it is possible to compare M-BWI with OMLP, a protocol generally superior to FMLP which has been demonstrated to be asymptotically optimal. Therefore, in the set of experiments described in this section, we set the nesting probability to 0.

First of all, let us confirm our interpretation of the results shown in Figure 8. In Figure 10, we see the results with the same parameters and no nested critical sections, and we observe that both OMLP and M-BWI behave slightly better than FMLP. This confirms that the superior performance of FMLP in the experiments of Figure 8 is due to the positive effect of grouping access to nested critical sections.



(a) a) (b) b)

Fig. 11 M-BWI and OMLP have very similar performance.



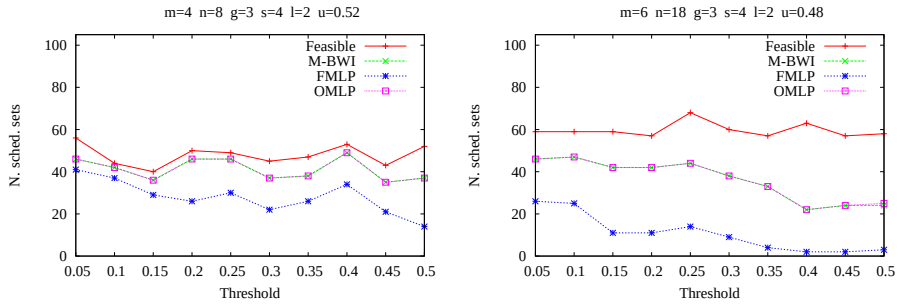
(a) a) (b) b)

Fig. 12 Two examples where M-BWI is slightly better than OMLP.

In almost all experiments, the performance of M-BWI and OMLP are so similar that the two curves are indistinguishable. Two typical scenarios are shown in Figure 11. In some rare cases, M-BWI shows a very small improvement over OMLP: two examples are shown in Figure 12. This is probably due to the isolation properties unique to the M-BWI protocol. Also in this case we show the impact of the threshold in Figure 13: again, the impact is minor. However, notice the great distance between the curves of FMLP and the curves of OMLP and M-BWI.

9 Experimental Results

In this section, we report performance figures obtained by running synthetically generated task sets on our implementation of M-BWI on the $LITMUS^{RT}$ operating system. The aim is to gather insights about how much overhead the protocol entails when executing on real hardware. We have generated the task sets parameters as described in Section 8. The hardware platform consists of a AMD Opteron processor with 48 cores, running at 1.9 GHz frequency. The cores are organised into 4



(a) a) (b) b)

Fig. 13 The impact of threshold on M-BWI and OMLP.

"islands" of 6 cores each, and all cores inside an island share the same L2 cache. In the experiments we selected only one island, and disabled the other three. In this way, the performance figures do not depend on unpredictable behaviours due to cache conflicts.

Therefore, 10 randomly chosen task sets among the ones generated for 6 CPUs, with different number of tasks N have been executed for 10 minutes each, while tracing the overheads with Feather-trace [7]. The number of short resources was fixed $N_{short} = 2 \cdot N$ and $N_{long} = \frac{M}{2} = 3$.

In this work, the scheduling overhead (i.e. the duration of the main scheduling function), the amount of time tasks wait (either being preempted, proxying or busy waiting) for a resource and the duration of lock and unlock operations are considered.

Scheduling Overhead. To evaluate the impact of M-BWI on the scheduler, we measured how long it takes for taking a scheduling decision in the following cases: (i) original $LITMUS^{RT}$ running the generated tasks sets but with tasks *not* issuing any resource request during their jobs ("Original" in the graphs); (ii) M-BWI enabled $LITMUS^{RT}$ but, again, with tasks not issuing resource requests ("No Res." in graphs); (iii) M-BWI enabled $LITMUS^{RT}$ with tasks actually locking and unlocking resources as prescribed in the task set ("M-BWI" in the graphs). Figure 14 shows the average duration of the scheduler function along with the standard deviation for the three cases, varying the number of tasks. The actual impact of M-BWI on the scheduler is limited, since the duration of the scheduling function is comparable for all the three cases, and independent from the number of tasks (when they exceed the number of available cores). In fact, in the proposed implementation, tasks that block do not actually leave the ready-queue, but stay there and act like proxies, and therefore the number of tasks the scheduler has to deal with is practically the same in all the three cases. It is, however, worth to note that the complexity added for enabling the proxying logic does not impair scheduling performances at noticeable levels.

Lock and Unlock Overheads. We also measured the overhead associated with the slow paths of locking and unlocking operations in the M-BWI code. For the lock

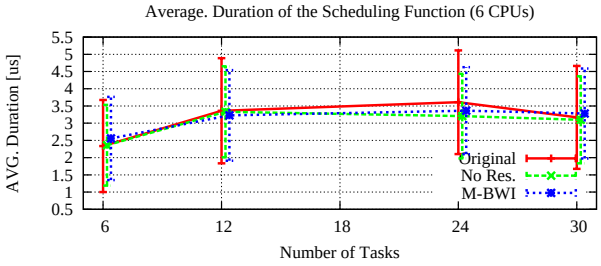


Fig. 14 Average duration of the scheduling function, along with the measured standard deviation (vertical segments).

path, we measured how long it takes, once it has been determined that a resource is busy, to find the proxy and ask the scheduler to execute it. In the unlock path, we measured how long it takes, once it has been determined that there are queued task waiting for the resource to be released, to reset the proxy relationship for the unlocking task and build up a new one for the next owner.

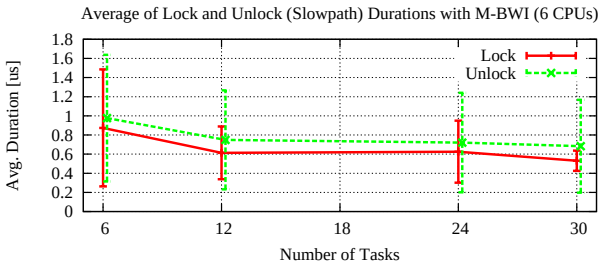


Fig. 15 Average lock and unlock slow paths durations in $LITMUS^{RT}$ with M-BWI (vertical segments highlight the measured standard deviation figures).

Figure 15 shows the average lock and unlock overheads with standard deviations. In general, locking requires less overhead than unlocking. This can be easily understood observing that, in this implementation, a lock operation only has to setup the blocking task as a proxy and then asks the scheduler to put this under operation. Unlocking requires to reset a proxy back to a normal task and finding the new owner of the resource, but also updating the proxying relationship with the new owner in all the tasks that are waiting for the resource and that were proxying the releasing task.

Waiting Times. Figure 16 shows the average and standard deviation of the resource waiting time, i.e., the time interval that elapses from when a task asks to lock a resource and when it actually is granted such permission. In M-BWI, during this time, the task can lie in the ready-queue, preempted by others, it can run and act as a proxy for the lock owner or it can busy wait, if the lock owner is already executing elsewhere. The idea behind this experiment is to show that in average, the delay in

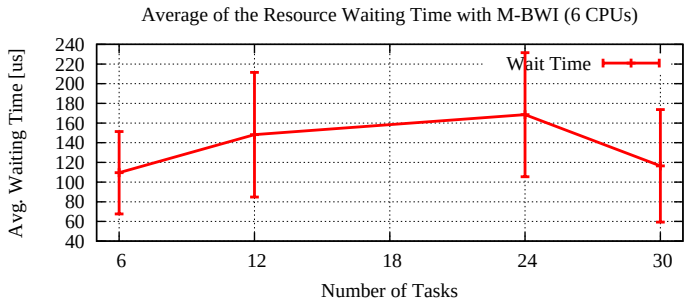


Fig. 16 Average resource waiting time as a function of the number of tasks. The vertical segments denote the measured standard deviation figures.

acquiring the resource is limited. Such information can be useful to soft real-time programmers that can have an idea of the average case in a practical setting.

Figure 16 shows that, on average, waiting for a resource happens for time interval comparable with the length of the critical sections (short ones range from 50 to $200\mu s$, long ones up to $500\mu s$). Obviously there are cases where the resource is available immediately or when the waiting time is large. Consider that, in these experiments, long critical sections were also present, each one of them able to last up to $500\mu s$, which is about the maximum value for the waiting time in the worst possible case. Interestingly, when the number of tasks becomes high enough, the waiting time tends to decrease. This mainly happens because of two reasons: first, it is less likely for many tasks to insist on the same resources; second, it is more likely for resource waiting tasks to have at least one running proxy helping the lock owner in releasing the lock, thus shortening its waiting time.

10 Conclusions and Future Work

In this paper, we presented the Multiprocessor Bandwidth Inheritance (M-BWI) protocol, an extension of BWI to symmetric multiprocessor systems. The protocol guarantees temporal isolation between non-interacting tasks, a property that is useful in open systems, where tasks can join and leave the system at any time. Like the Priority Inheritance Protocol, M-BWI does not require the user to specify any additional parameter, therefore it is readily implementable in real-time operating systems without any special API. We indeed implemented the protocol on the $LITMUS^{RT}$ real-time testbed, and we measured the overhead which is almost negligible for many practical applications. However, it is also possible to perform off-line schedulability analysis: by knowing the task-resource usage and the lengths of the critical sections, it is possible to compute the interference that a task can have on its resource reservation by other interacting tasks.

In the future we want to extend the protocol along different directions. First of all, it would be interesting to provide interference analysis also for partitioned and clustered scheduling algorithms, and compare it against other algorithms like M-SRP

and M-PCP. Also, we would like to implement the Clearing Fund mechanism [45] to return the bandwidth *stolen* by an interfering task to the original server.

Finally, we would like to implement M-BWI on Linux, on top of the SCHED_DEADLINE patch [30], in order to provide support to a wider class of applications.

References

1. Abeni L, Buttazzo G (1998) Integrating multimedia applications in hard real-time systems. In: Proc. IEEE Real-Time Systems Symposium, Madrid, Spain, pp 4–13
2. Anderson JH, Ramamurthy S (1996) A framework for implementing objects and scheduling tasks in lock-free real-time systems. In: Proc. of the IEEE Real-Time Systems Symposium (RTSS), IEEE Computer Society, pp 94–105
3. Behnam M, Shin I, Nolte T, Nolin M (2007) 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
4. Bertogna M, Cirinei M (2007) Response-time analysis for globally scheduled symmetric multiprocessor platforms. In: Proc. of the 28th IEEE Real-Time Systems Symposium (RTSS), Tucson, Arizona (USA)
5. Bertogna M, Checconi F, Faggioli D (2008) Non-Preemptive Access to Shared Resources in Hierarchical Real-Time Systems. In: Proceedings of the 1st Workshop on Compositional Theory and Technology for Real-Time Embedded Systems, Barcelona, Spain
6. Block A, Leontyev H, Brandenburg BB, Anderson JH (2007) A flexible real-time locking protocol for multiprocessors. In: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp 47–56
7. Brandenburg BB, Anderson JH (2007) Feather-trace: A light-weight event tracing toolkit. In: Proc. of the International Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT)
8. Brandenburg BB, Anderson JH (2010) Optimality results for multiprocessor real-time locking. In: Proc. of the IEEE Real-Time Systems Symposium (RTSS), IEEE Computer Society, pp 49–60
9. Brandenburg BB, Anderson JH (2012) The omlp family of optimal multiprocessor real-time locking protocols. Design Automation for Embedded Systems pp 1–66, DOI 10.1007/s10617-012-9090-1, URL `http://dx.doi.org/10.1007/s10617-012-9090-1`
10. Caccamo M, Sha L (2001) Aperiodic servers with resource constraints. In: Proceedings of the 22nd IEEE Real-Time Systems Symposium, (RTSS 2001), pp 161 – 170, DOI 10.1109/REAL.2001.990607
11. Chen CM, Tripathi SK (1994) Multiprocessor priority ceiling based protocols. In: tech. rep., College Park, MD, USA
12. Cho H, Ravindran B, Jensen ED (2007) Space-optimal, wait-free real-time synchronization. IEEE Trans Computers 56(3):373–384

13. Cucinotta T, Checconi F, Abeni L, Palopoli L (2010) Self-tuning schedulers for legacy real-time applications. In: Proceedings of the 5^{th} European Conference on Computer Systems (Eurosys 2010), European chapter of the ACM SIGOPS, Paris, France
14. Davis RI, Burns A (2006) Resource sharing in hierarchical fixed priority preemptive systems. In: Proceedings of the IEEE Real-time Systems Symposium
15. Devi UC, Leontyev H, Anderson JH (2006) Efficient synchronization under global edf scheduling on multiprocessors. In: Proceedings of the 18th Euromicro Conference on Real-Time Systems, pp 75–84
16. Easwaran A, Andersson B (2009) Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In: Proceedings of IEEE Real-Time Systems Symposium
17. Emberson P, Stafford R, Davis R (2010) Techniques for the synthesis of multiprocessor task sets. In: First International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time
18. Faggioli D, Lipari G, Cucinotta T (2008) 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
19. Faggioli D, Lipari G, Cucinotta T (2010) The multiprocessor bandwidth inheritance protocol. In: Proc. of the 22nd Euromicro Conference on Real-Time Systems (ECRTS 2010), pp 90–99
20. Feng X, Mok AK (2002) A model of hierarchical real-time virtual resources. In: Proc. 23^{rd} IEEE Real-Time Systems Symposium, pp 26–35
21. Fisher N, Bertogna M, Baruah S (2007) The design of an EDF-scheduled resource-sharing open environment. In: Proceedings of the 28th IEEE Real-Time System Symposium
22. Gai P, Lipari G, di Natale M (2001) Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In: Proceedings of the IEEE Real-Time Systems Symposium
23. Guan N, Ekberg P, Stigge M, Yi W (2011) Resource sharing protocols for real-time task graph systems. In: Proc. of the 23rd Euromicro Conference on Real-Time Systems (ECRTS 2011), Porto, Portugal
24. Herlihy MP, Wing JM (1990) Linearizability: a correctness condition for concurrent objects. ACM Trans Program Lang Syst 12:463–492, DOI http://doi.acm.org/10.1145/78969.78972, URL `http://doi.acm.org/10.1145/78969.78972`
25. van den Heuvel MM, Bril RJ, Lukkien JJ (2011) Dependable Resource Sharing for Compositional Real-Time Systems. In: 2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications, IEEE, pp 153–163, DOI 10.1109/RTCSA.2011.29
26. Holman P, Anderson JH (2006) Locking under pfair scheduling. ACM Trans Comput Syst 24(2):140–174, DOI 10.1145/1132026.1132028, URL `http://doi.acm.org/10.1145/1132026.1132028`

27. Jansen PG, Mullender SJ, Havinga PJ, Scholten H (2003) Lightweight edf scheduling with deadline inheritance. Tech. Rep. 2003-23, University of Twente, URL http://doc.utwente.nl/41399/
28. Lakshmanan K, de Niz D, Rajkumar R (2009) Coordinated task scheduling, allocation and synchronization on multiprocessors. In: Proceedings of IEEE Real-Time Systems Symposium
29. Lamastra G, Lipari G, Abeni L (2001) A bandwidth inheritance algorithm for real-time task synchronization in open systems. In: Proc. 22nd IEEE Real-Time Systems Symposium
30. Lelli J, Lipari G, Faggioli D, Cucinotta T (2011) An efficient and scalable implementation of global edf in linux. In: Proceedings of the International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)
31. Lipari G, Bini E (2004) A methodology for designing hierarchical scheduling systems. Journal of Embedded Computing 1(2)
32. Lipari G, Lamastra G, Abeni L (2004) Task synchronization in reservation-based real-time systems. IEEE Trans Computers 53(12):1591–1601
33. Lopez JM, Diaz JL, Garcia DF (2004) Utilization bounds for EDF scheduling on real-time multiprocessor systems. In: Real-Time Systems: The International Journal of Time-Critical Computing, vol 28, pp 39–68
34. Macariu G (2011) Limited blocking resource sharing for global multiprocessor scheduling. In: Proc. of the 23rd Euromicro Conference on Real-Time Systems (ECRTS 2011), Porto, Portugal
35. Mancina A, Faggioli D, Lipari G, Herder JN, Gras B, Tanenbaum AS (2009) Enhancing a dependable multiserver operating system with temporal protection via resource reservations. Real-Time Systems 43(2):177–210
36. Nemati F, Behnam M, Nolte T (2009) An investigation of synchronization under multiprocessors hierarchical scheduling. In: Proceedings of the Work-In-Progress (WIP) session of the 21st Euromicro Conference on Real-Time Systems (ECRTS'09), pp 49–52
37. Nemati F, Behnam M, Nolte T (2009) Multiprocessor synchronization and hierarchical scheduling. In: Proceedings of the First International Workshop on Real-time Systems on Multicore Platforms: Theory and Practice (XRTS-2009) in conjunction with ICPP'09
38. Nemati F, Behnam M, Nolte T (2011) Independently-developed real-time systems on multi-cores with shared resources. In: Proc. of the 23rd Euromicro Conference on Real-Time Systems (ECRTS 2011), Porto, Portugal
39. Nemati F, Behnam M, Nolte T (2011) Sharing resources among independently-developed systems on multi-cores. ACM SIGBED Review 8(1)
40. Niz DD, Abeni L, Saewong S, Rajkumar RR (2001) Resource sharing in reservation-based systems. In: In Proceedings of the 22nd IEEE Real-time Systems Symposium, pp 171–180
41. Palopoli L, Abeni L, Cucinotta T, Lipari G, Baruah SK (2008) Weighted feedback reclaiming for multimedia applications. In: Proceedings of the 6^{th} IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia 2008), Atlanta, Georgia, United States, pp 121–126, DOI 10.1109/ESTMED.2008.

4697009
42. Rajkumar R (1990) Real-time synchronization protocols for shared memory multiprocessors. In: Proceedings of the International Conference on Distributed Computing Systems, pp 116–123
43. Rajkumar R, Sha L, Lehoczky J (1988) Real-time synchronization protocols for multiprocessors. In: Proceedings of the Ninth IEEE Real-Time Systems Symposium, pp 259–269
44. Rajkumar R, Juvva K, Molano A, Oikawa S (1998) Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems. In: Proc. Conf. on Multimedia Computing and Networking
45. Santos R, Lipari G, Santos J (2008) Improving the schedulability of soft real-time open dynamic systems: The inheritor is actually a debtor. Journal of Systems and Software 81(7):1093–1104, DOI 10.1016/j.jss.2007.07.004
46. Sha L, Rajkumar R, Lehoczky JP (1990) Priority inheritance protocols: An approach to real-time synchronization. IEEE Transactions on Computers 39(9)
47. Shih I, Lee I (2003) Periodic resource model for compositional real-time guarantees. In: Proc. 24^{th} Real-Time Systems Symposium, pp 2–13
48. Sprunt B, Sha L, Lehoczky J (1989) Aperiodic task scheduling for hard-real-time systems. Journal of Real-Time Systems 1(1):27–60
49. Ward B, Anderson J (2012) Nested multiprocessor real-time locking with improved blocking. In: Proceedings of the 24th Euromicro Conference on Real-Time Systems