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 to symmetric multiprocessor systems. Similarly 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 guarantee a subset of interacting hard real-time tasks. Finally, the M-BWI protocol is neutral to the underlying scheduling policy, because it can be implemented both in global, clustered and partitioned scheduling.

After introducing the M-BWI protocol, in this paper we formally prove its isolation properties, and propose a schedulability analysis for hard real-time tasks. Then, we describe our implementation of the protocol for the $LITMUS^{RT}$ real-time testbed, and measure its overhead. Finally, we compare M-BWI against FMLP, another protocol for resource sharing in multiprocessor systems, introducing a schedulability analysis for M-BWI that proves to be less pessimistic than existing analysis techniques for FMLP.

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

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."

(*) Real-Time Systems Laboratory, Scuola Superiore Sant'Anna
Via G. Moruzzi 1, 56124, Pisa (Italy)
e-mail: {d.faggioli, g.lipari}@sssup.it
(§) Alcatel-Lucent Bell Labs
Blanchardstown Business & Technology Park, Dublin (Ireland)
e-mail: tommaso.cucinotta@alcatel-lucent.com

1 Introduction

Multi-core platforms are being increasingly used in all areas of computing. They constitute a mandatory 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 on 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 [23] of the accesses. On symmetric shared-memory multi-core platforms, a commonly used type of shared resource is an in-memory shared data structure 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, the 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 devise 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 [41] 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 [19, 44, 29], and with tasks that interact with each other using locks [10, 20, 37]. Lamastra et al. proposed the Bandwidth Inheritance (BWI) protocol [27, 30] that combines the Constant Bandwidth Server [1] with Priority Inheritance [43] 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 with both global and 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 [18]. In this extended paper the discussion is more complete and formal; the comparison with the FMLP protocol [6] has been added; evaluation is made through a real implementation of the proposed technique.

[1] More information is available at: `http://www.cs.unc.edu/~anderson/litmus-rt/`.

2 Related Work

Several solutions exist for sharing resources in multiprocessor systems. Most of these have been designed as extensions of uni-processor techniques [40, 39, 11, 31, 21, 26, 16]; fewer have been specifically conceived for multiprocessor systems [15, 6].

The Multiprocessor Priority Ceiling Protocol (MPCP) [40] and its later improvement [39] constitute an adaptation of PCP to work on fixed priority, partitioned multiprocessor scheduling algorithms. A recent variant [26] 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, both Gai et al. [21] and Lopez et al. [31] extended the SRP for partitioned EDF. These papers deal with critical sections shared between tasks running on different processors by means of FIFO-based spin-locks, and forbid 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 priority inheritance-like 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 [9] discuss the definition of blocking time and priority inversion in multi-processor systems, and present optimality results for resource sharing protocols. 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 [32] for global deadline-based schedulers, but this protocol does not support nesting of critical sections.

As it comes to sharing resources in reservation-based hierarchical systems[2], work has been done by Behnam et al. [3] and by Fisher et al. [20]. In both cases, 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. [35, 34, 36] 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 [22] addressed resource sharing in graph-based real-time task models, proposing a new protocol called ACP which tackles the particular issue that

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

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 the 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 [20], 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. [27, 30], and the non-preemptive access to shared resources by Bertogna et al. [5, 24]. 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. [25], 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 [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. The worst-case execution time (without blocking or preemption) of the longest critical section of τ_i on R_k is denoted by $\xi_i(R_k)$, and it is called the *length* of the critical section. The length $\xi_i(R_k)$ includes the duration of all nested critical sections.

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 $R_{i,1}$; 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 within 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 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 "photograph" 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 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 deadlock 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:

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

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 him for the sake of shortening the blocking time due to shared resources. 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 [41, 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 [3]. 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. Typically, when serving sporadic real-time tasks the server maximum budget should not be less than the task WCET, and the server period should not be larger than the task minimum inter-arrival time.

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*.

[3] 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.

Two examples of resource reservation algorithms are the Constant Bandwidth Server (CBS [1]), for dynamic priority scheduling, and the Sporadic Server (SS [45]), 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 [33].

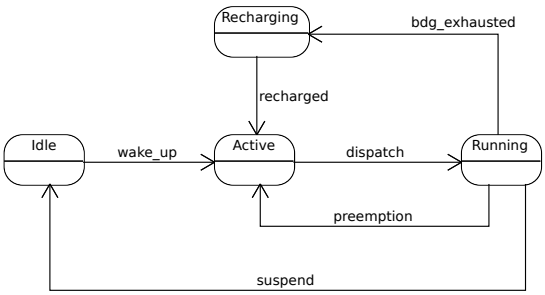


Fig. 1 State machine diagram of a resource reservation server.

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 [30, 17].

For uni-processor systems, the Bandwidth Inheritance Protocol (BWI, see [30]) 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 [30]. In this paper we extend the BWI protocol to the multi-processor case.

In [42], 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 tries to lock a resource whose lock owner is executing on a different processor. It makes no sense to execute the lock owner task on more than one CPU at the same time. However, just blocking the task and suspending the server may create problems to the resource reservation algorithm: as shown in [30], 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 time guarantee to the task.

To solve this problem, M-BWI lets the blocked task perform a busy wait inside its server. However, if the lock owner 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 the lock owner is executed in the server of the blocked task, 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.

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

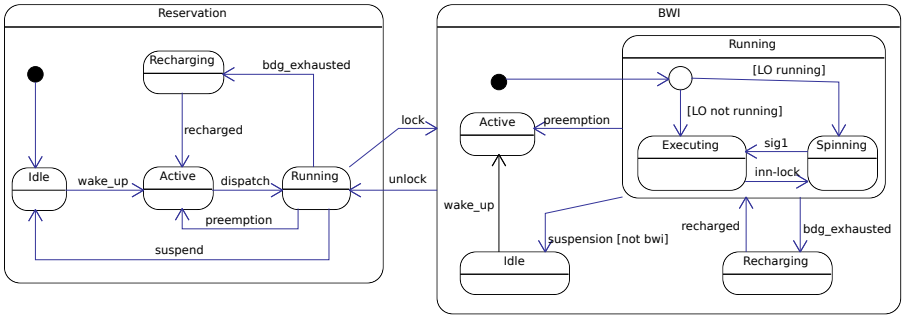


Fig. 2 State machine diagram of a resource reservation server when M-BWI is in place.

the set of all tasks blocked on resource R_k plus 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, `Running` 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 identical 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:

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

  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.

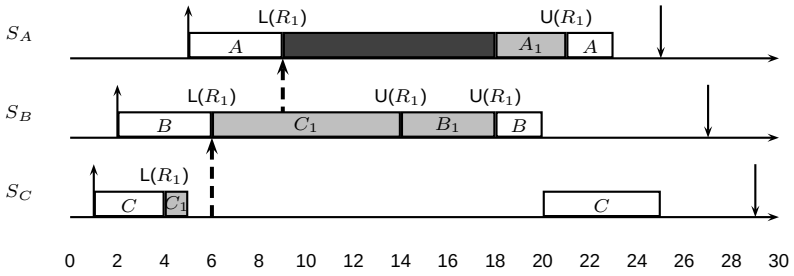


Fig. 3 First example, 3 tasks on 2 CPUs and 1 resource.

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.

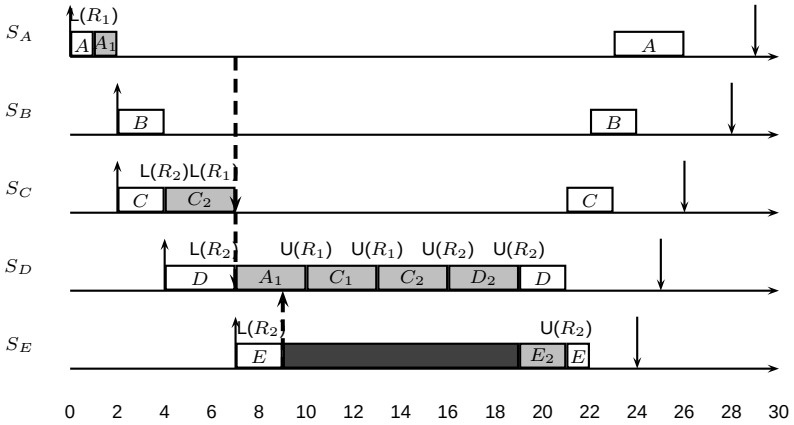


Fig. 4 Second example, 5 tasks on 2 CPUs with 2 resources — task τ_C accesses R_1 inside R_2.

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 $H_{E,A} = \{\tau_E \rightarrow R_2 \rightarrow \tau_C \rightarrow R_1 \rightarrow \tau_A\}$. 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 make any assumption on the scheduling algorithm (fixed or dynamic priority, 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 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 a weaker form of 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 [38, 13] can be used at run-time to appropriately dimension the budgets of the reservations.

6.1 Guarantees for Hard Real-Time Activities

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. In fact, if a set of non hard real-time tasks is allowed to arbitrarily interrupt and block the hard real-time task, the interference time can become very large. Therefore, as in the case of the BWI protocol for single processor systems [30], we assume that all the interfering tasks are themselves hard real-time tasks and will not miss their deadlines.

In this section, we will 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. However the analysis is quite general and can be easily extended to other schedulers and resource reservation algorithms.

Under this assumptions, the following two Lemmas hold.

Lemma 3 *Consider a task τ_i, served by server S_i and its set of interacting tasks Ψ_i. A task $\tau_j \in \Psi_i$ served by server S_j with $P_j > P_i$, can contribute to the interference I_i when S_j is not in* `BWI.Running`.

Proof Tasks τ_j and τ_i interact through at least one resource R_k. This means that there exist a blocking chain from τ_i to τ_j: $\tau_i \rightarrow \ldots \rightarrow R_k \rightarrow \tau_j$.

Under the assumption that all tasks in Ψ_i are hard real-time tasks (i.e. they never miss their deadline), then the following situation may happen: while S_j executes τ_j inside the critical section, it is preempted by S_i which blocks (directly or indirectly) on R_k. Therefore, τ_j inherits S_i and executes inside it, consuming its budget, for the duration of the critical section on R_k.

Notice that, unlike the Priority Inheritance Protocol for single processor systems, task τ_j can interfere with S_i many times on different critical sections even during the same instance.

In fact, consider the case that S_j and S_i are executing on two different processors: there is no rule that prevents the possibility of the two tasks to interfere many times on different critical sections even on the same resource.

To simplify the equation for the interference, we make the assumption that every task accesses a resource R_k with at most one critical section. The following Lemma restricts the number of interfering tasks on a specific resource.

Lemma 4 *Consider a task τ_i, served by server S_i on a system with m processors, and let R_k be a resource on which τ_i may block: $\tau_i \in \Gamma_k$. Then, at most $m-1$ tasks in Γ_k with server period less than P_i contribute to the interference I_i.*

Proof Since servers execute in tasks' deadline order, the running servers will be the m earliest deadline ones, at any given time. Then, the worst possible situation for a task τ_i (attached to S_i) is being one of the running ones, at the moment in which they are all trying to access R_k. Therefore, given the FIFO ordering policy, in the worst case it will have to wait for the other $m-1$ tasks to complete their requests, and suffering for their interference (in terms of busy waiting).

Let $\Psi_i^k = \{\tau_h \in \Psi_i \cap \Gamma_k | P_h > P_i\}$ denote the set of servers with larger period than S_i that can interfere with S_i on R_k. Let also $\Omega_i^k = \{\xi_h(R_k) \mid \tau_h \in \Gamma_k \wedge P_h < P_i\} - \{\xi_i(R_k)\}$ denote the set of maximal critical sections length of tasks interacting with τ_i with server periods smaller than P_i. Given the two Lemmas, the interference a server S_i is subject to, due to M-BWI, can be expressed as follows:

$$\forall R_k \mid \tau_i \in \Gamma_k, I_i^k = \sum_{j|\tau_j \in \Psi_i^k} \xi_j(R_k) + \biguplus^{m-1} \Omega_i^k \tag{4}$$

and

$$I_i = \sum_{k|\tau_i \in \Gamma_k} I_i^k \tag{5}$$

where $\biguplus^n S$ is the sum of the $min(n, \|S\|)$ largest elements of set S (and $\|S\|$ is the number of elements in S).

In open systems it is also possible that hard real-time tasks share some resources with soft real-time ones, e.g., if critical sections are part of a shared library. In this scenario, even if the duration of the critical sections are known in advance, the problem that soft real-time tasks can deplete the budget of their servers — even inside these code segments — has to be taken into account. When this happens, the conditions of Lemma 3 and 4 are no longer verified, and this means that all the potentially interfering tasks must be considered. An upper bound to the interference a server S_i incurs serving a hard task, due to the presence of soft tasks, is:

$$I_i^k = \sum_{j|\tau_j \in \Gamma_k, j \neq i} \xi_j(R_k) \tag{6}$$

If a system consists only of hard real-time tasks, then M-BWI may not the best solution. In fact, other protocols, specifically aimed at this kind of systems, might provide more a precise estimation of blocking times, and thus attain a superior performance. Where M-BWI is, as per the authors' knowledge, really unique, is in heterogeneous environments where temporal isolation is the key feature.

6.2 Examples

For better understanding the schedulability analysis and the interference calculation, the second example of section 5.2 is considered again. Let us list all possible blocking chains:

$$
\begin{aligned}
H_{A,C} &= \{\tau_A \rightarrow R_1 \rightarrow \tau_c\} \\
H_{C,A} &= \{\tau_C \rightarrow R_1 \rightarrow \tau_A\} \\
H_{C,D} &= \{\tau_C \rightarrow R_2 \rightarrow \tau_D\} \\
H_{C,E} &= \{\tau_C \rightarrow R_2 \rightarrow \tau_E\} \\
H_{D,E} &= \{\tau_D \rightarrow R_2 \rightarrow \tau_E\} \\
H_{D,A} &= \{\tau_D \rightarrow R_2 \rightarrow \tau_C \rightarrow R_1 \rightarrow \tau_A\} \\
H_{E,D} &= \{\tau_E \rightarrow R_2 \rightarrow \tau_D\} \\
H_{E,A} &= \{\tau_E \rightarrow R_2 \rightarrow \tau_C \rightarrow R_1 \rightarrow \tau_A\}
\end{aligned}
$$

Let us compute the usage the set Γ for R_1 and R_2: $\Gamma_1 = \{\tau_A, \tau_C, \tau_D, \tau_E\}$ and $\Gamma_2 = \{\tau_C, \tau_D, \tau_E\}$. Let us compute the upper bound on the interference I_C. Following directly from the definitions, $\Psi_C^1 = \{\tau_A\}$ and $\Omega_C^2 = \{\xi_D(R_2), \xi_E(R_2)\}$, the others being empty sets. Therefore, since there are 2 CPUs, and thus only 1 contribution from each Ω has to be considered, and assuming $\xi_D(R_2) > \xi_E(R_2)$, we obtain $I_C = \xi_A(R_1) + \xi_D(R_2)$.

The interference I_E only amounts to the contribution from Ψ_E^1 and Ψ_E^2. In fact, Ω_E^1 and Ω_E^2 are empty, since S_E is the server with the shortest period.

6.3 Remarks

The choice of using FIFO waking order for blocked tasks might be questionable, mainly because it does not reflect the priority/deadline of tasks and servers in the system, as it usually happens in real-time systems and literature.

Using a priority-based wake-up order is certainly possible with the M-BWI protocol. The lemma and theorems presented till now continue to be valid, and in particular the timing isolation property does not depend on the wake-up order. Such a priority-based policy can be useful to reduce the interference time of *important* tasks. However, it comes at the expenses of a larger interference for less-important tasks, and makes the analysis more difficult, as higher priority tasks may interfere more than once on the same critical section. The FIFO policy has at least the interesting property of being starvation free, which also makes it simpler to calculate blocking and interference times. Also, in most cases critical sections are very short, as reported by Brandenburg et al. [7], therefore we expect a limited amount of interference. Notice that the same choice has been made in other protocols, as FMLP [6] and M-SRP [21].

Another important remark to be made concerns with those servers which are busy waiting in `BWI.Running.Spinning` state. Busy waiting is a waste of resources that could be used to execute other tasks. Indeed, it is not difficult to modify the

protocol to reclaim such wasted busy waiting. For example, while the server is in the `BWI.Running.Spinning` state, it could execute other ready tasks. However, we must be careful in doing this, because the reclaimed task might try to access some resource, and this complicates the protocol. Also, the reclaiming task may pollute the cache and increase the computation time of the suspended task. Finally, notice that currently it is not possible to take into account the reclamation in the interference analysis. Therefore, considering that most critical sections are short [7], a reclamation policy may make the protocol unnecessarily complex, without a significant gain.

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 neat 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 wakeup, 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-preemptive 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

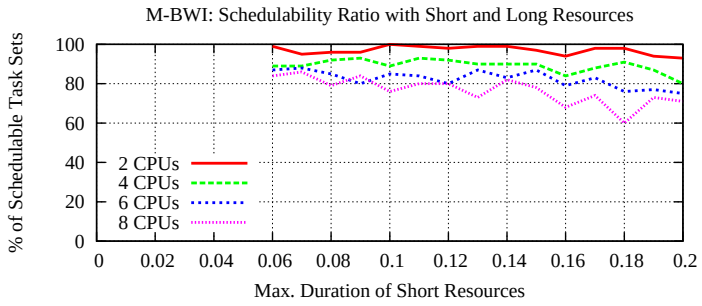
The closed-form expression for the interference time can be used to evaluate how large is the impact of M-BWI on the schedulability of hard real-time tasks in the system. To this end, we performed an analysis of the formula on synthetically generated task sets, and we compared the results against the Flexible Multiprocessor Locking Protocol – FMLP [6], another well-known algorithm for multiprocessor systems.

The task sets were generated according to the following algorithm. A variable number of CPUs $m \in \{2, 4, 6, 8\}$ have been considered. For each value of m, the maximum number of tasks N was set to $N \in \{m, 2 \cdot m, 4 \cdot m, 5 \cdot m\}$, and tasks were added to the set until this limit was reached or their total utilisation exceeded $m/2$. Each task has a processor utilisation chosen uniformly within $(0, U_{max}]$, and a computation time chosen uniformly within $[0.5ms, 500ms)$ (the task period is calculated accordingly). Execution times are inclusive of all the critical sections the tasks access.

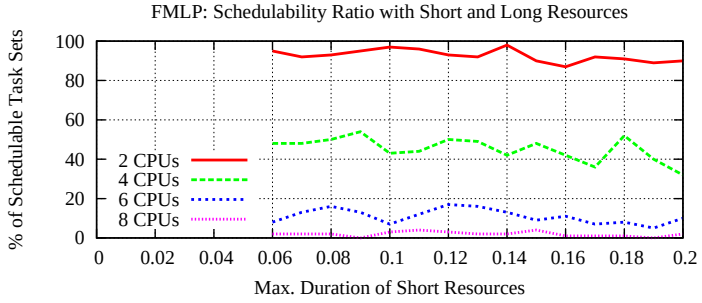
As per the resources, both short and long critical sections have been considered. All critical sections have length between $[60\mu s, 500\mu s]$. We consider *short* critical sections all those sections with length in $[60\mu s, \xi_{max})$, while long ones are within $[\xi_{max}, 500\mu s]$, where ξ_{max} is a parameter of the simulation. We consider that a resource is accessed only through one type of critical section, either short or long; therefore, we denote *short resources* the resources that are accessed only by sort critical sections; and *long resources* the ones that are accessed only by long critical sections. Each task has a probability of accessing 1, 2 or 3 short resources of 0.375, 0.50 and 0.125, respectively. Every long resource (if any) is accessed by 2, 3 or 4 tasks with a probability of 0.125, 0.625 and 0.25, respectively. Finally, for each task and each resource it uses, 1 or 2 nested resources are generated with probability 0.25 and 0.0625. Nested resources are always short. Each resource R_h, nested inside R_k by means of τ_j, will be accessed by other tasks that already use R_k with probability 0.6, and with probability 0.4 from the ones which do not.

We generated 100 task sets for each combination of all these parameters. Then, the interference time was computed according to Equation (5) for M-BWI, whereas for FMLP the blocking time was computed according to the Equations in [6]. In the case of M-BWI, a server has been prepared for each task with budget equal to the task computation time plus its interference, and period equal to the task period. In the case of FMLP, the blocking time was simply added to the task computation time. Finally, the schedulability of the set of servers (and of the set of tasks for FMLP) thus obtained has been checked using the test by Bertogna et al. [4]. We measure the *schedulability ratio*, i.e. the percentage of set of servers that are deemed schedulable against all task sets that are schedulable without considering resource access.

The remainder of this section shows some of the results of these simulations. For all the figures, insets show simulations for different values of U_{max}, varying between 0.2 and 0.8 in steps of 0.2; during each simulation ξ_{max} varied between $0.60\mu s$ and $200\mu s$, in steps of $10\mu s$.



(a) M-BWI



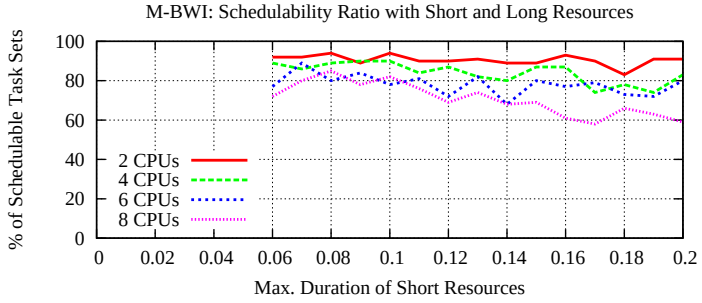
(b) FMLP

Fig. 5 Schedulability ratio for M-BWI and FMLP. Total utilisation is $U_{tot} = 0.4 \cdot m$; number of tasks is $N = 2 \cdot m$, and number of resources is $2 \cdot N$.

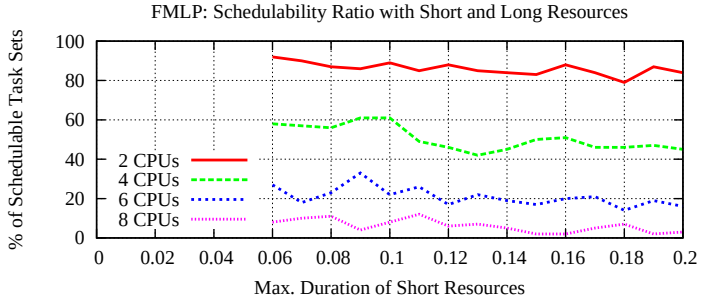
8.1 Experiments with Short and Long Critical Sections

In this first set of experiments both short and long critical sections have been considered. Figure 5 shows the case in which the number of tasks is 4 times the number of processors, the number of resources is twice the number of tasks, and the total utilisation is 0.4 times the number of CPUs.

Clearly, in this case the M-BWI schedulability analysis is more effective than the FMLP analysis. As the number of processors increases, the schedulability of FMLP reaches very low level. The reason is due to the fact that the schedulability test for FMLP has to account also for *blocking time* of all tasks. In fact, FMLP uses blocking and priority inheritance for long resources, and this may cause an *indirect blocking* on all intermediate tasks that are not directly involved in the interaction (see [6] for more details), whereas M-BWI has no notion of priority inversion and there is no indirect blocking. Also, consider that the schedulability analysis used in this paper is rather pessimistic: for FMLP, the blocking time of a task is directly added to its computation time, increasing the overall utilisation. However, by doing so the same blocking time contribution can be accounted for several times. We believe that a tighter schedulability analysis of global EDF with blocking would produce better results for FMLP.



(a) M-BWI



(b) FMLP

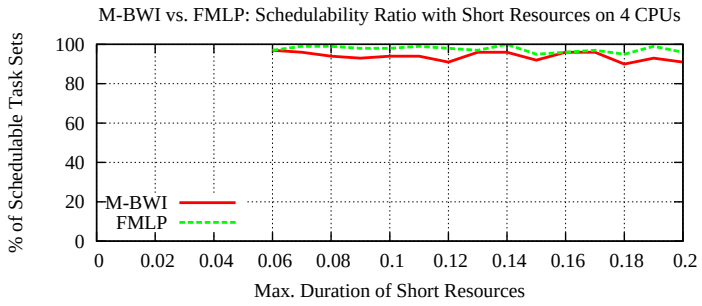
Fig. 6 Schedulability ratio for M-BWI and FMLP. Total utilisation is $U_{tot} = 0.4 \cdot m$; number of tasks is $N = 4 \cdot m$, and number of resources is $4 \cdot N$.

The same pattern repeats also with other combinations of the parameters. We report only another combination of parameters: in Figure 6 we show the results of the analysis when the number of tasks is four times the number of CPUs, the number of resources is equal to the number of tasks, whereas the total utilisation is 0.6 times the number of CPUs.

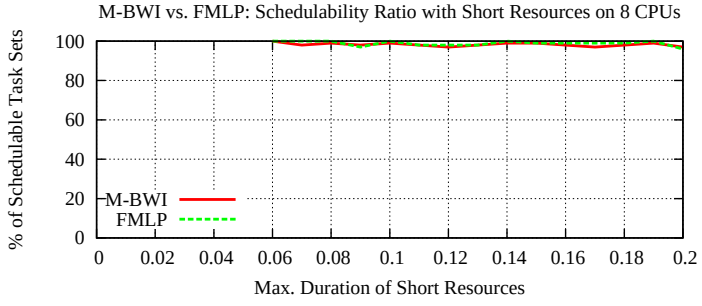
8.2 Experiments with Only Short Resources

Since it is both desirable and common for critical sections to be short, a second set of experiments has been performed where only short resources are used. In this case, FMLP only performs spin-locks plus inheritance, hence the blocked tasks is not suspended.

Figure 7 compares M-BWI and FMLP under two different conditions. In case a), the total utilisation is $U_{tot} = 0.4 \cdot m$, with 16 tasks and 16 resources on $m = 4$ CPUs; in case b), the total utilisation is $U_{tot} = 0.6 \cdot m$ with 32 tasks and 32 resources on $m = 8$ processors. In this case M-BWI performs slightly worse that FMLP. This is probably due to the inheritance mechanisms of FMLP that is more effective is shortening the overall blocking time that M-BWI mechanisms. In any



(a) $m = 4$, $U_{tot} = 0.4 \cdot m$



(b) $m = 8$, $U_{tot} = 0.6 \cdot m$

Fig. 7 Schedulability ratio for M-BWI and FMLP with short resources only. a) 16 tasks and 16 resources; b) 32 tasks and 32 resources.

case, both algorithm show very good performance, with a schedulability close to 100%.

It is important to highlight one final consideration about the length of short critical sections. Brandenburg et al. [7] reported that the length of critical sections in application code and in kernel code is shorter than 5 μsec for more than 90% of the cases. However, by setting $\xi_{max} = 5\mu$sec, our simulations showed a schedulability ratio of 100% in almost all cases. Therefore, for the sake of clarity and to exaggerate the impact of the critical sections on the schedulability analysis, we choose to set the length of short critical sections to be at least 60μsec. However, it is clear from Figure 7 that, even with such a conservative setting, the impact of the critical section length on the schedulability ratio is almost always negligible. Therefore, the use of algorithms such as M-BWI and FMLP is highly recommended for short critical sections.

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 operating system. The aim is to gather insights about how much overhead the protocol entails when

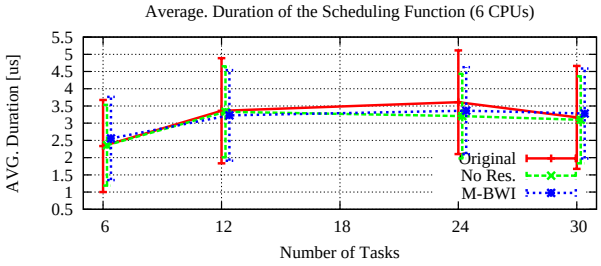


Fig. 8 Average duration of the scheduling function, along with the measured standard deviation (vertical segments).

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 "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 [8]. 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 8 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

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.

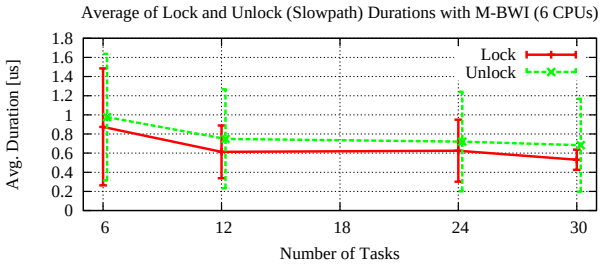


Fig. 9 Average lock and unlock slow paths durations in $LITMUS^{RT}$ with M-BWI (vertical segments highlight the measured standard deviation figures).

Figure 9 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.

It is useful to estimate these two forms of overheads to increase the accuracy of the hard real-time schedulability, including the overheads in the computation times of the task. In particular, we added the lock/unlock overheads for each critical section. After having converted the overheads from CPU cycles to ms, new graphs similar to the ones of section 8 can be produced, to check whether there are situations where the overheads introduced by the protocol impair schedulability. Figure 10 shows this for the case where $N = 2 \cdot m$ and $N_{short} = 2 \cdot N$, with both short and long resources.

Using the maximum value from lock and unlock overhead measurements produces the results shown in Figure 10, which looks identical to Figure 5.a, meaning that the overhead introduced by the proposed implementation of M-BWI does not introduce any further schedulability penalty.

Waiting Times. Figure 11 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 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.

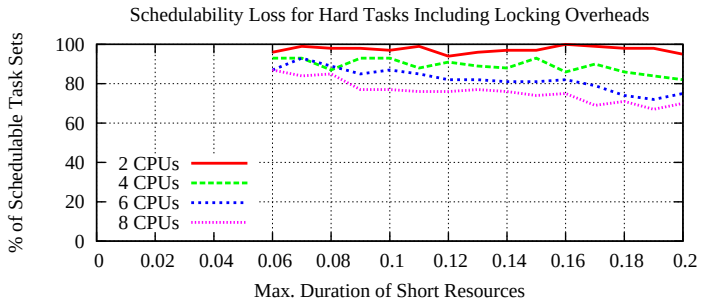


Fig. 10 Schedulability loss with $N = 4 \cdot m$ and $N_{short} = 2 \cdot N$ including the maximum of the overhead of locking and unlocking resources. In this case, both short and long resources were used with durations ranging in $[60\mu s, \xi_{max}]$ and $[\xi_{max}, 500\mu s]$, respectively.

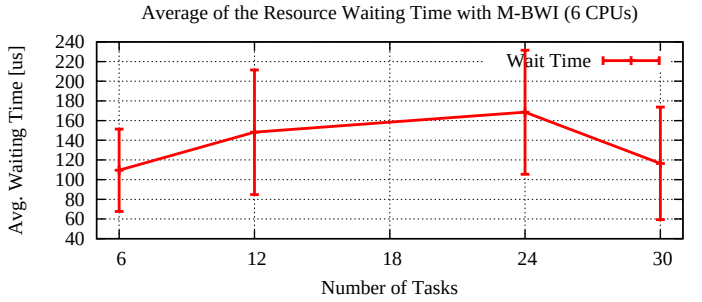


Fig. 11 Average resource waiting time as a function of the number of tasks. The vertical segments denote the measured standard deviation figures.

Figure 11 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 ready to be implemented in real-time operating systems without any special API. We indeed implemented the protocol on the $LITMUS^{RT}$ real-time testbed, and we measure 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 length of the critical sections, it is possible to compute the interference that a task can have on its resource reservation by other interacting tasks. We computed an upper bound on such interference for EDF global scheduling, and we showed that such interference is somehow limited even in the presence of long critical sections.

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 [42] 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 [28], 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 B, Calandrino JM, Block A, Leontyev H, Anderson JH (2008) Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin? In: 2008 IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE, pp 342–353, DOI 10.1109/RTAS.2008.27

8. 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)
9. 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
10. Caccamo M, Sha L (2001) Aperiodic servers with resource constraints. In: Proc. of the IEEE Real Time System Symposium (RTSS), London, UK
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. 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
18. 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
19. 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
20. 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
21. 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
22. 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
23. 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
24. 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
25. 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/
26. Lakshmanan K, de Niz D, Rajkumar R (2009) Coordinated task scheduling, allocation and synchronization on multiprocessors. In: Proceedings of IEEE Real-Time Systems Symposium
27. 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
28. 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)
29. Lipari G, Bini E (2004) A methodology for designing hierarchical scheduling systems. Journal of Embedded Computing 1(2)
30. Lipari G, Lamastra G, Abeni L (2004) Task synchronization in reservation-based real-time systems. IEEE Trans Computers 53(12):1591–1601
31. 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
32. 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
33. 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
34. 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
35. 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
36. 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
37. Nemati F, Behnam M, Nolte T (2011) Sharing resources among independently-developed systems on multi-cores. ACM SIGBED Review 8(1)
38. 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
39. Rajkumar R (1990) Real-time synchronization protocols for shared memory multiprocessors. In: Proceedings of the International Conference on Distributed Computing Systems, pp 116–123
40. 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
41. 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
42. 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
43. Sha L, Rajkumar R, Lehoczky JP (1990) Priority inheritance protocols: An approach to real-time synchronization. IEEE Transactions on Computers 39(9)
44. Shih I, Lee I (2003) Periodic resource model for compositional real-time guarantees. In: Proc. 24^{th} Real-Time Systems Symposium, pp 2–13
45. Sprunt B, Sha L, Lehoczky J (1989) Aperiodic task scheduling for hard-real-time systems. Journal of Real-Time Systems 1(1):27–60