The Multiprocessor BandWidth Inheritance Protocol

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

***Abstract*—In this paper we present the Multiprocessor Bandwidth Inheritance (M-BWI) protocol, an extension of the Bandwidth Inheritance (BWI) protocol to symmetric multiprocessor and multicore systems.**

Similarly to priority inheritance, M-BWI reduces priority inversion in reservation-based scheduling systems; it allows the coexistence of hard, soft and non-real-time tasks; it does not require any information on the temporal parameters of the tasks; hence, it is particularly suitable to open systems, where tasks can dynamically arrive and leave, and their temporal parameters are unknown or only partially known. Moreover, if it is possible to estimate such parameters as the worst-case execution time and the critical sections length, then it is possible to compute an upper bound to the task blocking time. Finally, the M-BWI protocol is neutral to the underlying scheduling scheme, since it can be implemented both in global and partitioned scheduling schemes.

I. Introduction

The wide popularity of multi-core platforms raised the interest of the real-time community for multiprocessor real-time scheduling. Recently, many authors focused their attention on multiprocessor scheduling and scheduling analysis, design methodologies, etc.

When using symmetric shared memory multi-core platforms, one popular programming model is to implement task communication through shared memory variables. To avoid inconsistencies due to concurrency and parallelism, access to shared variables must be protected by an appropriate access scheme. In the literature, many different approaches have been proposed until now, and it is not clear yet which one is going to be used in the future. Examples are *wait-free* [16] and *lock-free* [3] approaches. Recently, hardware supports for *transactional memory* systems have been proposed [34].

However, the most widely used techniques in the programming practice so far are based on *locks*: before accessing a shared memory area, a task must lock a *mutex semaphore* and unlock it after completing the access. The mutex can be locked by only one task at a time; if another tasks tries to lock an already locked semaphore, the task must *wait* for the previous one to unlock it.

In single processor systems, the waiting task is usually *blocked*, and the scheduler chooses a new task to be executed from the ready queue. The blocked task will be unblocked only when the mutex is unlocked by the *owner*. In multi-core systems, it may be useful to let the waiting task execute, performing an idle 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 reactivating 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. The rules of the protocol mandate whether a task blocks or it 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 *waiting-time* or *blocking-time* of a task; 2) off-line, we must be able to bound the waiting-time and include it in a schedulability analysis.

In *open real-time systems*, tasks can dynamically enter or leave the system at any time. Therefore, an admission control is needed to make sure that the new tasks do not jeopardize the schedulability of the already existing tasks. In addition, for robustness, security and safety issues, it is necessary to isolate and protect the temporal behavior 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 [33] have been proved as effective techniques to achieve the goals of temporal isolation and protection and real-time execution. Resource reservation techniques have initially been thought for independent tasks to be executed on single processors. Recently, they were extended to cope with hierarchical scheduling systems [21, 36, 25], and to cope with tasks that interact with each other using shared memory and mutex semaphores [14, 22]. Lamastra et al. proposed the Bandwidth Inheritance (BWI) protocol [26] that combines the Constant Bandwidth Server [1] with Priority Inheritance [35] to achieve bandwidth isolation in open systems.

a) Contributions of this paper: In this paper, we propose Multiprocessor BWI (M-BWI) that extends the BWI protocol to symmetric multiprocessor/multi-core systems. To reduce the task waiting time, the protocol combines busy waiting techniques with blocking and task migration. The protocol allows the coexistence of hard, soft and non-real-time tasks; it 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 guaran-

The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7 under grant agreement n.214777 "IRMOS – Interactive Realtime Multimedia Applications on Service Oriented Infrastructures".

tees for critical tasks: when it is possible to estimate the parameters of the task set, as worst-case execution times and lengths of the critical sections, it is possible to compute an upper bound to the task waiting time.

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

b) Organization of the paper: The reminder of this paper is organized as follows: section II analyzes the existing solutions to real-time multiprocessor synchronization. Section III illustrate the system model and introduces some basic terminology and definitions. Section V gives the details about the new synchronization protocol. Finally, section VI comments the results of the simulations that have been conducted and section VII concludes and foresees some future work.

II. Related Work

Numerous solutions for sharing resources in multiprocessors already exist. Most of these have been designed as extensions of uniprocessor approaches, such as [32, 31, 15, 28, 23, 24, 19]; fewer have been specifically conceived for multiprocessor systems, such as [18, 13].

The Multiprocessor Priority Ceiling Protocol (MPCP) has been proposed in [32], and then improved in [31]. It is an adaptation of PCP to work on fixed priority — partitioned only — multiprocessor scheduling algorithms. Another variant of MPCP has been recently presented in [24]. It is different from the previous ones in the fact that it introduces some "busy waiting". This succeeds in lowering the blocking times of higher priority tasks, but the protocol still addresses only partitioned, fixed priority scheduling.

Chen and Tripathi presented in [15] an extension of PCP, while both Gai et al. in [23] and Lopez et al. in [28] extended the SRP for partitioned EDF. They deal with critical sections shared between tasks running on different processors by means of FIFO-based spin-locks, and forbid their nesting.

As for global scheduling algorithms, Devi et al. proposed in [18] the analysis for non-preemptive execution of global critical sections and FIFO-based wait queues under EDF. Block et al. proposed the FMLP in [13] 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 not avoided in FMLP, but the degree of locking parallelism is reduced by asking the user to group the accesses to shared resources.

Recently, Easwaran and Andersson presented in [19] the generalization of PIP for globally scheduled multiprocessor systems. They also introduced a new solution, which is a tunable adaptation of PCP to such context, with the aim of limiting the number of times a low priority task can block a higher priority one.

As it comes to sharing resources in reservation and hierarchical systems[1], work has been done by Behnam et al. in [8] and by Fisher et al. in [22]. In both cases, a server that has not enough remaining budget to complete a critical section is blocked before entering it, waiting for replenishment. In [17] Davis and Burns propose a generalization of the SRP for hierarchical systems, where they allow servers that are running tasks inside critical sections to overcome their budget limitation.

For all these algorithms, any kind of scheduling analysis is only possible if computation times and critical sections lengths of the tasks are known in advance, which might be not true in an open systems. To the best of the authors' knowledge, the only two attempts to overcome this requirement are the BandWidth Inheritance protocol by Lipari et al. [26], and the non-preemptive access to shared resources by Bertogna et al. [11]. These approaches are well suited for open systems, but are limited to uniprocessors.

Finally, there is work ongoing by Nemati et al. [29, 30] 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. However, they again need full knowledge of all systems parameters, critical sections duration, etc., for the scheduling analysis to be performed.

III. System Model

This paper focuses on shared memory symmetric multiprocessor systems, consisting of m identical unit-capacity processors $P_1, \ldots, P_m$ that share a common memory space. More specifically, *open systems* are considered, where new tasks can dynamically arrive and be admitted into the system, or leave the system at any time. Also, the seamless support for *hard* real-time, *soft* real-time and *non* real-time tasks is among our goals.

A task τ_i is defined as a sequence of jobs $J_{i,j}$ – each job being a sequential piece of work to be executed on one processor at a time. Every job has an arrival time $a_{i,j}$, a computation time $c_{i,j}$ and a finishing time $f_{i,j} \geq a_{i,j} + c_{i,j}$. A task is sporadic if $a_{i,j+1} \geq a_{i,j+1} + T_i$, and T_i is the minimum inter-arrival time (MIT). If $\forall j a_{i,j+1} = a_{i,j} + T_i$ the task is periodic with T_i as its period. Finally, given the worst case execution time (WCET) of τ_i, $C_i = \max_j\{c_{i,j}\}$, its processor utilization U_i is defined as $U_i = \frac{C_i}{T_i}$.
Real-time tasks have a relative deadline D_i and an absolute deadline $d_{i,j} = a_{i,j} + D_i$. A deadline is missed by a job $J_{i,j}$ if $f_{i,j} > d_{i,j}$.

Hard real-time tasks must respect all their deadlines, otherwise their computation cannot be considered as correct. Soft real-time tasks can tolerate occasional and limited violations of their timing constraints, which usually lead to Quality of Service degradation. Non real-time tasks have no particular timing behavior to comply with.

To guarantee a-priori that hard real-time tasks will complete all their jobs before the absolute deadlines, it is necessary

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

to have a-priori information on their temporal behavior, i.e. worst-case execution time and access to shared resources. Given such information, it is possible to do an off-line schedulability analysis. Therefore, in the remainder of the paper it is assumed that we have the correct information on all hard real-time tasks.
For soft real-time and non-real time tasks, instead, no assumption will be made on the knowledge of their temporal behavior.

c) Critical Sections: Concurrently running tasks often need to interact through shared data structures, located in common memory areas. Since an uncontrolled access to these data may result into inconsistent states, they have to be protected by locks (or mutexes). In more detail, when τ_j successfully locks a resource R_l it said to become the lock owner of R_l. If any other task τ_i tries to lock R_l *owned* by τ_j it is blocked on R_l. When then τ_j releases R_l, one of the blocked tasks wakes up and becomes the new owner of R_l.

The code between a lock operation and the corresponding unlock operation on the same resource is called *critical section*. A critical section of task τ_k on resource R_j can be *nested* inside another critical section on a different resource R_h, if the task executes the locking operations in the following order: lock on R_h, lock on R_j, unlock on R_j and unlock on R_h. The worst case execution time (without blocking or preemption) of the longest critical section of τ_k is denoted by $\xi_k(R_j)$, and it is called the *length* of the critical section. The length $\xi_k(R_j)$ includes any nested critical section.

Classical mutexes are prone to unbounded priority inversion [35], which is an harmful phenomenon for real-time activities. Many solutions have been proposed, such as the Priority Inheritance, Priority Ceiling Protocols (PIP,PCP [35]) or the Stack Resource Policy (SRP [7]). In the case of nested critical section, the system can be subject to deadlock, unless a specific protocol is used (as the PCP or the SRP).

d) Multiprocessor Scheduling: The OS scheduler typically assigns priorities to each task and chooses which ones must run on each processor at any given time. In real-time scheduling literature, dynamic and static priority algorithms have been proposed, e.g., Earliest Deadline First and Rate Monotonic (EDF, RM [27]). From a different standpoint, scheduling algorithms can be classified as global or partitioned. Global algorithms use only one queue for all the tasks in the system, while in partitioned algorithms each processor has its own private scheduling queue. More details about achieved results in multiprocessor scheduling can be found in [5, 4, 10, 6, 9, 12].

What is notable to say is that the proposed synchronization mechanism is independent from the specific characteristics of the scheduler, and works with both dynamic and static priority, and under both global and partitioning approaches. Therefore, in the remainder of the paper, it is assumed without loss of generality that the scheduling algorithm is global EDF.

IV. Background

A. Resource Reservation

Resource Reservation has proven to be an effective technique to keep the deadline misses under control in Open Systems [33, 2]. It basically builds up on the concept of *server* as the main schedulable entity. A server S_i has a maximum budget Q_i, a period P_i and a bandwidth $B_i = Q_i/P_i$. Each task τ_i is attached to a server S_i and when the scheduler chooses to run S_i, τ_i is actually executed on that CPU. Typically, the fact that a reserved task τ_i is always able to execute at least Q_i over P_i time intervals is also guaranteed. Therefore, tasks are both confined — i.e., their capability of making their deadlines only depends on their own behavior — and protected against each other — i.e., they always receive their reserved share of the CPU, without any interference from other tasks — and this is called *bandwidth isolation*.

In this work, only the case where each server has one task attached is considered. Situations where more than a task, e.g., an entire application, are scheduled inside a server are deferred to future works.

Two examples of resource reservation algorithms are the Constant Bandwidth Server (CBS [2]), for dynamic priority scheduling, and the Sporadic Server (SS [37]), for fixed priority scheduling. 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 depleted as long as the server is dispatched. A server is `active` whenever its task is ready for execution, the server has some budget left, but some other server is being scheduled. When an active server is dispatched, it becomes `running`, and its served task is able to run. From there on, the server may:

- become `active`, if preempted by another server;
- become `recharging`, if its budget gets depleted;
- become `idle`, if its task blocks or suspends.

On the way out from `recharging` and `idle` many reservation algorithms check whether the budget and the priority/deadline of the server need to be updated.

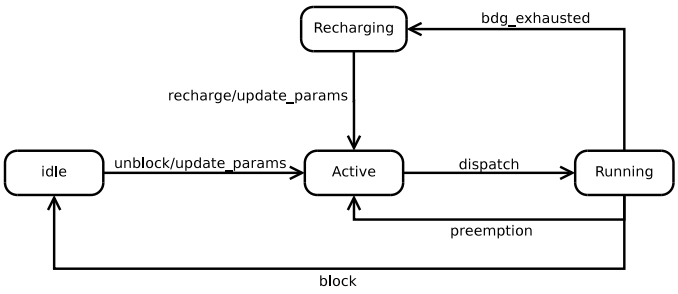


Figure 1: state machine diagram of a resource reservation server.

B. The BandWidth Inheritance Protocol

If tasks share some resources in a reservation based environment, they might start interfering, and the number and

the severity of deadline misses is likely to increase. In fact, a special kind of priority inversion is possible in such a case. However, allowing the lock owner server to overcome its budget, or trying to naively extend traditional protocols might lead to scheduling anomalies, as explained for example in [26, 20].

The BandWidth Inheritance Protocol (BWI, see [26]) already solves this issue for uniprocessor systems, by allowing the tasks that hold some resources to run also in the servers of their lock owners. This helps in anticipating the resource release event, and prevents inversions. A task τ_i that tries to a resource lock R_h either becomes its lock owner or blocks, and some other task τ_j *inherits* S_i. This means τ_j is attached to S_i and thus it is able to run when either S_j (its default server) or S_i is dispatched. Notice that, if a chain of blocked tasks need to be followed to find a non-blocked one this is done. When τ_j later releases R_h, if τ_i takes it, τ_i has to replace τ_j in all the servers it inherited in the meanwhile (S_i excepted).

A blocking chain for a task τ_i is a sequence $\{\tau_1, R_1, \tau_2, \ldots, R_{n-1}, \tau_n\}$ of alternating tasks and resources such that: (i) $\tau_1 = \tau_i$; (ii) both τ_n and τ_{n+1} lock R_n; (iii) each task τ_k (with $1 < k < n$) locks R_k in a critical section nested inside another critical section on R_{k-1}. Proper nested access to critical section is assumed, thus a task never appears more than once in each blocking chain, and deadlock situations are not possible. There might exist more than one blocking chain for a task τ_i, and H_i^h denote the h-th one.

V. Multiprocessor Bandwidth Inheritance

Due to their heterogeneous nature, open systems significantly benefit from multiprocessor support, probably much more than safety critical real-time ones. The BWI protocol is a natural candidate for use in open systems, so it seems natural to use the BWI on multiprocessor systems. Unfortunately, the extension of BWI over multiprocessors is not trivial.

An important problem to be solved is what happens when a task tries to lock an already locked resource, and the lock owner is executing on a different processor. In this case, it makes no sense to attach the lock owner to the server of the blocked task, since it is not possible to execute the same task on two processors at the same time. On the other hand, blocking the task and suspending the server may create problems to the resource reservation algorithm: the suspended server must be treated as if its task were terminated and an unblocking must be treated as a new instance of the server. In this condition, it may be impossible to provide time guarantees to the task, as shown in [26].

In this case, as it will be described later, M-BWI lets the blocked task perform an active waiting inside its server. However, if the lock owner is not executing, because its server has been preempted or exhausted its budget while inside the critical section, the inheritance mechanisms of BWI must still be applied, otherwise the waiting time could be too long. Therefore, it is necessary to understand what is the status of the lock owner before taking a decision on how to resolve the contention. The final choice to be taken is how to order the queue of tasks blocked on a locked resources.

This section gives full details about M-BWI protocol rules and properties.

A. State Machine

A server using the M-BWI protocol has some additional states. The new state machine is depicted in Figure 2. For the sake of completeness, the diagram also considers the events of a task blocking on a non M-BWI mutex, or self-suspending, which are not expanded in the paper for space reasons.

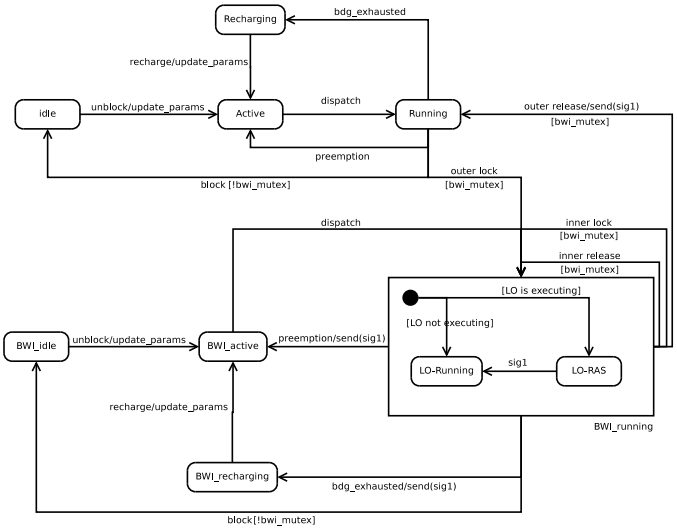


Figure 2: State machine diagram of a resource reservation server when M-BWI is in place.

As long as the task does not try to take a M-BWI lock, the server follows its original behavior. However, when τ_j try to take a lock – whatever it manages or not – its server S_j start behaving as in the bottom part of the diagram. Obviously, the same will happen to the tasks in λ_j (see later) and to their servers.

Some of the new states are replication of their original counterparts, e.g., `recharging` and `BWI_recharging`, and have been added just to make the diagram simpler to understand. This is not true for the `BWI_running` state, which has also been split in two sub-states: `LO-Running`, which stands for Lock Owner Running, and `LO-RAS`, which stands for *Lock Owner Running in Another Server*.

When a server S_j enters state `LO-Running`, it executes a task, either τ_j or the lock owner of the resource upon which τ_j is blocked. If τ_j or its lock owner are already running in some other server (S_r) on a different CPU, S_j enters the `LO-RAS` sub state. A server in this state executes *preemptively* a busy wait until: (i) it is preempted, or (ii) it exhausts its budget. These two events are modelled in the diagram as signals that are broadcasted to all the `LO-RAS` servers, and consumed by only one of them.

B. Protocol Rules

The M-BWI protocol works accordingly to the following blocking and scheduling rules. Let $\lambda_j = \{\tau_k \mid \tau_k \rightarrow \tau_j \text{ for some } h\}$ be the set of tasks blocked on τ_j, and let $\Lambda_j = \{S_k \mid \tau_k \in \lambda_j\} \cup \{S_j\}$ be the set of servers currently inherited by τ_j (S_j included). Then:

M-BWI blocking rule
: When a task τ_i blocks trying to lock an already owned resource R_h, the chain of blocked tasks is followed until one that is not blocked is found – let it be τ_j. Therefore, τ_j inherits S_i and all the servers in Λ_i.

M-BWI scheduling rule I
: When a server $S_k \in \Lambda_j$ is dispatched, it runs the lock owner (τ_j, in the `LO-Running` state). If τ_j is already executing somewhere else, it performs a busy wait (`LO-RAS` state). Whenever S_k is preempted or exhausts the budget while running τ_j, one of the other server that were busy waiting will start executing it.

M-BWI scheduling rule II
: When every $S_l \in \Lambda_j$ blocks on something *not* related to M-BWI, all the servers in Λ_j become idle (`BWI_idle` state). When it unblocks, all $S_l \in \Lambda_j$ become active again (`BWI_active` state).

M-BWI unlocking rule
: When τ_j unlocks R_h and wakes up τ_i, τ_j is discarded from S_i and τ_i replaces it in all $S_l \in \Lambda_j$.

M-BWI waking order
: When more than one task is blocked waiting for locking R_h, access is granted in FIFO ordering, i.e., tasks enters the critical section on R_h according to the order they issued the lock request.

C. Examples

To better explain M-BWI, this section contains two complete examples, conceived to highlight the rules of the protocol.

In the figures below, each time line is a server, and the default task of server S_A is τ_A. 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 that server at that instant. Light gray rectangles are task executing non critical code, dark gray rectangles are critical sections and black rectangles corresponds to the server busy waiting. Which critical section is being executed by which task can again be inferred by the *execution* label, thus A_1 denote task τ_A executing a critical section on resource R_1. Finally, arrows represents "inheritance events", i.e., tasks inheriting servers as consequences of some blockings.

The schedule for the first example is depicted in Figure 3. It consists of 3 tasks accessing only 1 resource, scheduled on 2 processors.

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. Then, when at time 9 τ_A tries also to lock R_1, both τ_C and τ_B inherit S_A, and both S_A and S_B wants to execute τ_C. Therefore, as prescribed by scheduling rule I, one of the two servers has to start busy waiting (S_A in this example). Also, the FIFO wakeup policy is highlighted in this example: when at time 14 τ_C releases R_1, τ_B grabs the lock because it made the locking request before τ_A.

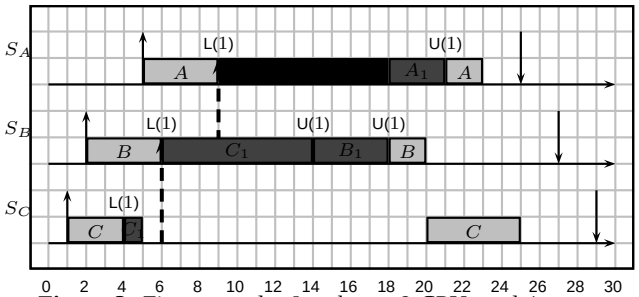


Figure 3: First example, 3 tasks on 2 CPUs and 1 resource.

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

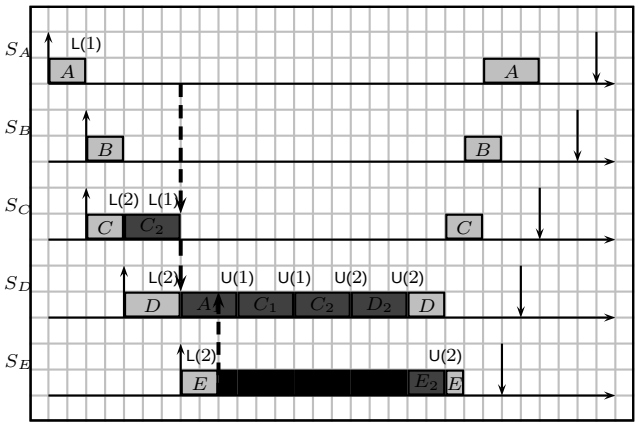


Figure 4: Second example, 5 tasks on 2 CPUs with 2 resources — one accessed nested inside the other by one task.

Notice that both τ_D and τ_E, despite they only use R_2, are blocked by τ_A, which uses only R_1. This is because the behavior of τ_C establishes the blocking chains $H_D = (\tau_D, R_2, \tau_C, R_1, \tau_A)$ and $H_E = (\tau_E, R_2, \tau_C, R_1, \tau_A)$. For the same reason S_D and S_E are subject to the interference either by busy waiting or executing τ_A until it releases R_1.

D. Formal Correctness

In this section formal proofs of the following are given: (i) a task only executes when it is ready, and never in more than one server at a time; (ii) no server misses its scheduling deadline. The former is the basic property for complying with the system model, and proof is given in Lemma 1 and 2. The latter is proven in Theorem 2 and it means that:

1) bandwidth isolation among non interacting tasks attached to servers is always enforced,
2) tasks attached to servers are not automatically guaranteed to meet *their* deadlines. However, as long as it is possible to compute the *interference* of other tasks, hard guarantees can be provisioned.

Thus, if the system is correct and feasible with a resource reservation algorithm of any king, the following lemmas and theorems hold if M-BWI is used on-top of it.

Lemma 1. *M-BWI will never cause a task τ_j to execute on more than one server at the same time.*

Proof: By contradiction. 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 `LO-Running`. However, the scheduling rule I ensures that there is only one of these server in the `LO-Running` state. Here the contradiction, and the lemma follows. ■

Lemma 2. *M-BWI will never cause a blocked or suspended task τ_i to execute in any server.*

Proof: This directly follows from the blocking rule and from scheduling rule II. Suppose the lemma is true when τ_i is not blocked or suspended. According to the blocking rule, if τ_i blocks, its lock owner inherits all the servers in Λ_i. This means it can execute — instead of τ_i — when they are dispatched, and the lemma is still true. Thus, according to scheduling rule I, if τ_i blocks or suspends, all the servers in Λ_i turn `BWI_idle` and can no longer be dispatched and execute τ_i. Hence the lemma. ■

Theorem 1. *An (`BWI_`)`active` or (`BWI_`)`running` server S_i always has in its task list exactly one ready or running task.*

Proof: Suppose initially S_i is (`BWI_`)`active` or (`BWI_`)`running` with only one ready (running) task τ_k attached. It is not important if τ_k is its default task, and the theorem holds.

Task blocking and suspending can decrease the number of ready or running tasks in a server. However, if it reaches zero S_i becomes (`BWI_`)`idle`, and the theorem still holds. On the contrary, task unblocking or resuming always raise the number of ready or running tasks from zero to one, since it must have been preceded by a corresponding blocking or suspending event, and the thesis keeps being respected.

According to the blocking rule, as long as τ_k blocks, its lock owner inherits S_i. τ_k is thus quitting ready state, and its lock owner may be ready, running, blocked or suspended. If it is ready or running, S_i remains `BWI_running`, with such lock owner as the only task to run. If it is blocked or suspended, S_i becomes `BWI_idle`, and in both cases the theorem holds.

Finally, according to the unblocking rule, the unblocking of τ_k — either if τ_k is the default task or a lock owner — turns it back to ready or running state and make S_i discard the former lock owner. Moreover, S_i becomes either `BWI_running` or `running`, with τ_k as the only runnable task, which means the theorem follows. ■

Corollary 1. *There is no way, for a lower priority server S_l to prevent a higher priority server S_h from being dispatched, if is `active` (`BWI_active`), or to continue executing it it is `BWI_running` – i.e., a server never blocks.*

Proof: S_h is a `running` or `BWI_running` server, with one runnable/running task τ_k attached to it. The only means of server blockings to occur are when τ_k blocks, suspends or blocks.

Given Theorem 1, in all such cases S_h either becomes `idle` (`BWI_idle`), or stays `BWI_running` — no matter if in `LO-Running` or in `LO-RAS`. In the former case, there is no blocking involved, since the server scheduler only sees a server deactivation and treat it accordingly. Since, obviously, no blocking is involved in the latter case as well, the corollary follows. ■

Theorem 2. *A server S_i never misses its scheduling deadline.*

Proof: If the set of server is feasible, it means that a scheduling test, among the many existing, has been chosen accordingly with the system configuration, i.e., fixed or dynamic priority and partitioned or global scheduling. This is correct since it has be shown, e.g., in [1], that the resulting schedule of a resource reservation based system is the same as the one of a set of real-time tasks τ_i — one per server S_i — each with bandwidth demand limited by Q_i/P_i.

However, if during the test no blocking time is taken into account, the results are valid only if servers never block. Therefore, the result is valid as long as servers do not block, and given corollary 1 the theorem follows. ■

In an open system, temporal isolation and protection are key features. However, it is also important to be able to bound the blocking time of a task on a resource, so to be able to guarantee task deadlines. In Section V-F, we present a method to compute the interference that tasks impose on servers that execute hard and and soft real-time tasks.

E. M-BWI Design and Implementation Considerations

When more than a task is waiting for a lock, which one has to be woken up when the owner releases the lock is a design choice that deserves some consideration here. The two most widespread alternatives are (i) FIFO and (ii) priority based wakeup. In FIFO, waiting tasks are provided access to the resource according to the order they issued the requests, i.e., it is the task that blocked for first that now grabs the lock. On the other hand, if priority is considered as queueing discipline, the waiting task with the highest priority will be the next lock-owner, independently from when the request was issued. Priority based queue handling is very effecting in guaranteeing the highest priority tasks get quickly the resource their requesting, but this might starve the other tasks and cause their servers an high amount of interference. Hence, due to space constraints, this paper only considers the case lock ownership is granted to blocked tasks in FIFO order. This also make it easier to compare M-BWI with some other important resource-sharing protocol such as MSRP and FMLP, which both use FIFO based locks.

Another important consideration to be made concerns the busy waiting each server performs while in `LO-RAS` state. In fact, the properties of the protocol are enforced as long as a server `LO-RAS` server stays schedulable and depletes its budget while running, but it is not mandatory for it to waste processor time by busy waiting. A smart enough implementation of M-BWI can avoid busy waits and let some

other task run while keeping depleting the LO-RAS server budget. This way many tasks may potentially receive more processing time than it can be expected by off-line system analysis. E.g., non real-time interactive tasks will respond with reduced latency. It is also possible to see this extra time intervals as some sort of reclaiming mechanism made available to even real-time tasks, but precise consideration on how it have to be properly accounted for and how to drawn some benefits –with respect to scheduling analysis– from it are deferred to future research.

F. M-BWI Interference Time Computation

Knowing some information about the tasks in the system, e.g., what tasks access which resources, for how long, etc., an estimation of the interference time I_i each server will incur on can be given. The interference time I_i is defined as the 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 kind of time intervals:

- the ones when tasks other than τ_i executes inside S_i;
- the ones when τ_i is blocked and S_i busy waits in LO-RAS state.

Hence, schedulability guarantees to hard real-time activities in the system are given by Theorem 3.

Theorem 3. *A hard real-time task τ_i, with WCET C_i and MIT T_i attached to a server $S_i = (Q_i = C_i + I_i, P_i = T_i)$ never misses its scheduling deadline.*

Proof: As for Theorem 4 in [26], well known results (e.g., from [1, 2]) ensures that S_i never postpone its deadline if never executing more than Q_i, and this guarantees that τ_i always makes its scheduling deadline. With M-BWI, the budget of S_i can be consumed both by the τ_i up to C_i, and by execution of other tasks and busy waiting up to I_i. Hence the theorem follows. ∎

The set of tasks that are directly or indirectly (i.e., by means of a blocking chain due to critical section nesting) interact with a resource R_j is defined as

$$\Gamma_j = \{\tau_l \mid \exists H_k^h = (\ldots \tau_l \ldots R_j \ldots)\} \tag{1}$$

Theorem 3 also implies that if the system comprises only of hard real-time tasks, servers are scheduled in task's priority order. Thus, as Corollary 1 states that with M-BWI a server never blocks, the m earliest deadline (BWI_)active servers are always executing. Under these conditions, the following two Lemmas hold.

Lemma 3. *For each resource R_j a task $\tau_l \in \Gamma_j$ contributes to the interference to a server S_i if $T_l \leq P_i$.*

Proof: ∎

Lemma 4. *For each resource R_j at most $m-1$ tasks $\tau_l \in \Gamma_j$ with $T_l < P_i$ contribute to the interference on a server S_i.*

Proof: ∎

Let $\Phi_i^j = \{\tau_l \mid \tau_l \in \Gamma_j \wedge \tau_l$ uses $R_j \wedge P_l \geq P_i\} - \{\tau_i\}$ be the set of tasks (attached to servers) with larger period than τ_i (S_i) that can interfere with τ_i (S_i) itself. Let also $\Omega_i^j = \{\xi_l(R_j) \mid \tau_l \in \Gamma_j \wedge P_l < P_i\} - \{\xi_i(R_j)\}$ be the set of maximal critical sections length of tasks interacting with τ_i (attached to servers) with smaller period than τ_i (S_i). Given the two Lemmas, the interference a server S_i is subject due to M-BWI, can be expressed as follows:

$$I_i^j = \sum_{k|\tau_k \in \Phi_i^j} \xi_k(R_j) + \biguplus^{m-1} \Omega_i^j \tag{2}$$

and

$$I_i = \sum_j I_i^j \tag{3}$$

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

In Open Systems it is possible that hard real-time tasks actually share some resources with some soft real-time one, e.g., if critical sections are part of a shared library. In this scenario, even if the duration of the critical section 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. This basically implies that all the potentially interfering tasks should be always considered, since all the assumptions on the deadlines of the server are no longer true. An upper bound to the interference a server S_i — servicing a hard task — incurs on because of the presence of soft tasks is:

$$I_i^j = \sum_{k|\tau_k \in \Gamma_j, k \neq i} \xi_k(R_j) \tag{4}$$

It must be said that if a system consists only of hard real-time tasks, then M-BWI is probably not the best solution. In fact, other protocols –specifically aimed at that– might provide more precise estimation of blocking times that have to be considered in admittance tests. Where M-BWI is –as per the authors' knowledge– really unique, is in heterogeneous environments where isolation is the key feature for making it possible that hard real-time, soft real-time and non real-time tasks to coexist.

VI. Simulation Results

The closed-form expression for the interference time derived above can be used to evaluate how, and under what conditions, the interference that M-BWI introduces affects the schedulability of hard real-time tasks in the system. To this purpose, the effectiveness of the protocol has been evaluated through extensive simulations. Synthetic task sets and shared resources have been generated, according to the following parameters.

Simulations have been carried out for $m = \{2, 4, 8\}$ CPUs. Each time, the maximum number of tasks was set to $N = 5 \cdot m$, and tasks are added to the task set until this limit is reached or their total utilization exceeds $m/2$.

Each task has a processor utilization chosen uniformly within $(0, U_{max}]$, and a computation time chosen uniformly

within $[0.5ms, 500ms)$ (the task period is calculated accordingly). Tasks execution time includes the execution of any critical section it will use.

As per the resources, both short and long critical sections have been considered. Short resources are accessed by critical sections with a duration uniformly chosen within $[10\mu s, \xi_{max})$, while long ones within $[80\mu s, 120\mu s]$. Each task has a probability of accessing 0, 1, 2 or 3 short resources of 0.125, 0.25, 0.50 and 0.125, respectively. On the other hand, each 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 accesses, 1 or 2 nested resources are generated with a probability of 0.25 and 0.0625, respectively. Nested resources are always short and their length is obtained exactly as above. A resource R_h nested inside R_k by means of τ_j is always accessed by τ_j but it may also be accessed by any other task that accesses R_h with probability 0.5.

The results are obtained by generating 1000 task sets for each combination of the parameters of the experiment, and then inflating the computation time of each task by the interference it suffers. After that, checking how many of the generated task sets remained schedulable was done using the response time based test by Bertogna et al. [9].

In the first set of experiments, $N_{short} = N/2$ short resources and $N_{long} = m/2$ long resources have been generated, and then nested requests are added as described. Different simulations have been performed, changing the value of U_{max} among 0.2, 0.4, 0.6 and 0.8, and each of them for $\xi_{max} = 10, 20, 30, 40, 50, 60, 70, 80\mu s$.

Figure 5 shows that in presence of both short and long resources, especially when U_{max} is small (which results in higher number of tasks in the task set), the schedulability loss is significant (insets (a) and (b)). This is due to the accumulation of interference of the pessimistic upper bound. However, it is interesting to see that, if the number of tasks is kept small, even in presence of long resources, of short resources lasting much more than what it is expected from them, and even if individual tasks utilizations are quite high, the loss is about 30% on 8 CPUs, and much better on 4 or 2 CPUs (insets (c) and (d)). This also suggests that if the number of hard real-time activities is small enough – which is common case in open systems – the M-BWI protocol can be used without wasting too much bandwidth.

Nevertheless, given the fact that it is both desirable and common for critical sections to be short, a second set of experiments has been performed where only $N_{short} = N/2$ short resources (and the nested ones generated from them) were used. Again, different runs for the same combinations of values of U_{max} and ξ_{max} as above have been studied. Results in Figure 6 are much more encouraging, since even in worst possible conditions, e.g., many small tasks interacting on resources with high ξ_{max} as depicted in inset (a), the M-BWI protocol only suffers from moderate schedulability loss. Again, if the number of hard real-time interacting tasks is limited, the protocol causes almost no waste of CPU capacity. Moreover, in these cases (insets (c) and (d)), the actual length of the short critical sections does not seem to negatively affect schedulability.

In short, these results reflect what could have been expected from the M-BWI protocol, since it main focus is on providing isolation: the protocol can effectively support a few hard real-time tasks by providing temporal isolation and by bounding the interference time. The protocol is very suited to soft real-time tasks, as we expect that in low contention systems, the average interference time (and thus the overhead of the protocol) is particularly low.

VII. Conclusions and Future Work

In this paper we present the Multiprocessor Bandwidth Inheritance (M-BWI) protocol, an extension of the Bandwidth Inheritance (BWI) protocol to symmetric multiprocessor and multicore systems. The protocol is particularly suitable to open systems, where tasks can enter and leave the system at any time, and hard, soft and non real-time tasks can coexist.

After describing the protocol, we proposed a method to calculate an upper bound to the interference due to blocking on shared resources. Thanks to this upper bound, it is possible to compute the budget to be assigned to hard real-time tasks in order to guarantee they will meet their deadlines in the worst-case.

The proposed upper bound is very pessimistic. As a future work we plan to improve the expression by a more careful analysis. In addition, we plan to implement the algorithm to estimate the average interference time of a task under different operating conditions.

References

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

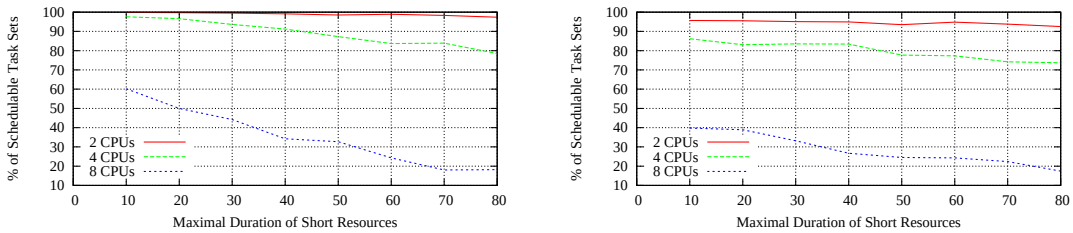
[2] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. IEEE Real-Time Systems Symposium*, pages 4–13, Madrid, Spain, Dec. 1998.

[3] J. H. Anderson and S. Ramamurthy. A framework for implementing objects and scheduling tasks in lock-free real-time systems. In *IEEE Real-Time Systems Symposium*, pages 94–105. IEEE Computer Society, 1996. ISBN 0-8186-7689-2.

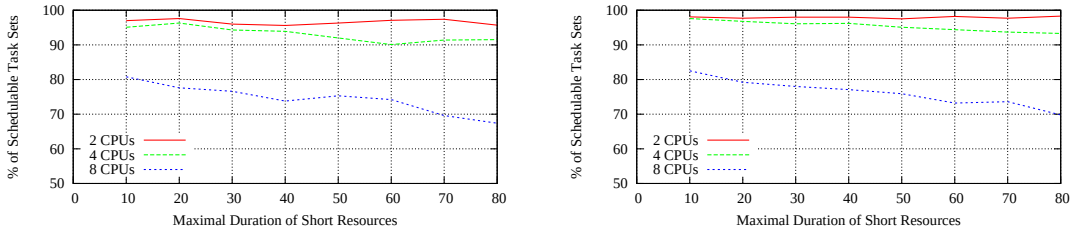
[4] B. Andersson. *Static-Priority Scheduling on Multiprocessors*. PhD thesis, Department of Computer Engineering, Chalmers University, 2003.

[5] B. Andersson, S. Baruah, and J. Jansson. Static-priority scheduling on multiprocessors. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 193–202. IEEE Computer Society Press, December 2001.

[6] T. P. Baker. An analysis of fixed-priority schedulability on a multiprocessor. *Real-Time Systems: The International Journal of Time-Critical Computing*, 32(1–2):49–71, 2006.

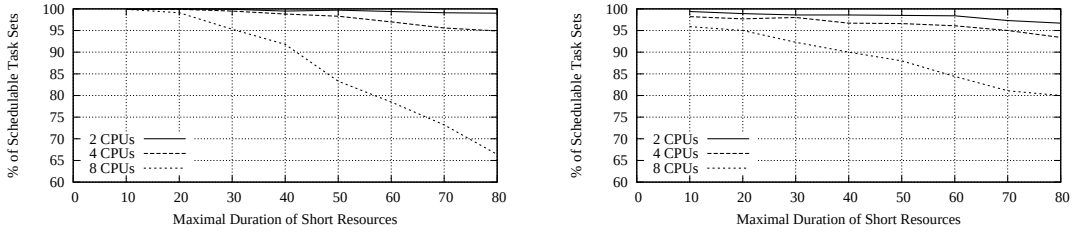


(a) $U_{max} = 0.2$ (b) $U_{max} = 0.4$

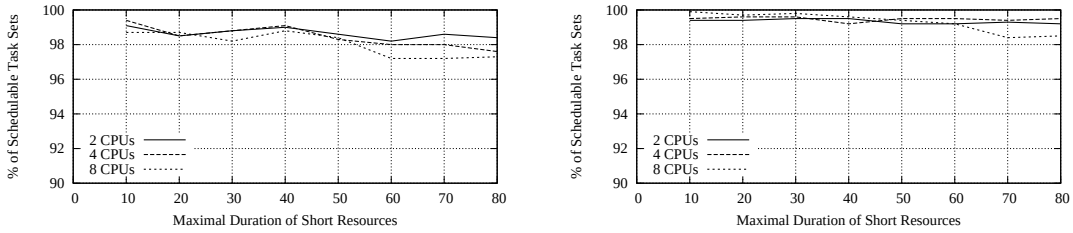


(c) $U_{max} = 0.6$ (d) $U_{max} = 0.8$

Figure 5: Schedulability loss due to M-BWI for hard tasks, varying the maximal duration of short resources. Insets show simulations with different values used for U_{max}. In these experiments, both short and long resources were present.



(a) $U_{max} = 0.2$ (b) $U_{max} = 0.4$



(c) $U_{max} = 0.6$ (d) $U_{max} = 0.8$

Figure 6: Schedulability loss due to M-BWI for hard tasks, varying the maximal duration of short resources. Insets shows simulations with different values for U_{max}. In this experiments, only short resources were present.

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

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

[9] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *28th IEEE Real-Time Systems Symposium (RTSS)*, Tucson, Arizona (USA), 2007.

[10] M. Bertogna, M. Cirinei, and G. Lipari. New schedulability tests for real-time tasks sets scheduled by deadline monotonic on multiprocessors. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, Pisa, Italy, December 2005. IEEE Computer Society Press.

[11] M. Bertogna, F. Checconi, and D. Faggioli. An Implementation of the Earliest Deadline First Algorithm in Linux. In *Proceedings of the 1st Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, Dec. 2008.

[12] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on Parallel and Distributed Systems*, 2008. doi:

http://doi.ieeecomputersociety.org/10.1109/TPDS.2008.129.

[13] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson. 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*, pages 47–56, 2007.

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

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

[16] H. Cho, B. Ravindran, and E. D. Jensen. Space-optimal, wait-free real-time synchronization. *IEEE Trans. Computers*, 56(3):373–384, 2007.

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

[18] U. C. Devi, H. Leontyev, and J. H. Anderson. Efficient synchronization under global edf scheduling on multiprocessors. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 75–84, 2006.

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

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

[21] X. Feng and A. K. Mok. A model of hierarchical real-time virtual resources. In *Proc.* 23^{rd} *IEEE Real-Time Systems Symposium*, pages 26–35, Dec. 2002.

[22] N. Fisher, M. Bertogna, and S. Baruah. The design of an EDF-scheduled resource-sharing open environment. In *Proceedings of the 28th IEEE Real-Time System Symposium*, 2007.

[23] P. Gai, G. Lipari, and M. di Natale. 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*, Dec. 2001.

[24] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Proceedings of IEEE Real-Time Systems Symposium*, 2009.

[25] G. Lipari and E. Bini. A methodology for designing hierarchical scheduling systems. *Journal of Embedded Computing*, 1(2), 2004.

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

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

[28] J. M. Lopez, J. L. Diaz, and D. F. Garcia. Utilization bounds for EDF scheduling on real-time multiprocessor systems. In *Real-Time Systems: The International Journal of Time-Critical Computing*, volume 28, pages 39–68, 2004.

[29] F. Nemati, M. Behnam, and T. Nolte. 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*, September 2009.

[30] F. Nemati, M. Behnam, and T. Nolte. 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)*, pages 49–52, July 2009.

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

[32] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the Ninth IEEE Real-Time Systems Symposium*, pages 259–269, 1988.

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

[34] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In P. B. Gibbons and C. Scheideler, editors, *SPAA*, pages 221–228. ACM, 2007. ISBN 978-1-59593-667-7.

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

[36] I. Shih and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc.* 24^{th} *Real-Time Systems Symposium*, pages 2–13, Dec. 2003.

[37] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Journal of Real-Time Systems*, 1(1):27–60, 1989.