The Multiprocessor BandWidth Inheritance Protocol

Dario Faggioli, Giuseppe Lipari, Tommaso Cucinotta
ReTiS Lab, Scuola Superiore Sant'Anna, CEIICP
Via G. Moruzzi 1, 56124, Pisa (Italy)
e-mail: {d.faggioli, g.lipari, t.cucinotta}@sssup.it

***Abstract*—In this paper, the Multiprocessor Bandwidth Inheritance (M-BWI) protocol is presented, which constitutes 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 the attention on multiprocessor scheduling, analysis and design methodologies.

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* [14] and *lock-free* [2] approaches. Recently, hardware supports for *transactional memory* systems have been proposed [32]. 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 its *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. These rules 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 account for it in a schedulability analysis methodology.

In *open real-time systems*, 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 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 [31] proved themselves as effective techniques to achieve the goals of temporal isolation and protection, and real-time execution. 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, 34, 23], and with tasks that interact with each other using shared memory and mutual exclusion semaphores (mutex) [12, 20]. Lamastra et al. proposed the Bandwidth Inheritance (BWI) protocol [24] that combines the Constant Bandwidth Server [1] with Priority Inheritance [33] to achieve bandwidth isolation in open systems.

A. Contributions of this paper

In this paper, the Multiprocessor BWI (M-BWI) protocol is proposed, that extends the original BWI scheme to symmetric multiprocessor/multicore systems. In order to reduce task waiting times, busy waiting techniques are combined with blocking and task migration, in M-BWI. The protocol allows for 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.

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" and n.248465 "S(o)OS – Service-oriented Operating Systems."

Nevertheless, the protocol supports hard real-time guarantees for critical tasks: when it is possible to estimate such parameters of the task set as the worst-case execution times and the duration of the critical sections, it is possible to compute an upper bound to the task waiting times.

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

B. Organization of the paper

The reminder of this paper is organized as follows: in section II, existing solutions to real-time multiprocessor synchronization are analyzed. In section III, the system model is introduced, along with some basic terminology and definitions, while the needed background concepts are provided in section IV. Details about the new synchronization protocol are provided in section V, while its implications on scheduling analysis are discussed in section VI. Results obtained by simulating the protocol behavior are discussed in section VII. Finally, conclusions are drawn in section VIII, along with possible directions for 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 [30, 29, 13, 26, 21, 22, 17]; fewer have been specifically conceived for multiprocessor systems, such as [16, 11].

The Multiprocessor Priority Ceiling Protocol (MPCP) [30] and its later improvement [29] constitute an adaptation of PCP to work on fixed priority — partitioned only — multiprocessor scheduling algorithms. A recent variant [22] of MPCP differs 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 [13] an extension of PCP, while both Gai et al. [21] and Lopez et al. [26] 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.

Concerning global scheduling algorithms, Devi et al. proposed [16] the analysis for non-preemptive execution of global critical sections and FIFO-based wait queues under EDF. Block et al. proposed the FMLP [11] 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 [17] the generalization of PIP for globally scheduled multiprocessor systems. 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.

As it comes to sharing resources in reservation and hierarchical systems[1], work has been done by Behnam et al. [7] 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, till the replenishment. Davis and Burns proposed [15] a generalization of the SRP for hierarchical systems, where servers that are running tasks inside critical sections are allowed to overcome the budget limit.

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 system. To the best of the authors' knowledge, the only two attempts to overcome this requirement are the BandWidth Inheritance protocol by Lipari et al. [24], and the non-preemptive access to shared resources by Bertogna et al. [9]. These approaches are well suited for open systems, but are limited to uniprocessors.

Finally, there is work ongoing by Nemati et al. [27, 28] 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, in order to perform the scheduling analysis, they again need full knowledge of all system parameters (e.g., critical section durations, etc.).

III. SYSTEM MODEL

In this paper the focus is 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 the goals of M-BWI.

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 $a_{i,j} \geq a_{i,j+1} + T_i$, and T_i is the minimum inter-arrival time (MIT). If $\forall j \; a_{i,j+1} = a_{i,j} + T_i$, then the task is *periodic* with period T_i. Finally, if $C_i = \max_j \{c_{i,j}\}$ is the worst-case execution time (WCET) of τ_i, then 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.

---

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

To guarantee a-priori that hard real-time tasks will complete all their jobs before the absolute deadlines, it is necessary to have a-priori information on their temporal behavior, i.e., their execution times and the shared resources they access. Given such information, it is possible to do an off-line schedulability analysis. Therefore, in the remainder of the paper it is assumed that accurate information on hard real-time tasks is available. For soft real-time and non-real time tasks, instead, no assumption is made on the knowledge of their temporal behavior.

A. Critical Sections

Concurrently running tasks often need to interact through shared data structures, located in common memory areas. Since an uncontrolled access to this 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 is said to become the *lock owner* of R_l. If any other task τ_i tries to lock R_l while it is owned by τ_j, then τ_i blocks on R_l. This is denoted by $\tau_i \rightarrow R_l$. Later, when τ_j releases R_l, one of the blocked tasks wakes up and becomes the new owner of R_l, if any. 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 durations of all nested critical sections, if present.

Classical mutexes are prone to unbounded priority inversion [33], which is an harmful phenomenon for real-time activities. Many solutions have been proposed, such as the Priority Inheritance and Priority Ceiling Protocols (PIP, PCP [33]) or the Stack Resource Policy (SRP [6]). In the case of nested critical sections, the system can be subject to deadlock, unless a specific protocol is used (such as the PCP or the SRP).

B. 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 [25]). 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 [4, 3, 5, 8, 10].

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 Reservations have proven to be effective techniques to keep the deadline misses under control in Open Systems [31, 1]. They basically build 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. 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 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 will be investigated in future work.

Two examples of resource reservation algorithms are the Constant Bandwidth Server (CBS [1]), for dynamic priority scheduling, and the Sporadic Server (SS [35]), 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 actually 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.

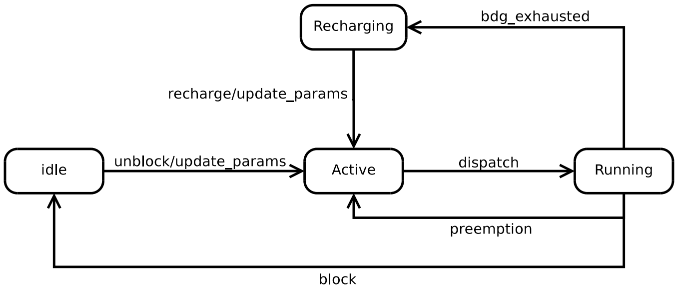
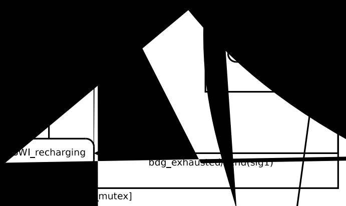


Figure 1. state machine diagram of a resource reservation server.


WI_recharging
bdg_exhausted/ nd(sig1)
[mutex]

the nomenclature, this ending task is sometimes referred to, in what follows, as the "lock owner" for τ_j and R_h. 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 it exhausts its budget; or (ii) its lock owner is preempted or it exhausts its budget. These events are modelled in the diagram as a signal `sig1` that is broadcasted to all the `LO-RAS` servers, and consumed by only one of them.

B. Protocol Rules

The M-BWI protocol works according to the following blocking and scheduling rules. Let λ_j denote the set of tasks blocked waiting for τ_j to release some resource: $\lambda_j = \{\tau_k \mid \tau_k \rightarrow \tau_j\}$. Also, let Λ_j denote the set of servers currently inherited by τ_j (S_j included): $\Lambda_j = \{S_k \mid \tau_k \in \lambda_j\} \cup \{S_j\}$. Then, the protocol rules may be stated as follows:

- **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:** whenever 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 servers that were busy waiting will start executing it.
- **M-BWI scheduling rule II:** whenever a server $S_k \in \Lambda_j$ 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 unblocking rule:** when τ_j unlocks R_h and wakes τ_i up, 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 enter the critical section on R_h according to the order they issued the lock request.

C. Examples

To better explain how M-BWI works, two complete examples are shown in this section, conceived to highlight the rules of the protocol.

In the figures below, each time line represents 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 tasks executing non critical code, dark gray rectangles are critical sections and black 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, 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 accessing only 1 resource, scheduled on 2 processors.

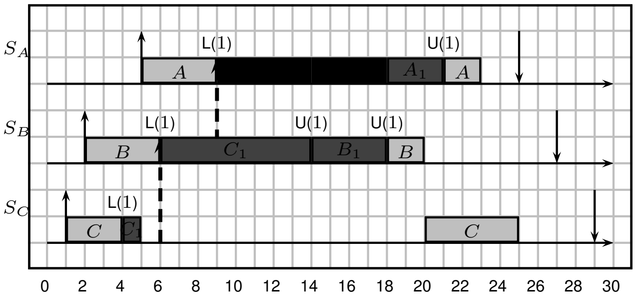


Figure 3. First example, 3 tasks on 2 CPUs and 1 resource.

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. 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 want to execute τ_C. Therefore, as prescribed by the 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.

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.

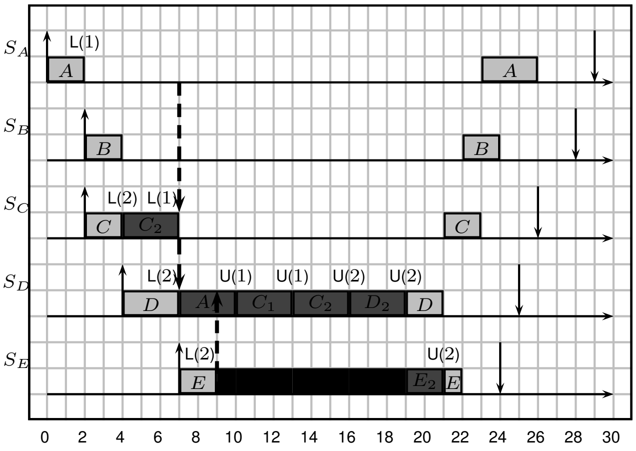


Figure 4. Second example, 5 tasks on 2 CPUs with 2 resources — one accessed nested inside the other by one task.

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 facts 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 kind, then 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 servers 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 become `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 attached 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 for the theorem to hold.

Task blocking and suspending events 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 events 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: With M-BWI a server never blocks.

Proof: A server never blocks if there is no way, for a lower priority server S_l to prevent a higher priority server S_h from being dispatched, if it is (`BWI_`)`active`, or to continue executing if it is `BWI_running`.

In fact, let S_h be a (`BWI_`)`running` server with one runnable/running task τ_k attached to it. The only way a server blocking can occur is if τ_k suspends or blocks.

Given Theorem 1, in all such cases S_h either becomes (`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 treats 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: It has been 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 WCET equal to the reservation budget Q_i and period equal to the reservation P_i. Therefore, feasibility of the set of servers may be verified by exploiting any of the available tests, according to the in-place scheduling algorithm, i.e., fixed or dynamic priority and partitioned or global scheduling.

However, if no blocking times are taken into account in the test, then its outcome is valid only if servers never block. Therefore, given corollary 1, the theorem follows. ■

E. Important Considerations

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/deadline wakeup order might be possible, and its costs and benefits are being studied and will be analyzed in future works. However, FIFO ordering has at least the interesting property of being starvation free, which also makes it simpler to calculate blocking and interference times, and that is why it has been chosen here (as also done in the FMLP or the M-SRP).

Another important consideration to be made regards server busy waiting in `LO-RAS` state. What is important for M-BWI is that a server, while in `LO-RAS` state, (i) stays schedulable and (ii) if running depletes its budget while running. Therefore, wasting processor time by preemptively busy waiting is something that can be avoided. For instance, a smart enough implementation of M-BWI would let some other task run, while keeping depleting the `LO-RAS` server budget. Even from the analysis point of view these "extra time" intervals could be identified and redistributed as a sort of reclaiming mechanisms, to improve the guarantees to hard or soft tasks.

VI. M-BWI SCHEDULABILITY ANALYSIS

A. Isolation for Soft Activities

In an open system, temporal isolation and protection of the different components are key features. M-BWI has been designed exactly for that purpose, i.e., to seamlessly allow accessing critical sections in isolation on multiprocessor environments. The theorems, lemmas and corollaries demonstrated above are all it is necessary to state that bandwidth isolation among the different applications of a reservation-based system is provided by M-BWI not only without additional calculations, but, more importantly, without any need for modifications neither to the scheduler nor to the reservation algorithm.

B. Guarantees for Hard Activities

Open systems may also include hard real-time applications, for which an estimation of the parameters (computation times, critical sections length, etc.) have been performed. For these tasks, it is a must to be able to bound the time they stay blocked on a resource, so that their deadlines may be guaranteed. From the perspective of M-BWI, this can be done computing the interference their server will be subject to.

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 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 `LO-RAS` state.

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

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 [24], well known results (e.g., from [1]) ensure that S_i never postpones its deadline if never executing more than Q_i. This guarantees that τ_i always makes its scheduling deadline. With M-BWI, the budget of S_i can be consumed by execution of both τ_i up to C_i, and by other tasks and busy waiting loops, 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 includes solely 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 $R_j \mid \tau_i \in \Gamma_j$ a task $\tau_l \in \Gamma_j$ with $T_l \geq P_i$ can contribute to the interference on S_i.

Proof: τ_l and τ_i interact with R_j and since they are guaranteed-behavior hard real-time tasks, it is true that $D_i \leq D_l$. Thus, if τ_i blocks on R_j and τ_l is the owner (or even if it issued a request for R_h before τ_i, given the FIFO ordering), it will happen that τ_l inherits S_i, causing interference to it, and never the vice-versa. ■

Lemma 4: For each resource $R_j \mid \tau_i \in \Gamma_j$ 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: 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_j. Therefore, given the FIFO ordering policy, it will in the worst case have to wait for the other $m-1$ tasks to complete their requests, and suffering for their interference (in terms of busy waiting). ■

Let $\Phi_i^j = \{\tau_l \mid \tau_l \in \Gamma_j \wedge P_l \geq P_i\} - \{\tau_i\}$ denote 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 $\Omega_i^j = \{\xi_l(R_j) \mid \tau_l \in \Gamma_j \wedge P_l < P_i\} - \{\xi_i(R_j)\}$ denote 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 to, due to M-BWI, can be expressed as follows:

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

and

$$I_i = \sum_{j|\tau_i \in \Gamma_j} I_i^j \tag{3}$$

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 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 durations 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:

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

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 this kind of systems, might provide more 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 isolation is the key feature for making it possible for hard real-time, soft real-time and non real-time tasks to coexist.

VII. 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 some 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 were added to the task set until this limit was reached or their total utilization exceeded $m/2$.

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. [8].

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} between 0.2 and 0.8 at steps of 0.2, and varying ξ_{max} between $10\mu s$ and $80\mu s$ at steps of $10\mu s$.

Figure 5 shows that in presence of both short and long resources, especially when U_{max} is small (which results in a higher number of tasks), 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, then the loss is about 30% on 8 CPUs, and much better on 4 or 2 CPUs (insets (c) and (d)), despite the presence of long resources, short resources lasting much more than expected, and individual tasks with high utilization. This also suggests that if the number of hard real-time activities is small enough – which is a 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 of a 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 validate the initial expectations on the M-BWI protocol concerning its capability of 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.

VIII. CONCLUSIONS AND FUTURE WORK

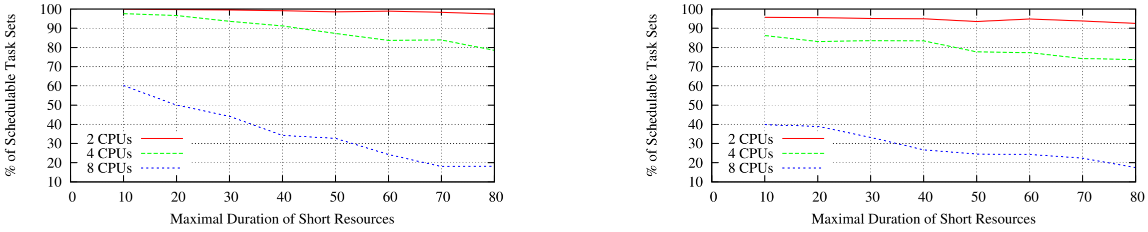
In this paper, the Multiprocessor Bandwidth Inheritance (M-BWI) protocol has been presented, an extension of the Bandwidth Inheritance (BWI) protocol to symmetric multiprocessor and multicore systems. The protocol is particularly suitable to open systems, where tasks can enter and leave the system at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

After describing the protocol, a method to calculate an upper bound to the interference due to blocking on shared resources has been derived. This makes it 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. Also, the schedulability penalty that incurs when taking this interference into account has been evaluated.

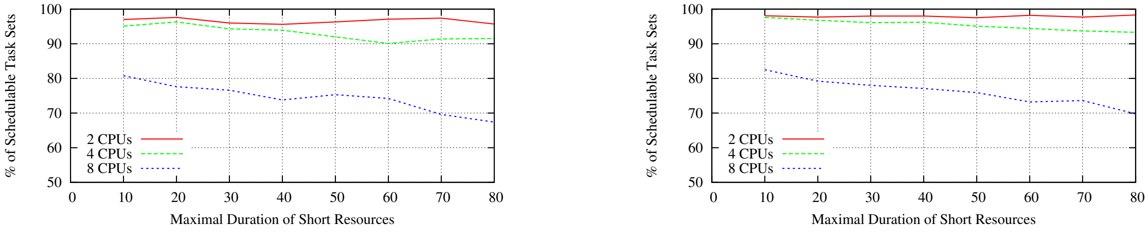
However, the proposed upper bound is very pessimistic, and more careful analysis that may improve its expression is already planned as a future work. In addition, we plan to implement the algorithm to estimate the average interference time of a task under different operating conditions.

REFERENCES

[1] L. Abeni 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.

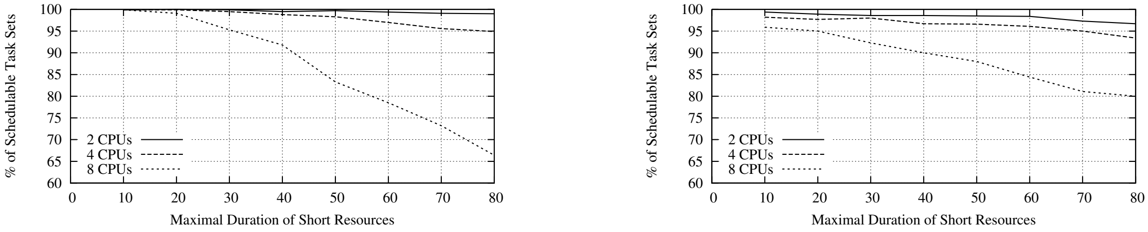


(a) $U_{max} = 0.2$ (b) $U_{max} = 0.4$

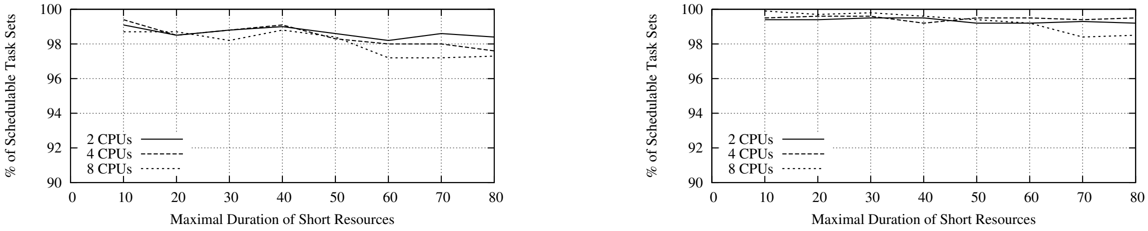


(c) $U_{max} = 0.6$ (d) $U_{max} = 0.8$

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 show simulations with different values for U_{max}. In this experiments, only short resources were present.

[2] 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.

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] P. Gai, G. Lipari, and M. di Natale. Minimizing memory utilization of real-time task sets in single and multiprocessor systems-on-a-chip. In *Proceedings of the IEEE Real-Time Systems Symposium*, Dec. 2001.

[22] 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.

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

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.