Resource Sharing among Prioritized Real-Time Applications on Multiprocessors

Sara Afshar, Nima Khalilzad, Farhang Nemati, Thomas Nolte
Email: sara.afshar@mdh.se
MRTC, Mälardalen University, Sweden

ABSTRACT
In this paper, we propose a new protocol for handling resource sharing among prioritized real-time applications composed on a multiprocessor platform. We propose an optimal priority assignment algorithm which assigns unique priorities to the applications based on information in their interfaces. We have performed experimental evaluations to compare the proposed protocol (called MSOS-Priority) to the current state of the art locking protocols under multiprocessor partitioned scheduling, i.e., MPCP, MSRP, FMLP, MSOS, and OMLP. The evaluations show that MSOS-Priority mostly performs significantly better than alternative approaches.

1. INTRODUCTION
The emergence of multi-core platforms and the fact that they are to be the defacto processors has attracted a lot of interest in the research community regarding multiprocessor software analysis and runtime policies, protocols and techniques.

The industry can benefit from multi-core platforms as these platforms facilitate hardware consolidation by co-executing multiple real-time applications on a shared multi-core platform. The applications may have been developed assuming the existence of various techniques, e.g., relying on a particular scheduling policy. The applications may share mutually exclusive resources. On the other hand, in industry, large and complex systems are commonly divided into several subsystems (components) which are developed in parallel and in isolation. The subsystems will eventually be integrated and co-execute on a multi-core platform. Hence, compositional analysis, i.e., analysis that enables inferring properties of the integrated system from properties of its building components, can facilitate the integration phase.

Two main approaches for scheduling real-time systems on multicores exist; global and partitioned scheduling [1, 2]. Under global scheduling, e.g., Global Earliest Deadline First (G-EDF), tasks are scheduled by a single scheduler and each task can be executed on any processor, i.e., migration of tasks among processors is permitted. Under partitioned scheduling, tasks are statically assigned to processors and tasks within each processor are scheduled by a uniprocessor scheduling protocol, e.g., Rate Monotonic (RM) or Earliest Deadline First (EDF). In this paper we focus on partitioned scheduling where tasks of each application are allocated on a dedicated processor.

In our previous work [3] we proposed a synchronization protocol, called Multiprocessor Synchronization protocol for real-time Open Systems (MSOS), for handling resource sharing among independently-developed real-time applications on multi-core platforms. In an open system, applications can enter and exit during run-time. The schedulability analysis of each application is performed in isolation and its demand for global resources is summarized in a set of requirements which can be used for the global scheduling when co-executing with other applications. Validating these requirements is much easier than performing the whole schedulability analysis. Thus, a run-time admission control program would perform much better when introducing a new application or changing an existing one. The protocol assumes that each real-time application is allocated on a dedicated core. Furthermore, MSOS assumes that the applications have no assigned priority and thus access to shared resources is granted in a FIFO manner. However, to increase schedulability of real-time systems priority assignment is a common solution. One of the objectives of this paper is to extend MSOS to be applicable to prioritized applications accessing mutually exclusive resources.

1.1 Contributions
The main contributions of this paper are as follows: (1) We extend MSOS such that it supports resource sharing among prioritized real-time applications allocated on a shared multi-core platform. For a given real-time application we derive an interface which includes parametric requirements. To distinguish between the two, i.e., the existing MSOS and the new MSOS, we refer them as MSOS-FIFO and MSOS-Priority respectively. (2) We propose an optimal priority assignment algorithm which assigns unique priorities to the applications based on the information specified in their interfaces regarding shared resources. (3) We have performed several experiments to evaluate the performance of MSOS-Priority against MSOS-FIFO as well as the state of the art locking protocols for partitioned scheduling, i.e., MPCP, MSRP, FMLP, and OMLP.

1.2 Related Work
In this section we present a non-exhaustive set of most related synchronization protocols for managing access to mutually exclusive

---

1 This work was partially supported by the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), and Mälardalen Real-Time Research Centre (MRTC)/Mälardalen University.

2 Copyright is held by the authors.
resources on multiprocessors. We specially focus on protocols under partitioned scheduling algorithms.

The existing synchronization protocols can be categorized as suspend-based and spin-based protocols. In suspend-based protocols a task requesting a locked resource keeps the processor and performs spin-lock (busy wait).

**MPCP:** Rajkumar presented MPCP (Multiprocessor Priority Ceiling Protocol) [4] for shared memory multiprocessors hence allowing for synchronization of tasks sharing mutually exclusive resources using partitioned FPS (Fixed Priority Scheduling). MPCP is a suspend-based protocol where tasks waiting for a global resource suspend. A global resource is a resource shared among tasks across processors. Lakshmanan et al. [5] extended a spin-based alternative of MPCP.


**FMLP:** Block et al. [7] presented FMLP (Flexible Multiprocessor Locking Protocol) which is a synchronization protocol for multiprocessors that can be applied to both partitioned and global scheduling algorithms, i.e., P-EDF and G-EDF. Brandenburg and Anderson in [8] extended partitioned FMLP to the fixed priority scheduling policy. Under partitioned FMLP global resources are categorized into long and short resources. Tasks blocked on long resources suspend while tasks blocked on short resources perform busy wait. In an evaluation of partitioned FMLP [9], the authors differentiate between long FMLP and short FMLP where all busy wait. In an evaluation of partitioned FMLP [9], the authors differentiate between long FMLP and short FMLP where all resources suspend while tasks blocked on short resources perform

**OMLP:** Brandenburg and Anderson [10] proposed a new suspend-based locking protocol, called OMLP (On-the-fly Locking Protocol). OMLP is asymptotically optimal (i.e., the total blocking for any task set is a constant factor of blocking that cannot be avoided for some task sets in the worst case) under the suspension-oblivious schedulability analysis where the suspended tasks are assumed to occupy processors and thus blocking is counted as demand. Under partitioned OMLP, a task accessing a global resource cannot be preempted by any task until it releases the resource.

**MSOS:** Recently we presented MSOS [3] which is a suspend-based synchronization protocol for handling resource sharing among real-time applications in an open system on multi-core platforms. MSOS-FIFO assumes that the applications are not assigned any priority and thus applications waiting for a global resource are enqueued in an associated global FIFO-based queue. In this paper we present an alternative of MSOS, called MSOS-Priority to be applicable to prioritized applications when accessing mutually exclusive resources.

In the context of priority assignment, Audsley’s Optimal Priority Assignment (OPA) [11] for priority assignment in uniprocessors is the most related and similar to our priority assignment algorithm. Davis and Burns [12] have shown that OPA can be extended to fixed priority multiprocessor global scheduling if the schedulability of a task does not depend on the priority ordering among higher priority or lower priority tasks. Our proposed algorithm is a generalization of OPA which can be applicable to assigning priorities to applications based on their requirements.

2. TASK AND PLATFORM MODEL

We assume that the multi-core platform is composed of identical, unit-capacity processors with shared memory. Each core contains a different real-time application $A_k(p_k, l_k)$ where $p_k$ is the priority of application $A_k$. Application $A_k$ is represented by an interface $I_k$ which abstracts the information regarding shared resources. Applications may use different scheduling policies. In this paper we focus on schedulability analysis of fixed priority scheduling. From the scheduling point of view our approach can be classified as partitioned scheduling where each application can be seen as a partition (a set of tasks) allocated on one processor.

An application consists of a task set denoted by $\tau_k$, which consists of $n_k$ sporadic tasks, $\tau_i(T_i, C_i, \rho_i, \{S_{i,q,k}\})$ where $T_i$ denotes the minimum inter-arrival time (period) between two successive jobs of task $\tau_i$ with worst-case execution time $C_i$ and $\rho_i$ as its priority. For the sake of simplicity we assume that the tasks have implicit deadlines, i.e., the relative deadline of any job of $\tau_i$ is equal to $T_i$. A task, $\tau_q$, has a higher priority than another task, $\tau_i$, if $\rho_q > \rho_i$. For the sake of simplicity we also assume that each task as well as each application has a unique priority. The tasks in application $A_k$ share a set of resources, $R_{A_k}$, which are protected using semaphores. The set of shared resources $R_{A_k}$ consists of two subsets of different types of resources; local and global resources. A local resource is only shared by tasks in the same application while a global resource is shared by tasks from more than one application. The sets of local and global resources accessed by tasks in application $A_k$ are denoted by $R_{L_k}$ and $R_{G_k}$ respectively. The set of critical sections, in which task $\tau_i$ requests resources in $R_{A_k}$ is denoted by $\{C_{i,q,k}\}$, where $C_{i,q,k}$ is the worst-case execution time of the $p^{th}$ critical section of task $\tau_i$, in which the task locks resource $R_q$. We denote $C_{i,q,k}$ as the worst-case execution time of the longest critical section in which $\tau_i$ requests $R_q$. In the context of requesting resources, when it is said that a task $\tau_i$ is granted access to a resource $R_q$ it means that $R_q$ is available to $\tau_i$, however it does not necessarily mean that $\tau_i$ has started using $R_q$ unless we concretely state that $\tau_i$ is accessing $R_q$ which means that $\tau_i$ has entered its critical section. Furthermore, when we state that access to $R_q$ is granted to $\tau_i$ it implies that $R_q$ is locked by $\tau_i$. Note that, $R_{A_k}$ is the set of resources such as $\{R_{q_1}, R_{q_2}, ..., R_{q_n}\}$ that tasks inside application $A_k$ share. In this paper, we focus on non-nested critical sections. A job of task $\tau_i$ is specified by $J_i$.

3. MSOS-FIFO FOR NON-PRIORITIZED REAL-TIME APPLICATIONS

In this section we briefly present an overview of our synchronization protocol MSOS-FIFO [3] which originally was developed for non-prioritized real-time applications.

3.1 Definitions

3.1.1 Resource Hold Time (RHT)

The RHT of a global resource $R_q$ by task $\tau_i$ in application $A_k$ denoted by $RHT_{q,k,i}$, is the maximum time interval starting from the time instant $\tau_i$ locks $R_q$ and ending at the time instant $\tau_i$ releases $R_q$. Thus, the resource hold time of a global resource, $R_q$, by application $A_k$ denoted by $RHT_{q,k}$, is as follows:

$$RHT_{q,k} = \max \{RHT_{q,k,i} : \tau_i \in A_k\}$$
where $\tau_{q,k}$ is the set of tasks in application $A_k$ sharing $R_q$.

### 3.1.2 Maximum Resource Wait Time

The maximum resource wait time, denoted by $RWT_{q,k}$ for a global resource $R_q$ in application $A_k$, is the worst-case time that any task $\tau_i$ within $A_k$ may wait for other applications on $R_q$ whenever $\tau_i$ requests $R_q$. Under MSOS-FIFO, the applications waiting for a global resource are enqueued in an associated FIFO queue. Hence the worst case occurs when all tasks within other applications have requested $R_q$ before $\tau_i$. As we will see (Section 4), this assumption is not valid for the case when the applications are prioritized.

### 3.1.3 Application Interface

An application, $A_k$, is represented by an interface $I_k(Q_k, Z_k)$ where $Q_k$ represents a set of requirements. An application $A_k$ is schedulable if all the requirements in $Q_k$ are satisfied. A requirement in $Q_k$ is a linear inequality which only depends on the maximum resource wait times of one or more global resources, e.g., $2RWT_{q,k} + 3RWT_{j,k} \leq 18$. The requirements of each application are extracted from its schedulability analysis in isolation. $Z_k$ in the interface represents a set; $Z_k = \{\ldots, Z_l, Z_k, \ldots\}$, where $Z_{q,k}$, called Maximum Application Locking Time (MALT), represents the maximum duration of time that any task $\tau_i$ in any other application $A_l$ ($l \neq k$) may be delayed by tasks in $A_k$ whenever $\tau_i$ requests $R_q$.

### 3.2 General Description of MSOS-FIFO

Access to the local resources is handled by a uniprocessor synchronization protocol, e.g., PCP or SRP. Under MSOS-FIFO each global resource is associated with a global FIFO queue in which applications requesting the resource are enqueued. Within an application the tasks requesting the global resource are enqueued in a local queue; either priority-based or FIFO-based queues. Per each request to a global resource in the application a placeholder for the application is added to the global queue of the resource. When the resource becomes available to the application, i.e., a placeholder of the application is at the head of the global FIFO, the eligible task, e.g., at the top of global FIFO queue, within the application is granted access to the resource.

To decrease interference of applications, they have to release the locked global resources as soon as possible. In other words, the length of resource hold times of global resources have to be as short as possible. This means that a task $\tau_i$ that is granted access to a global resource $R_q$, should not be delayed by any other task $\tau_j$, unless $\tau_j$ holds another global resource. To achieve this, the priority of any task $\tau_i$ within an application $A_k$ requesting a global resource $R_q$ is increased immediately to $\rho_i + \rho_{\max}(A_k)$, where $\rho_{\max}(A_k) = \max \{\rho_i | \tau_i \in A_k\}$. Boosting the priority of $\tau_i$ when it is granted access to a global resource will guarantee that $\tau_i$ can only be delayed or preempted by higher priority tasks executing within a $gcs$. Thus, the RHT of a global resource $R_q$ by a task $\tau_i$ is computed as follows:

$$RHT_{q,k,i} = C_{s_{i,q}} + H_{i,q,k}$$

where $H_{i,q,k} = \sum_{\forall \tau_l \in A_k, \rho_l < \rho_i} C_{s_{l,i}}$ and $R_l \notin R_{A_l,k}^q, l \neq q$.

An application $A_l$ can delay another application $A_k$ on a global resource $R_q$ up to $Z_{q,j}$ time units whenever any task within $A_k$ requests $R_q$. The worst-case waiting time $RWT_{q,k}$ of $A_k$ to wait for $R_q$ whenever any of its tasks requests $R_q$ is calculated as follows:

$$RWT_{q,k} = \sum_{A_l \neq A_k} Z_{q,l}.$$  

In [3] we derived the calculation of $Z_{q,k}$ of a global resource $R_q$ for an application $A_k$, as follows:

**for FIFO-based local queues:**

$$Z_{q,k} = \sum_{\tau_i \in \tau_{q,k}} RHT_{q,k,i}$$

**for Priority-based local queues:**

$$Z_{q,k} = |\tau_{q,k}| \max_{\tau_i \in \tau_{q,k}} \{RHT_{q,k,i}\}$$

where $|\tau_{q,k}|$ is the number of tasks in $A_k$ sharing $R_q$.

Note that, the maximum blocking time on $R_q$ that tasks in $A_k$ can introduce to any task, $\tau_x$, in other applications is when all tasks in $A_k$ request $R_q$ earlier than $\tau_x$. This means that the number of $A_k$'s placeholders in $R_q$'s FIFO queue is equal to the number of tasks in $A_k$ that share $R_q$. This results in $|\tau_{q,k}|$ times as it can be seen in Equation 5.

### 4. MSOS-PRIORITY (MSOS FOR PRIORITIZED REAL-TIME APPLICATIONS)

In this section we present MSOS-Priority for real-time applications with different levels of priorities. The general idea is to prioritize the applications on accessing mutually exclusive global resources. To handle resource access the global queues have to be priority-based. When a global resource becomes available, the highest priority application in the associated global queue is eligible to use the resource. Within an application the tasks requesting a global queue are enqueued in either a priority-based or a FIFO-based local queue. When the highest priority application is granted access to a global resource, the eligible task within the application is granted access to the resource. If multiple requested global resources become available for an application they are accessed in the priority order of their requesting tasks within the application.

A disadvantage argued about spin-based protocols is that the tasks waiting on global resources perform busy wait and hence waste processor time. However, it has been shown [13] that cache-related preemption overhead, depending on the working set size (WSS) of jobs (WSS of a job is the amount of memory that the job needs during its execution) can be significantly large. Thus, performing busy wait in spin-based protocols in some cases benefits the schedulability as they decrease preemptions comparing to suspend-based protocols. As our experimental evaluations show, the larger preemption overheads generally decrease the performance of suspend-based protocols significantly. However, as shown by results of our experiments, MSOS-Priority almost always outperforms all other suspend-based protocols. Furthermore, in many cases MSOS-Priority performs better than spin-based protocols even if the preemption overhead is relatively high.

Under MSOS-FIFO, a lower priority task $\tau_l$ executing within a $gcs$ can be preempted by another higher priority task $\tau_l$ within a $gcs$ if they are accessing different resources. This increases the number of preemptions which adds up the preemption overhead to $gcs$'es and thus making RHT’s longer. To avoid this, we modify this rule
in MSOS-Priority to reduce preemptions. To achieve this the priority of a task \( \tau_i \) accessing a global resource \( R_q \) has to be boosted enough that no other task, even those that are granted access to other global resources can preempt \( \tau_i \).

4.1 Request Rules

Rule 1: Whenever a task \( \tau_i \) in an application \( A_k \) is granted access to a global resource \( R_q \) the priority of \( \tau_i \) is boosted to \( \rho_i + \rho^{\text{max}}(A_k) \). This ensures that if multiple global resources become available to \( A_k \), they are accessed in the order of priorities of tasks requesting them. However, as soon as \( \tau_i \) accesses \( R_q \), i.e., starts using \( R_q \), its priority is further boosted to \( 2 \rho^{\text{max}}(A_k) \) to avoid preemption by other higher priority tasks that are granted access to other global resources. This guarantees continued access to a global resource.

Rule 2: If \( R_q \) is not locked when \( \tau_i \) requests it, \( \tau_i \) is granted access to \( R_q \). If \( R_q \) is locked, \( A_k \) is added to the global priority-based queue of \( R_q \) if \( A_k \) is not already in the queue. \( \tau_i \) is also added to the local queue of \( R_q \) and suspends.

Rule 3: At the time \( R_q \) becomes available to \( A_k \) the eligible task within the local queue of \( R_q \) is granted access to \( R_q \).

Rule 4: When \( \tau_i \) releases \( R_q \), if there is no more tasks in \( A_k \) requesting \( R_q \), i.e., the local queue of \( R_q \) in \( A_k \) is empty, \( A_k \) will be removed from the global queue, otherwise \( A_k \) will remain in the queue. The resource becomes available to the highest priority application in \( R_q \)’s global queue.

5. SCHEDULABILITY ANALYSIS UNDER MSOS-PRIORITY

In this section we derive the schedulability analysis of MSOS-Priority for prioritized applications. Furthermore we describe extraction of interfaces of such applications.

5.1 Computing Resource Hold Times

Similar to Lemma 1 in [3], it can be shown that whenever a task \( \tau_i \) is granted access to a global resource \( R_q \), it can be delayed by at most one gcs per each higher priority task \( \tau_j \) where \( \tau_j \) uses a global resource other than \( R_q \). However, once \( \tau_i \) starts using \( R_q \), no task can preempt it (Rule 1). This avoids preemptions of a task while executing within a gcs.

On the other hand, once a lower priority task \( \tau_k \) starts using a global resource \( R_q \) before \( \tau_i \) is granted access to \( R_q \), \( \tau_k \) will delay \( \tau_i \) as long as \( \tau_i \) is using \( R_q \) because \( \tau_i \) cannot be preempted (Rule 1). It is easy to see that \( \tau_i \) will not anymore be delayed by lower priority tasks that are granted access to global resources other than \( R_q \); whenever \( \tau_i \) is granted access to a global resource \( R_q \), in the worst-case it can be delayed for duration of the largest gcs among all lower priority tasks in which they share global resources other than \( R_q \).

Thus \( RHT_{q,k,i} \) is computed as follows:

\[
RHT_{q,k,i} = Cs_{i,q} + H_{i,q,k} + \max_{\forall \tau_l \in R_{q,k}, \rho_l > \rho_i} \{Cs_{j,l}\}. \tag{6}
\]

5.2 Blocking Times under MSOS-Priority

Under MSOS-Priority, by blocking time we mean delays that any task \( \tau_i \) may incur from local lower priority tasks and as well as from other applications due to mutually exclusive resources in the system. Local tasks of \( \tau_i \) are the tasks that belong to the same application as \( \tau_i \).

Similar to MSOS-FIFO, there are three possible blocking terms that a task \( \tau_i \) may incur. The first and second terms are blocking incurred from the local tasks and are calculated the same way as for MSOS-FIFO [3]. Hence, because of space limitation we skip repeating explanation about how to derive the calculations of the two first blocking terms shown in Equations 7 and 8 respectively. The third blocking term is the delay incurred from other applications and is calculated in a totally different way from that in MSOS-FIFO. The blocking terms are as follows:

5.2.1 Local blocking due to local resources, denoted by \( B_{i,1} \)

This caters for the upper bound for the total blocking time that \( \tau_i \) incurs from lower priority tasks using local resources and is calculated as follows:

\[
B_{i,1} = \min \left\{ n_i^L + 1, \sum_{\rho_j < \rho_i} \left[ T_i/T_j \right] n_j^L(\tau_i) \right\} \max_{\rho_j < \rho_i} \{Cs_{j,q}\} \tag{7}
\]

where \( \text{ceil}(R_i) = \max \{ \rho_j | \tau_i \in \tau_j \} \), \( n_i^L \) is the number of gcs’es of \( \tau_i \) (i.e. the number of \( \tau_i \’s \) requests on global resources), and \( n_j^L(\tau_i) \) is the number of critical sections in which \( \tau_j \) requests local resources with ceiling higher than the priority of \( \tau_i \).

5.2.2 Local blocking due to global resources, denoted by \( B_{i,2} \)

This is the upper bound for the maximum blocking time that \( \tau_i \) incurs from lower priority tasks using global resources and can be calculated as follows:

\[
B_{i,2} = \sum_{\rho_j < \rho_i} \min \left\{ n_i^G + 1, \left[ T_i/T_j \right] n_j^G \right\} \max_{R_q \in R_{A_k}^G} \{Cs_{j,q}\} \tag{8}
\]

Equation 8 contains all the possible delay introduced to the execution of task \( \tau_i \) from all gcs’es of lower priority tasks including gcs’es in which they share a global resource with \( \tau_i \). Task \( \tau_i \) incurs this type of blocking because of priority boosting of lower priority tasks which are granted access to global resources.

5.2.3 Remote blocking, denoted by \( B_{i,3} \)

An application \( A_k \) may introduce different values of remote blocking times to tasks in other applications. We clarify this issue by means of an example:

Example 1: Suppose that a task \( \tau_q \) in an application \( A_k \) requests a global resource \( R_q \) which is already locked by a task within application \( A_k \). In this case \( A_k \) will be added to the global queue of \( R_q \) if the queue does not already contain \( A_k \) (Rule 2). If \( A_k \) has a priority lower than that of \( A_k \), \( A_k \) may lock \( R_q \) it cannot lock \( R_q \) anymore as long as \( A_k \) is in the global queue, i.e., as long as there are more tasks in \( A_k \) requesting \( R_q \). On the other hand if \( A_k \) has a higher priority than \( A_k \), before \( A_k \) is granted access to \( R_q \), it will be blocked by \( A_k \) on \( R_q \) as long as \( A_k \) is in the global queue, i.e., as long as there are tasks in \( A_k \) requesting \( R_q \). In this case the maximum delay that \( \tau_q \) incurs from \( A_k \) during \( \tau_q \’s \) period is a function of the maximum number of requests from \( A_k \) to \( R_q \) during \( T_i \).

Thus the amount of remote blocking introduced by \( A_k \) to any task
Whenever any task $\tau_i$ in application $A_k$ requests a global resource $R_q$, it may be delayed by multiple jobs of each task within a higher priority application that request $R_q$. All these jobs requesting $R_q$ will be granted access to $R_q$ before $\tau_i$. The maximum delay that $\tau_i$ incurs from these jobs in any time interval $t$ is a function of the maximum number of them executing during $t$.

**Definition 1.** Maximum Application Locking Time (MALT), denoted by $Z_{q,k}(t)$, represents the maximum delay any task $\tau_i$ in any lower priority application $A_k$ may incur from tasks in $A_k$ during time interval $t$, each time $\tau_i$ requests resource $R_q$.

In order to calculate the total execution of all critical sections of all tasks in application $A_k$ in which they use global resource $R_q$ during time interval $t$, we first need to calculate the total execution (workload) of all critical sections of each individual task in $A_k$ in which it requests $R_q$ during $t$. The maximum number of jobs generated by task $\tau_j$ during time interval $t$ equals $\lceil \frac{t}{\tau_j} \rceil + 1$. On the other hand, whenever a job $J_j$ of $\tau_j$ locks $R_q$ it holds $R_q$ for at most $RHT_{q,k,j}$ time units. $J_j$ may lock $R_q$ at most $n_{j,q}^G$ times where $n_{j,q}^G$ is the maximum number of requests to $R_q$ issued by any job of $\tau_j$. Thus, the total workload of all critical sections of $\tau_j$ locking $R_q$ during time interval $t$ is denoted by $W_{j,t}(R_q)$ and is computed as follows:

$$W_{j,t}(R_q) = (\lceil \frac{t}{\tau_j} \rceil + 1) n_{j,q}^G RHT_{q,k,j}. \quad (9)$$

Now we can compute the maximum application locking time $Z_{q,k}(t)$ that is introduced by tasks in $A_k$ to any task sharing global resource $R_q$ in any lower priority application:

$$Z_{q,k}(t) = \sum_{\tau_j \in \tau_{q,k}} W_{j,t}(R_q). \quad (10)$$

Equation 10 can be computed in isolation and without requiring any information from other applications because the only variable is $t$ and other parameters, e.g., $RHT_{q,k,j}$, are constants which means they are calculated using only local information. Thus, $Z_{q,k}(t)$ remains as a function of only $t$.

**Definition 2.** The Maximum Resource Wait Time (RWT) for a global resource $R_q$ incurred by task $\tau_i$ in application $A_k$, denoted by $RWT_{q,k,i}(t)$, is the maximum duration of time that $\tau_i$ may wait for remote applications on resource $R_q$ during any time interval $t$.

A RWT under MSOS-Priority, considering delays from lower priority applications (Equation 10) and higher priority applications (Equation 10), can be calculated as follows:

$$RWT_{q,k,i}(t) = \sum_{\rho_{A_k} < \rho_{A_{i}}} Z_{q,i}(t) + n_{i,q}^G \max_{\rho_{A_k} > \rho_{A_{i}}} \{RHT_{q,i}\}. \quad (11)$$

Under MSOS-FIFO, a RWT for a global resource is a constant value which is the same for any task sharing the resource. However, a RWT under MSOS-Priority is a function of time interval $t$ and may differ for different tasks.

**Computing Remote Blocking:** Equation 11 can be used to compute remote blocking $B_{i,3}$ for task $\tau_i$. Based on Lemma 1 the maximum delay introduced by lower priority applications on a global resource $R_q$ to any task requesting $R_q$ is the same for all the tasks. Thus, regardless of the type of the local queues (FIFO-based or priority-based) the second term in the computation of $RWT_{q,k,i}$, shown in Equation 11, is the same for all tasks requesting $R_q$. The first term is also independent of the type of local queues as the total interference from higher priority applications during the period of each task is the same for both types of local queues. Hence, despite of the type of local queues, $B_{i,3}$ can be calculated as follows:

$$B_{i,3} = \sum_{\forall R_q \in \mathcal{R}_{G}} RWT_{q,k,i}. \quad (12)$$

### 5.3 Interface

The interface of an application $A_k$ has to contain information regarding global resources which is required for schedulability analysis when the applications co-execute on a multi-core platform. It has to contain the requirements that have to be satisfied for $A_k$ to be schedulable. Furthermore, the interface has to provide information required by other applications sharing resources with $A_k$.

Looking at Equation 11, the calculation of the RWT of a task $\tau_i$ in application $A_k$, for a global resource $R_q$, requires MALT’s, $Z_{q,k}(t)$, from higher priority applications as well as RHT’s, $RHT_{q,k,i}$, from lower priority applications. This means that to be able to calculate the RWT’s, the interfaces of the applications have to provide both RHT’s and MALT’s for global resources they share. Thus the interface of an application $A_k$ is represented by $I_k(Q_k, Z_k, RHT)$ where $Q_k$ represents a set of requirements, $Z_k$ is a set of MALT’s and a RHT is a function of time interval $t$. MALT’s in the interface of application $A_k$ are needed for calculating the total delay introduced by $A_k$ to lower priority applications sharing resources with $A_k$. RHT in the interface is a set of RHT’s of global resources shared by application $A_k$. RHT’s are needed for calculating the total delay introduced by $A_k$ to higher priority applications.

#### 5.3.1 Extracting the Requirements

The requirements in the interface of an application under MSOS-Priority are extracted similar to MSOS-FIFO [3]. The difference is that RWT’s under MSOS-Priority may have different value for each task.

Starting from the schedulability condition of $\tau_i$, the maximum value of blocking time $D_{i}^{\text{max}}$ that $\tau_i$ can tolerate without missing its deadline can be calculated as follows:

$$0 < \exists t \leq T_i rbf_{EP}(i,t) \leq t, \quad (13)$$
where \( \text{rbf}_{FP}(i, t) \) denotes request bound function of \( \tau_i \) which computes the maximum cumulative execution requests that could be generated from the time that \( \tau_i \) is released up to time \( t \), and is computed as follows:

\[
\text{rbf}_{FP}(i, t) = C_i + B_i + \sum_{\rho_i < \rho_j} ([t/T_j] | C_j).
\]

By substituting \( B_i \) by \( B_i^{\text{max}} \) in Equations 13 and 14, \( B_i^{\text{max}} \) can be calculated as follows:

\[
B_i^{\text{max}} = \max_{0 \leq t \leq T_i} (t - (C_i + \sum_{\rho_i < \rho_j} ([t/T_j] | C_j))).
\]

The total blocking of task \( \tau_i \) is the summation of three blocking terms calculated in Section 5.2:

\[
B_i = B_{i,1} + B_{i,2} + B_{i,3}.
\]

Since \( B_{i,1} \) and \( B_{i,2} \) totally depend on internal factors, i.e., the parameters from the application that \( \tau_i \) belongs to, they are considered as constant values, i.e., they depend on only internal factors of \( \tau_i \)'s application. Thus, Equation 16 can be rewritten as follows:

\[
B_i = \gamma_i + \sum_{\forall R_k \in R_{G_k}^q} \sum_{\tau_k \in \tau_{G_k}} \text{RWT}_{q,k,i}
\]

where \( \gamma_i = B_{i,1} + B_{i,2} \).

Equation 17 shows that the total blocking time of task \( \tau_i \) is a function of maximum resource wait times of \( \tau_i \) for the global resources accessed by \( \tau_i \). With the achieved \( B_i^{\text{max}} \) and Equation 17 a requirement can be extracted:

\[
\gamma_i + \sum_{\forall R_k \in R_{G_k}^q} \sum_{\tau_k \in \tau_{G_k}} \text{RWT}_{q,k,i} \leq B_i^{\text{max}}
\]

or

\[
\sum_{\forall R_k \in R_{G_k}^q} \sum_{\tau_k \in \tau_{G_k}} \text{RWT}_{q,k,i} \leq B_i^{\text{max}} - \gamma_i
\]

5.3.2 Global Schedulability Test

The schedulability of each application is tested by validating its requirements. Any application \( A_k \) is schedulable if all its requirements in \( Q_k \) are satisfied. Validating the requirements in \( Q_k \) requires maximum resource wait times, e.g., \( \text{RWT}_{q,k,i} \), of global resources accessed by tasks within \( A_k \) which are calculated using Equation 11.

One can see that most of the calculations in the schedulability analysis of applications can be performed off-line and in isolation. The global schedulability analysis remains as testing a set of requirements which are in form of linear inequalities. This makes MSOS an appropriate synchronization protocol for open systems on multi-cores where applications can enter during run-time. An admission control program can easily test the schedulability of the system by revalidating the requirements in the interfaces.

As shown in Section 5.3.1, in an application each task sharing global resources produces one requirement, i.e., the number of requirements in the application’s interface equals to the number of its tasks sharing global resources. In the worst-case all tasks in all applications share global resources. The global schedulability test requires that all requirements in all applications are validated. Thus, given that \( B_i^{\text{max}} \) and \( \gamma_i \) are calculated in advance, the complexity of an interface-based admission test is of \( O(m \times n) \) where \( m \) is the number of applications and \( n \) is the number of tasks per application.

6. OPTIMAL ALGORITHM FOR ASSIGNING PRIORITIES TO APPLICATIONS

MSOS-Priority has the potential to increase the schedulability if appropriate priorities are assigned to the applications. In this section, to assign unique priorities to the applications, we propose an optimal algorithm similar to the algorithm presented by Davis and Burns [12]. The algorithm is based on the interface-based schedulability test which only requires information in the interfaces. The algorithm is optimal in the sense that if it fails to assign priorities to applications, any hypothetically optimal algorithm will also fail. The pseudo code of the algorithm is shown in Figure 1.

```plaintext
1. List remainedAppList ← all applications sharing resources;
2. for each application A in remainedAppList
3.   A.priority ← 0;
4. end for
5. while (remainedAppList is not empty)
6.   List ScheduledApps ← {};
7.   List NotScheduledApps ← {};
8. for each application A in remainedAppList
9.   if all requirements of A are satisfied
10.      add A to ScheduledApps;
11.     else
12.        add A to NotScheduledApps;
13.      end if
14. end for
15. if ScheduledApps is empty
16.   return fail;
17. remainedAppList ← NotScheduledApps;
18. for each application A in remainedAppList
19.   A.priority ← A.priority + (number of applications in ScheduledApps);
20. end for
21. incr ← 0;
22. for each application A in ScheduledApps
23.   A.priority ← A.priority + incr;
24. incr ← incr + 1;
25. end for
26. end while
27. return succeed;
```

**Figure 1:** The priority assignment algorithm

Initially all applications are assigned lowest priority, i.e., 0 (Line 3). The algorithm tries to, in an iterative way, increase the priority of applications. In each stage it leaves the applications that are schedulable (Line 10) and increases the priority of not schedulable applications (the for-loop in Line 18). The priority of all unschedulable applications is increased by the number of the schedulable applications in the current stage (Line 19). If there are more than one schedulable applications in the current stage, their priorities are increased in a way that each application gets a unique priority: the first application’s priority is increased by 0, the second’s is increased by 1, the third’s is increased by 2, etc (the for-loop in Line 22). When testing the schedulability of an application \( A_k \), the algorithm assumes that all the applications that have the same priority as \( A_k \) are higher priority applications. This assumption helps to test if \( A_k \) can tolerate all the remaining applications if they get a higher priority than \( A_k \). Thus, when calculating RWT’s based on Equation 11 the algorithm changes condition \( p_{Ak} < p_{Ai} \) in the first term to \( p_{Ak} \leq p_{Ai} \). Figure 2 illustrates an example of the algorithm. In the example shown in Figure 2, there are four applications sharing resources. The algorithm succeeds to assign...
priorities to them in three stages. First the algorithm gives the lowest priority to them, i.e., \( \rho_{A_k} = 0 \) for each application. In this stage the algorithm realizes that applications \( A_1 \) and \( A_3 \) are schedulable but \( A_2 \) and \( A_4 \) are not schedulable, thus the priority of \( A_2 \) and \( A_4 \) are increased by 2 which is the number of schedulable applications, i.e., \( A_1 \) and \( A_3 \). Both \( A_1 \) and \( A_3 \) are schedulable, hence to assign unique priorities, the algorithm increases the priority of \( A_1 \) and \( A_3 \) by 0 and 1 respectively. Please notice that increasing the priority of the schedulable applications can be done in any order since their schedulability has been tested assuming that all the other ones have higher priority. Thus the order in which the priorities of these applications are increased will not make any of them unschedulable. In the second stage, only applications \( A_2 \) and \( A_4 \) are remained. At this stage the algorithm finds that \( A_4 \) is not schedulable, hence its priority has to be increased. In the last stage, \( A_4 \) also becomes schedulable and since all applications are now schedulable the algorithm succeeds. If at any stage the algorithm cannot find any schedulable application, meaning that none of the remaining applications can tolerate the other ones to have higher priorities, the algorithm fails.

In Audsley’s priority assignment algorithm [11] to find a solution (if any) at most \( m(m+1)/2 \) schedulability tests will be performed where \( m \) is the number of tasks to be prioritized. Similarly, in our algorithm to find a solution (if any), in the worst case at each stage only one application is schedulable and is assigned a priority. In the next stage the schedulability of all the remaining applications has to be performed again. In this case after the algorithm is finished, the schedulability test for the applications with priority \( m, m-1, \ldots, 2 \) has been performed \( m, m-1, \ldots, 2 \) times respectively, and hence the maximum number of schedulability tests is \( m(m+1)/2 \) where \( m \) is the number of applications to be prioritized.

**Lemma 2.** The priority assignment algorithm under the proposed resource sharing structure assumptions is optimal, i.e., if the algorithm fails to assign unique priorities any hypothetically optimal algorithm under the same scheduling assumptions will also fail.

**Proof.** We assume that the priority assignment algorithm at some stage fails. Let assume that it fails at stage \( f \) (\( 1 < f \leq m \) where \( m \) is the number of applications), i.e., no schedulable application was found among the remaining unschedulable ones at stage \( f \). We assume that at stage \( f - 1 \) \( k \) applications were schedulable, thus at stage \( f \) there are \( m - k \) applications that became unschedulable with the priority level \( k \) (assuming the lowest priority level as zero). By algorithm definition, at stage \( f \) for each application in the set \( m - k \) applications, the algorithm assumes that other applications in this set have higher priorities, even though they have been assigned the same priority level as \( k \). Therefore, increasing their priority level will not make them schedulable. On the other hand, at stage \( f - 1 \), these \( m - k \) tasks with priority levels lower than that of \( k \) were unschedulable so that the algorithm continued to stage \( f \). Therefore, lowering the priority of these \( m - k \) tasks also will not make them schedulable. This means that, no other algorithm that has the same system assumptions can assign a priority level to these set of applications to make the system schedulable. This finishes the proof.

7. **Schedulability Tests Extended with Preemption Overhead**

If the tasks allocated on a processor do not share resources, since any job can preempt at most one job during its execution, it suffices to inflate the worst-case execution time of each task by one preemption overhead [14]. This type of preemption which originates from different priority levels of tasks is common under all synchronization protocols discussed in this paper, hence, we assume that this overhead is already inflated in the worst-case execution times. When tasks share local resources under the control of a uniprocessor synchronization protocol, e.g., PCP, an additional preemption overhead has to be added to the worst-case execution times. We assume that the worst-case execution times are also inflated with this preemption overhead as the synchronization protocols under partitioned scheduling algorithms generally assume reusing a uniprocessor synchronization protocol for handling local resources.

However, when tasks share global resources, depending on the synchronization protocol used, the preemption overhead may not be the same for different protocols.

7.1 **Local Preemption Overhead**

Under a suspend-based protocol, e.g., MSOS-Priority, PCP, whenever a task \( \tau_i \) requests a global resource if the resource is locked by a task in a remote processor (application), \( \tau_i \) suspends. While \( \tau_i \) is suspending, lower priority tasks can execute and request global resources as well. Later on when \( \tau_i \) is resumed and finishes using the global resource, it can be preempted by those lower priority tasks when they are granted access to their requested global resources. Each lower priority task \( \tau_j \) can preempt \( \tau_i \) up to \( \left\lfloor \frac{\tau_i}{\tau_j} \right\rfloor \) times. On the other hand \( \tau_i \) cannot preempt \( \tau_j \) more than \( \tau_i/\tau_j + 1 \) times. Thus, \( \tau_i \) can be preempted by any lower priority task \( \tau_j \) at most \( \min\{\tau_i/\tau_j + 1, \left\lfloor \frac{\tau_i}{\tau_j} \right\rfloor \} \) times.

Task \( \tau_i \) may also experience extra preemptions from higher priority tasks requesting global resources. Whenever a higher priority task \( \tau_h \) requests a global resource which is locked by remote tasks, it suspends and thus \( \tau_h \) has the chance to execute. When \( \tau_h \) is granted access to the resource it will preempt \( \tau_i \). This may happen up to \( \left\lfloor \frac{\tau_i}{\tau_h} \right\rfloor \) times.

Thus, the total number of extra preemptions that a task \( \tau_i \) may experience from local tasks, because of suspension on global resources, is denoted by \( L_{preem_l} \) and is calculated as follows:

\[
L_{preem_l} = \sum_{\rho_h < \rho_i} \min\{\tau_i/\tau_j + 1, \left\lfloor \frac{\tau_i}{\tau_j} \right\rfloor \} + \sum_{\rho_h > \rho_i} \left\lfloor \frac{\tau_i}{\tau_h} \right\rfloor.
\]  

(20)

The preemption overhead in Equation 20 is due to suspension of tasks while they are waiting for global resources. Spin-based protocols as well as OMLP do not suffer from this preemption overhead.

![Figure 2: Illustrative example for the priority assignment algorithm](image-url)
at all as they do not let a task suspend while waiting for a global resource.

7.2 Remote Preemption Overhead

Besides the preemption overhead a task \( \tau_i \) may incur from local tasks, it may incur preemption overhead propagated from tasks on remote processors/applications. Under a synchronization protocol, when a task \( \tau_i \) is allowed to be preempted while it is using a global resource \( R_q \), i.e., \( \tau_i \) is within a gcs, the preemption overhead will make the critical section longer which in turn makes remote tasks wait longer for \( R_q \). The more preemptions \( \tau_i \) can experience within a gcs the more remote preemption overhead it will introduce to the remote tasks. FMLP, OMLP and MSOS-Priority do not let a task using a global resource be preempted, i.e., tasks execute non-preemptively within a gcs, therefore they are free from remote preemption overhead. However, under MPCP and MSOS-FIFO a task within a gcs can be preempted by other tasks within gcs es and thus remote preemption overhead has to be included in their schedulability tests. Under MPCP, a task within a gcs can be preempted by gcs es from both lower priority and higher priority tasks [4]. Under MSOS-FIFO a task within a gcs can only be preempted by higher priority tasks within their gcs es. Under both MPCP and MSOS-FIFO a gcs of a task \( \tau_i \), in which it accesses a global resource \( R_q \) can be preempted by at most one gcs per each task \( \tau_j \) in which it accesses a global resource other than \( R_q \). This is because the preempting task \( \tau_j \) will not have chance to execute and enter another gcs before \( \tau_i \) releases \( R_q \). The reason is that the priority of a task within a gcs is boosted to be higher than any priority of the local tasks.

Under MPCP the priority of a gcs of a task \( \tau_i \), in which it requests a global resource \( R_q \) is boosted to its remote ceiling which is the summation of the highest priority of any remote task that may request \( R_q \) and the highest priority in the local processor plus one. Thus under MPCP, a gcs can be preempted by any gcs with a higher remote ceiling. Consequently, under MPCP the maximum number of preemptions a gcs of \( \tau_i \) may incur, equals to the maximum number of tasks containing a gcs with a higher remote ceiling. On the other hand, under MSOS-FIFO a gcs of \( \tau_i \) in which it requests a global resource \( R_q \) can only be preempted by gcs es of higher priority tasks in which they access a resource other than \( R_q \). Thus, under MSOS-FIFO the maximum number of preemptions a gcs of \( \tau_i \) in which it access \( R_q \), may incur equals to the maximum number of higher priority tasks with a gcs in which they access any global resource other than \( R_q \).

The length of gcs es has to be inflated by the preemption overhead they may incur. This means gcs es become longer and under MSOS-FIFO it leads to longer RHT’s.

8. EXPERIMENTAL EVALUATION

In this section we present our experimental evaluations for comparison of MSOS-Priority to other synchronization protocols under the fixed priority partitioned scheduling algorithm. We compared the performance of protocols with regard to the schedulability of protocols using response time analysis. We have evaluated suspend-based as well as spin-based protocols. All spin-based synchronization protocols perform the same with respect to global resources, because in all of them, a task waiting for a global resource performs busy wait. Thus the blocking times in those protocols are the same. We present the results of the spin-based protocols in one group and represent the protocols by SPIN. In this category we put MSRP, FMLP (short resources), as well as a version of MSOS-FIFO in which tasks waiting for global resources perform busy wait. However, the suspend-based protocols, i.e., MSOS-Priority, MPCP, FMLP (long resources), OMLP and MSOS-FIFO perform differently in different situations and thus we present their performance individually.

8.1 Experiment Setup

We determined the performance of the protocols based on the schedulability of randomly generated task sets under each protocol. The tasks within each task set allocated on each processor were generated based on parameters as follows. The utilization of each task was randomly chosen between 0.01 and 0.1, and its period was randomly chosen between 10ms and 100ms. The execution time of each task was calculated based on its utilization and period. For each processor, tasks were generated until the utilization of the tasks reached a cap or a maximum number of 30 tasks were generated. The utilization cap was randomly chosen from \{0.3, 0.4, 0.5\}.

The number of global resources shared among all tasks was 10. The number of critical sections per each task was randomly chosen between 1 and 6. The length of each critical section was randomly chosen between 5\( \mu \)s and 225\( \mu \)s with steps of 20\( \mu \)s, i.e., 5, 25, 45, etc.

Preemption overhead: The preemption overhead that we chose was inspired by measurements done by Bastoni et al. in [13] where they measured the cache-related preemption overhead as a function of WSS of tasks. To cover a broad range of overhead, i.e., from very low (or no) per-preemption overhead to very high per-preemption overhead, for each task set the per-preemption overhead was randomly chosen (in \( \mu \)s) from \{0, 20, 60, 140, 300, 620, 1260, 2540\}. This covers preemption overhead for tasks with very small WSS, e.g., 4 kilobytes, as well as tasks with very large WSS, e.g., around 4 megabytes.

We generated 1 million task sets. In the generated task sets the number of task sets were between 115 and 215 for each setting, where the number of settings was 6336. We repeated the experiments three times and we did not observe any significant difference in the obtained results. This means that 1 million randomly generated samples can be representative for our settings.

8.2 Results

The results of our experiments show that different synchronization protocols can be more sensitive to some factors than others, meaning that depending on different settings some of the protocols may perform better.

When ignoring preemption overhead, MSOS-Priority, MPCP and SPIN mostly perform significantly better than other protocols. MSOS-Priority performs better than both MPCP and SPIN as the number of processors and (or) the length of critical sections (Figures 3(a) and 3(b)) is increased. However, increasing the utilization cap and (or) the number of critical sections per task punishes MSOS-Priority more than MPCP and SPIN. Figures 3(c) and 3(d) show the schedulability performance of the protocols against the length of critical sections and number of processors respectively, when the preemption overhead is ignored. As shown in Figures 3(d), OMLP is less sensitive to increasing the number of critical sections as it drops more smoothly compared to the rest of the protocols. For 6 critical sections per task, OMLP performs better than all protocols except MSOS-Priority and SPIN.

As one can expect, the performance of the suspend-based protocols
Percentage of Schedulability

Figure 3: Performance of synchronization protocols when the preemption overhead is ignored.

Figure 4: Performance of synchronization protocols as the per-preemption overhead increases. Number of processors=12, utilization cap=0.3, number of critical sections per task=3, length of critical sections=25 µs.

does not survive either as the preemption overhead reaches 20 µs. Hence, only under these two protocols tasks may experience remote preemption overhead which according to the results seems to be expensive.

The local preemption overhead regarding suspension is common for all suspend-based protocols. As shown in Figure 4, when the preemption overhead is very low, e.g., 20 µs per-preemption, the suspend-based protocols are affected less. MPCP does not survive as the per-preemption overhead reaches 60 µs and MSOS-FIFO does not survive either as the preemption overhead reaches 140 µs. For per-preemption overhead around 300 µs only MSOS-Priority survives and when the per-preemption overhead reaches 620 µs none of the suspend-based protocols survive.

So far we have seen that MSOS-Priority generally outperforms suspend-based protocols and in many cases it even performs better than spin-based protocols. However, it has not been clear how effective the priority assignment algorithm (Section 6) is and how much it helps the MSOS-Priority protocol to perform better. To investigate the effectiveness of the priority assignment algorithm we performed experiments in which we compared the performance of MSOS-Priority where the priorities of applications are assigned by the priority assignment algorithm to the performance of MSOS-Priority where the priorities were assigned randomly. The results showed that the priority assignment algorithm increases the schedulability of MSOS-Priority significantly. As shown in Figure 5, the priority assignment algorithm boosts the performance of MSOS-Priority significantly specially when the number of applications (processors) is increased. The reason is that a larger number of ap-
Figure 5: Performance of MSOS-Priority where priorities of the applications are assigned by the priority assignment algorithm against its performance where the priorities are assigned randomly. Utilization cap=0.3, number of critical sections per task=3, length of critical sections=85 µs.

applications gives the priority assignment algorithm more flexibility when it assigns priorities to the applications.

9. CONCLUSION
In this paper, we have presented a new alternative of our previously presented synchronization protocol MSOS for independently-developed real-time applications on multi-cores [3]. MSOS was originally developed for applications that are not prioritized on accessing shared resources. In this paper we extend MSOS to support prioritized applications. In the new MSOS, called MSOS-Priority, we have extended the notion of maximum Resource Wait Time (RWT) as well as Maximum Application Locking Time (MALT) which have to be functions of arbitrary time intervals. Moreover, we have proposed an optimal priority assignment algorithm to assign priorities to applications under MSOS-Priority.

We have performed an extensive experimental evaluation where the results showed that MSOS-Priority, when combined with the priority assignment algorithm, mostly performs significantly better than the existing suspend-based synchronization protocols under partitioned scheduling. In many cases it also outperforms spin-based protocols as well. Beside the good performance of MSOS-Priority, it offers the possibility of using it in open systems on a multi-core platform where an application is allocated on a dedicated core. An admission control program can perform better by using the interface-based global scheduling offered by MSOS-Priority since most of the complex calculations in the schedulability analysis of applications is performed off-line. Finally, MSOS generally offers real-time applications to be developed and analyzed in isolation and in parallel.

The schedulability analysis of MSOS-Priority can be improved by tightening of the calculations of the local blocking terms as well as MALT’s, e.g., by using actual critical section lengths rather than using a multiple of the longest critical sections. Future work includes tightening the blocking terms. All the existing locking protocols mentioned in this paper require shared memory platforms. Another future work is to develop synchronization protocols for real-time applications on multi-cores by means of message passing instead of shared memory synchronization.

10. REFERENCES


