Resource Sharing among Prioritized Real-Time Applications on Multiprocessors*

Farhang Nemati and Thomas Nolte
MRTC, Mälardalen University, Sweden
Email: {farhang.nemati, thomas.nolte}@mdh.se

***Abstract*—MSOS (Multiprocessors Synchronization protocol for real-time Open Systems) is a synchronization protocol for handling resource sharing among independently-developed real-time applications (components) on multi-core platforms. MSOS does not consider any priority setting among applications. To handle resource sharing based on the priority of applications, in this paper we propose a new protocol that allows for resource sharing among prioritized real-time applications on a multi-core 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 existing MSOS as well as to the current state of the art locking protocols under multiprocessor partitioned scheduling, i.e., MPCP, MSRP, FMLP and OMLP. The evaluations show that MSOS-Priority mostly performs significantly better than alternative approaches.**

I. 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.

Two main approaches for scheduling real-time systems on multi-cores 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.

* 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.

In our previous work [3] we proposed a synchronization protocol for handling resource sharing among independently-developed real-time applications on multi-core platforms called Multiprocessor Synchronization protocol for real-time Open Systems (MSOS). 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 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 when accessing mutually exclusive resources.

A. 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 protocol for partitioned scheduling, i.e., MPCP, MSRP, FMLP, and OMLP. To further explore the correlation of performance of the protocols to different parameters, e.g., number of processors, number of critical sections, length of critical sections, we have used a statistical method called Principal Component Analysis (PCA)[4] which is used to explore patterns in data with multiple dimensions (variables).

B. Related Work

In this section we present a non-exhaustive set of most related synchronization protocols for managing access to mutually exclusive 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 resource that is shared across processors suspends if the resource is locked by another task. In spin-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) [5] 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. [6] extended a spin-based alternative of MPCP.

MSRP: Gai et al. [7] presented MSRP (Multiprocessor Stack-based Resource allocation Protocol), which is a spin-based synchronization protocol. Under MSRP, tasks blocked on a global resource perform busy wait. A task inside a global critical section (gcs) executes non-preemptively.

FMLP: Block et al. [8] 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 [9] 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 [10], the authors differentiate between long FMLP and short FMLP where all global resources are only long and only short respectively. Thus, long FMLP and short FMLP are suspend-based and spin-based synchronization protocols respectively. In both alternatives the tasks accessing a global resource execute non-preemptively.

OMLP: Brandenburg and Anderson [11] proposed a new suspend-based locking protocol, called OMLP (O(m) Locking Protocol). OMLP is an *suspension-oblivious* protocol. Under a suspension-oblivious locking protocol, the suspended tasks are assumed to occupy processors and thus blocking is counted as demand. In difference with OMLP, other suspend-based protocols, are suspend-aware where suspended tasks are not assumed to occupy their processors. OMLP is *asymptotically optimal*, which means that 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. An asymptotically optimal locking protocol however does not mean it can perform better than non-asymptotically optimal protocols. Our experimental evaluations confirm this fact (Section VIII). Under 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) [12] for priority assignment in uniprocessors is the most related and similar to our priority assignment algorithm. Davis and Burns [13] have shown that OPA can be extended to fixed priority multiprocessor global scheduling if the schedulability of a task does not dependent on priority ordering among higher priority or among lower priority tasks. Our proposed algorithm is a generalization of OPA which can be applicable to assigning priorities to applications based on their requirements. Our algorithm can perform more efficiently than OPA since the schedulability test used by our algorithm is much simpler than that used in [13]. On the other hand, as we will show later in this paper (Section VI), although our algorithm has the same complexity as OPA, in some cases our algorithm will perform less schedulability tests than OPA.

II. 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 $A_k(\rho A_k, I_k)$ where ρA_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 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 τ_{A_k} which consists of n sporadic tasks, $\tau_i(T_i, C_i, \rho_i, \{Cs_{i,q,p}\})$ where T_i denotes the minimum inter-arrival time (period) between two successive jobs of task τ_i with worst-case execution time C_i and ρ_i as its priority. For the sake of simplicity we assume the tasks have implicit deadlines, i.e., the relative deadline of any job of τ_i is equal to T_i. However, with minor changes in the analysis the assumption of explicit deadlines can also be valid. A task, τ_h, has a higher priority than another task, τ_l, if $\rho_h > \rho_l$. 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 $R^L_{A_k}$ and $R^G_{A_k}$ respectively. The set of critical sections, in which task τ_i requests resources in R_{A_k} is denoted by $\{Cs_{i,q,p}\}$, where $Cs_{i,q,p}$ is the worst-case execution time of the p^{th} critical section of task τ_i in which the task locks resource R_q. We denote $Cs_{i,q}$ as the worst-case execution time of the longest critical section in which τ_i requests R_q. In the context of requesting resources, when it is said that a task τ_i is granted access to a resource R_q it means that R_q is available to τ_i, however it does not necessarily mean that τ_i has started using R_q unless we concretely state that τ_i is accessing R_q which means that τ_i has entered its critical section. Furthermore, when we state that access to R_q is granted to τ_i it implies that R_q is locked by τ_i. In this paper, we focus on non-nested critical sections. A job of task τ_i, is specified by J_i.

III. The 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.

A. Definitions

1) Resource Hold Time (RHT): The RHT of a global resource R_q by task τ_i in application A_k denoted by $\mathrm{RHT}_{q,k,i}$, is the maximum duration of time the global resource R_q can be locked by τ_i, i.e., $\mathrm{RHT}_{q,k,i}$ is the maximum time interval starting from the time instant τ_i locks R_q and ending at the time instant τ_i releases R_q. Thus, the resource hold time of a global resource, R_q, by application A_k denoted by $\mathrm{RHT}_{q,k}$, is as follows:

$$RHT_{q,k} = \max_{\tau_i \in \tau_{q,k}} \{RHT_{q,k,i}\} \tag{1}$$

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

2) Maximum Resource Wait Time: For a global resource R_q in application A_k, denoted by $\mathrm{RWT}_{q,k}$, is the worst-case time that any task τ_i within A_k may wait for other applications on R_q whenever τ_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 τ_i. As we will see (Section IV), this assumption is not valid for the case that the applications are prioritized.

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., $2RWT_{1,k} + 3RWT_{3,k} \leq 18$. The requirements of each application are extracted from its schedulability analysis in isolation. Z_k in the interface represents a set; $Z_k = \{\ldots, Z_{q,k}, \ldots\}$, where $Z_{q,k}$, called Maximum Application Locking Time (MALT), represents the maximum duration of time that any task τ_x in any other application A_l $(l \neq k)$ may be delayed by tasks in A_k whenever τ_x requests R_q.

B. 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 local 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 τ_i that is granted access to a global resource R_q, should not be delayed by any other task τ_j, unless τ_j holds another global resource. To achieve this, the priority of any task τ_i within an application A_k requesting a global resource R_q is increased immediately to $\rho_i + \rho^{max}(A_k)$, where $\rho^{max}(A_k) = \max\{\rho_i | \tau_i \in \tau_{A_k}\}$. Boosting the priority of τ_i when it is granted access to a global resource will guarantee that τ_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 τ_i is computed as follows:

$$RHT_{q,k,i} = Cs_{i,q} + H_{i,q,k} \tag{2}$$

where $H_{i,q,k} = \sum_{\substack{\forall \tau_j \in \tau_{A_k},\, \rho_i < \rho_j \\ \wedge\ R_l \in R^G_{A_k},\, l \neq q}} Cs_{j,l}$.

An application A_l can delay another application A_k on a global resource R_q up to $Z_{q,l}$ 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:

$$RWT_{q,k} = \sum_{A_l \neq A_k} Z_{q,l} \tag{3}$$

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:

$$Z_{q,k} = \sum_{\tau_i \in \tau_{q,k}} RHT_{q,k,i} \tag{4}$$

for Priority-based local queues:

$$Z_{q,k} = |\tau_{q,k}| \max_{\tau_i \in \tau_{q,k}} \{RHT_{q,k,i}\} \tag{5}$$

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

IV. The 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 accessing the resources 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 [14] that cache-related preemption overhead, depending on the working set size (WSS) of jobs (WSS of 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 τ_l executing within a gcs can be preempted by another higher priority task τ_h within a gcs if they are accessing different resources. This increases the number of preemptions which adds up the preemption overhead to gcses 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 τ_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 τ_i.

A. Request Rules

Rule 1: Whenever a task τ_i in an application A_k is granted access to a global resource R_q the priority of τ_i is boosted to $\rho_i + \rho^{max}(A_k)$. 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 τ_i accesses R_q, i.e., starts using R_q, its priority is further boosted to 2 $\rho^{max}(A_k)$ 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 τ_i requests it, τ_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 queueτ_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 τ_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.

V. 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.

A. Computing Resource Hold Times

Similar to Lemma 1 in [3], it can be shown that whenever a task τ_i is granted access to a global resource R_q, it can be delayed by at most one gcs per each higher priority task τ_j where τ_j uses a global resource other than R_q. However, once τ_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 τ_l starts using a global resource R_s before τ_i is granted access to R_q, τ_l will delay τ_i as long as τ_l is using R_s because τ_l cannot be preempted (Rule 1). It is easy to see that τ_i will not anymore be delayed by lower priority tasks that are granted access to global resources other than R_q; whenever τ_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:

$$RHT_{q,k,i} = Cs_{i,q} + H_{i,q,k} + \max_{\substack{\forall \tau_l \in \tau_{A_k},\, \rho_i > \rho_l \\ \wedge\, R_s \in R^G_{A_k},\, s \neq q}} \{Cs_{l,s}\} \quad (6)$$

B. Blocking times under MSOS-Priority

Under MSOS-Priority, by blocking time we mean delays that any task τ_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 τ_i are the tasks that are belong to the same application as τ_i.

Similar to MSOS-FIFO, there are three possible blocking terms that a task τ_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:

1) Local blocking due to local resources, denoted by $B_{i,1}$: Is the upper bound for the total blocking time that τ_i incurs from lower priority tasks using local resources and is calculated as follows:

$$B_{i,1} = \min\{n_i^G + 1, \sum_{\rho_j < \rho_i} \lceil T_i/T_j \rceil n_j^L(\tau_i)\} \max_{\substack{\rho_j < \rho_i \\ \wedge\, R_l \in R_{A_k}^L \\ \wedge\, \rho_i \leq ceil(R_l)}} \{Cs_{j,l}\} \quad (7)$$

where $ceil(R_l) = \max\{\rho_i |\ \tau_i \in \tau_{l,k}\}$, n_i^G is the number of gcses of τ_i, and $n_j^L(\tau_i)$ is the number of the critical sections in which τ_j requests local resources with ceiling higher than the priority of τ_i.

2) Local blocking due to global resources, denoted by $B_{i,2}$: Is the upper bound for the maximum blocking time that τ_i incurs from lower priority tasks using global resources and can be calculated as follows:

$$B_{i,2} = \sum_{\substack{\rho_j < \rho_i \\ \wedge\, \{\tau_i, \tau_j\} \subseteq \tau_{A_k}}} \min\{n_i^G + 1, \lceil T_i/T_j \rceil n_j^G\} \max_{R_q \in R_{A_k}^G} \{Cs_{j,q}\} \quad (8)$$

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

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 τ_x in an application A_l requests a global resource R_q which is already locked by a task within application A_k. In this case A_l 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 lower priority than A_l, after A_k releases R_q it cannot lock R_q anymore as long as A_l is in the global queue, i.e., as long as there are more tasks in A_l requesting R_q. On the other hand if A_k has a higher priority than A_l, before A_l 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 τ_x incurs from A_k during τ_x'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 τ_x in any other application A_l depends on: (i) if A_k has a lower or higher priority than A_l, (ii) the period of τ_x.

Lemma 1. *Under MSOS-Priority, whenever any task τ_i in an application A_k requests a global resource R_q, only one lower priority application can block τ_i; this delay is at most $\max_{\rho A_k > \rho A_l}\{RHT_{q,l}\}$ time units.*

Proof: At the time τ_i in A_k requests R_q, if a lower priority application A_l has already locked R_q, it will delay A_k for at most $RHT_{q,l}$ time units. Since access to global resources is granted to applications based on their priorities, after R_q is released by A_l no more lower priority applications will have a chance to access R_q before A_k. ■

Whenever any task τ_i in 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 τ_i. The maximum delay that τ_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 τ_x in any lower priority application A_l may incur from tasks in A_k during time interval t, each time τ_x 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 τ_j during time interval t equals $\lceil \frac{t}{T_j} \rceil + 1$. On the other hand, whenever a job J_j of τ_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 τ_j. Thus, the total workload of all critical sections of τ_j locking R_q during time interval t is denoted by $W_j(t, R_q)$ and is computed as follows:

$$W_j(t, R_q) = (\lceil \tfrac{t}{T_j} \rceil + 1)\ n_{j,q}^G\ RHT_{q,k,j} \quad (9)$$

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. *Maximum Resource Wait Time (RWT) for a global resource R_q incurred by task τ_i in application A_k, denoted by $\mathrm{RWT}_{q,k,i}(t)$, is the maximum duration of time that τ_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 (Lemma 1) and higher priority applications (Equation 10), can be calculated as follows:

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

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. The RWT for a global resource R_q of a task τ_i in application A_k during the period of τ_i equals to $RWT_{q,k,i}(T_i)$ which covers all delay introduced from both higher priority and lower priority applications sharing R_q:

$$RWT_{q,k,i} = \sum_{\rho A_k < \rho A_l} Z_{q,l}(T_i) + \; n^G_{i,q} \max_{\rho A_k > \rho A_l} \{RHT_{q,l}\} \quad (12)$$

where $RWT_{q,k,i}(T_i)$ is denoted by $RWT_{q,k,i}$.

Computing Remote Blocking: Equation 12 can be used to compute remote blocking $B_{i,3}$ for task τ_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 12, 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:

$$B_{i,3} = \sum_{\substack{\forall R_q \in R^G_{A_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} RWT_{q,k,i} \quad (13)$$

C. 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 12, the calculation of the RWT of a task τ_i, in application A_k, for a global resource R_q, requires MALT's, e.g., $Z_{q,h}(t)$, from higher priority applications as well as RHT's, e.g., $RHT_{q,l}$, 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 MALT 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.

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 τ_i, the maximum value of blocking time B_i^{max} that τ_i can tolerate without missing its deadline can be calculated as follows:

τ_i is schedulable using the fixed priority scheduling policy if the following statement holds:

$$0 < \exists t \leq T_i \quad \texttt{rbf}_{\texttt{FP}}(i,t) \leq t, \quad (14)$$

where $\texttt{rbf}_{\texttt{FP}}(i,t)$ denotes *request bound function* of τ_i which computes the maximum cumulative execution requests that could be generated from the time that τ_i is released up to time t, and is computed as follows:

$$\texttt{rbf}_{\texttt{FP}}(i,t) = C_i + B_i + \sum_{\rho_i < \rho_j} (\lceil t/T_j \rceil C_j) \quad (15)$$

By substituting B_i by B_i^{max} in Equations 14 and 15, B_i^{max} can be calculated as follows:

$$B_i^{max} = \max_{0<t\leq T_i} (t - (C_i + \sum_{\rho_i < \rho_j} (\lceil t/T_j \rceil C_j))) \quad (16)$$

The total blocking of task τ_i is the summation of three blocking terms calculated in Section V-B:

$$B_i = B_{i,1} + B_{i,2} + B_{i,3} \quad (17)$$

Since $B_{i,1}$ and $B_{i,2}$ totally depend on internal factors, i.e., the parameters from the application that τ_i belongs to, they are considered as constant values, i.e., they depend on only internal factors of τ_i's application. Thus, Equation 17 can be rewritten as follows:

$$B_i = \gamma_i + \sum_{\substack{\forall R_q \in R^G_{A_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} RWT_{q,k,i} \quad (18)$$

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

Equation 18 shows that the total blocking time of task τ_i is a function of maximum resource wait times of τ_i for the global resources accessed by τ_i. With the achieved B_i^{max} and Equation 18 a requirement can be extracted:

$$\gamma_i + \sum_{\substack{\forall R_q \in R^G_{A_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} RWT_{q,k,i} \leq B_i^{max} \quad (19)$$

or:

$$\sum_{\substack{\forall R_q \in R^G_{A_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} RWT_{q,k,i} \leq B_i^{max} - \gamma_i \quad (20)$$

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., $RWT_{q,k,i}$ of global resources accessed by tasks within A_k which are calculated using Equation 12.

One can see that most of the calculations in the scheduling 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 V-C1, 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 the complexity of interface-based scheduling is O(m n) where m is the number of applications and n is the number of tasks per application.

VI. The 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 assigns unique priorities to the applications we propose an optimal algorithm similar to the algorithm presented by Davis and Burns [13]. The algorithm is based on interface-based scheduling 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

```
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 SchedulableApps ← { };
7        List NotSchedulableApps ← { };
8        for each application A in remainedAppList
9            if all requirements of A are satisfied
10                 add A to SchedulableApps;
11           else
12                 add A to NotSchedulableApps;
13           end if
14       end for
15       if SchedulableApps is empty
16             return fail;
17       remainedAppList ← NotSchedulableApps;
18       for each application A in remainedAppList
19           A.priority ← A.priority + (number of applications in SchedulableApps);
20       end for
21       incr ← 0;
22       for each application A in SchedulableApps
23           A.priority ← A.priority + incr;
24           incr ← incr + 1;
25       end for
26 end while
27 return succeed;
```

Fig. 1. The Priority Assignment Algorithm

code of the algorithm is shown in Figure 1. 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 12 the algorithm changes condition $\rho A_k < \rho A_l$ in the first term to $\rho A_k \leq \rho A_l$.

Figure 2 illustrates an example of the algorithm.

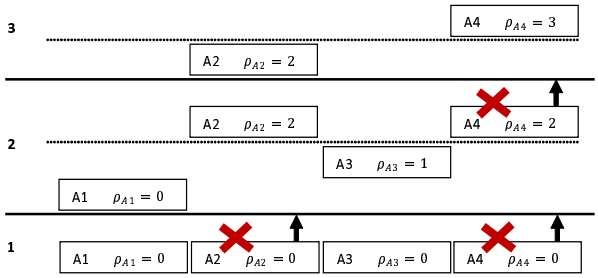


Fig. 2. Illustrative Example for the Priority Assignment Algorithm

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., $\rho A_i = 0$ 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 [12] 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 $m, m-1, \ldots, 2, 1$ has been performed $m, m-1, \ldots, 2, 1$ 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.

However, it may happen that at a stage, x number of applications are schedulable where $x > 1$. In this case the priority of all remaining applications (i.e. applications that are unschedulable at the current stage) will be increased by x (Figure 1, Line 19 of the algorithm). This means that, the maximum number of schedulability tests for each of the remaining applications would be decreased by x, i.e., the number of stages the algorithm runs is decreased by x. The more similar stages exist the lower the maximum number of schedulability tests will be. As a result the maximum number of stages and consequently the number schedulability tests are decreased. This is not the case in Audsley's OPA; depending on the order of selecting tasks (or applications), it is still possible that $m(m+1)/2$ schedulability tests would be performed, e.g., OPA finds a solution in exactly m stages. E.g., in the illustrative example in Figure 2, OPA will assign priorities in 4 stages, and if it selects the applications in order A_4, A_2, A_3, A_1, it will perform 4, 3, 1, 1 schedulability tests for A_4, A_2, A_3 and A_1 respectively, and in total 9 tests will be performed. On the other hand, our algorithm assigns priorities in 3 stages and it performs 3, 2, 1, 1 schedulability tests for A_4, A_2, A_3 and A_1 respectively, and in total 7 tests are performed.

Lemma 2. *The priority assignment algorithm is optimal, i.e., if the algorithm fails to assign unique priorities any hypothetically optimal algorithm will also fail.*

Proof: We assume that the priority assignment algorithm at some stage fails, lets assume that it fails at stage f ($1 \leq f \leq m$ where m is the number of applications), i.e., at stage f the algorithm does not find any schedulable application and thus fails. This means that there is no application in the system that can be schedulable with priority $f-1$. We assume that a hypothetically optimal algorithm succeeds to assign unique priorities to applications. This means that any application A_k is assigned a unique priority where $0 \leq \rho A_k \leq m-1$. Thus there is a schedulable application that has priority equal to $f-1$. This is in contradiction with the assumption with which the priority assignment algorithm fails. ■

VII. 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 [15]. 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., SRP, 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.

A. Local Preemption Overhead

Under a suspend-based protocol, e.g., MSOS-Priority, MPCP, OMLP, whenever a task τ_i requests a global resource if the resource is locked by a task in a remote processor (application), τ_i suspends. While τ_i is suspending, lower priority tasks can execute and request global resources as well. Later on when τ_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 τ_l can preempt τ_i up to $\lceil T_i/T_l \rceil n_l^G$ times. On the other hand τ_l cannot preempt τ_i more than $n_i^G + 1$ times. Thus, τ_i can be preempted by any lower priority task τ_l at most $\min\{n_i^G+1, \lceil T_i/T_l \rceil n_l^G\}$ times.

Task τ_i may also experience extra preemptions from higher priority tasks requesting global resources. Whenever a higher priority task τ_h requests a global resource which is locked by remote tasks, it suspends and thus τ_i has the chance to execute. When τ_h is granted access to the resource it will preempt τ_i. This may happen up to $\lceil T_i/T_h \rceil n_h^G$ times.

Thus, the total number of extra preemptions that a task τ_i may experience from local tasks, because of suspension on global resources, is denoted by $Lpreem_i$ and is calculated as follows:

$$Lpreem_i = \sum_{\rho_l < \rho_i} \min\{n_i^G + 1, \lceil T_i/T_l \rceil n_l^G\} + \sum_{\rho_h > \rho_i} \lceil T_i/T_h \rceil n_h^G \quad (21)$$

The preemption overhead in Equation 21 is due to suspension of tasks while they are waiting for global resources. Spin-based protocols do not suffer from this preemption overhead at all as they do not let a task suspend while waiting for a global resource.

B. Remote Preemption Overhead

Besides the preemption overhead a task τ_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 τ_r is allowed to be preempted while it is using a global resource R_q, i.e., τ_r 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 τ_r 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 gcses and thus remote preemption overhead has to be included in their schedulability tests. Under MPCP, a task within a gcs can be preempted by gcses from both lower priority and higher priority tasks [5]. Under MSOS-FIFO a task within a gcs can only be preempted by higher priority tasks within their gcses. Under both MPCP and MSOS-FIFO a gcs of a task τ_i in which it accesses a global resource R_q can be preempted by at most one gcs per each task τ_j in which it accesses a global resource other than R_q. This is because the preempting task τ_j will not have chance to execute and enter another gcs before τ_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 τ_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 τ_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 τ_i in which it requests a global resource R_q can only be preempted by gcses 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 τ_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 gcses has to be inflated by the preemption overhead they may incur. This means gcses become longer and under MSOS-FIFO it leads to longer RHT's.

VIII. 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 regarding 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.

A. 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 [14] 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 μs) 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.

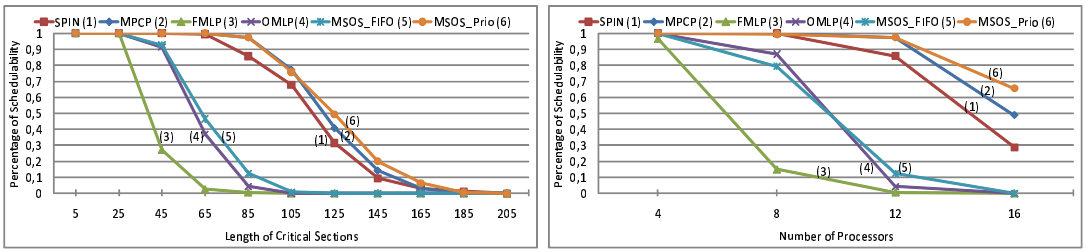
We generated 1 million task sets. In the generated task sets the number of task sets were between 115 and 215 for each setting, where the number of settings was 6336. We repeated the experiments three times and we did not observe any significant difference in the obtained results. This means that 1 million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

B. 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 protocols may perform better.

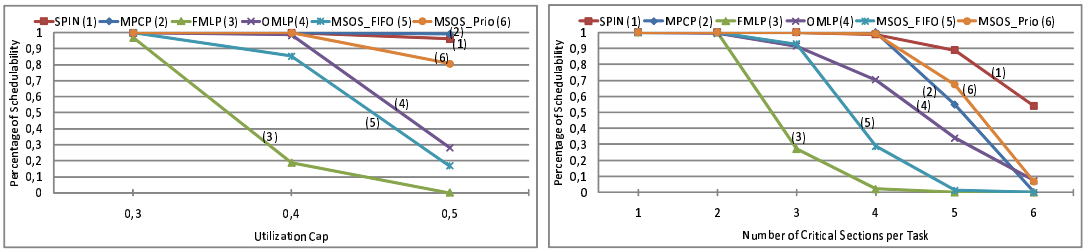
When ignoring preemption overhead, MSOS-Priority, MPCP and SPIN mostly perform significantly better than other protocols. MSOS-Priority performs better than both MPCP and SPIN as the number of processors and (or) the length of critical sections (Figures 3(a) and 3(b))[1] is increased. However, increasing 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

[1]Please notice that in all the figures showing the results of the experiments the lines connecting points are used to illustrate the trends and they do not represent any regression.



(a) Performance of synchronization protocols against the length of critical sections. Number of processors=12, utilization cap=0.3, number of critical sections per task=3.

(b) Performance of synchronization protocols against the number of processors. Utilization cap=0.3, number of critical sections per task=3, length of critical sections=85 μs.



(c) Performance of synchronization protocols against the utilization cap. Number of processors=8, number of critical sections per task=3, length of critical sections=45 μs.

(d) Performance of synchronization protocols against the number of critical sections per task. Number of processors=12, utilization cap=0.3, length of critical sections=45 μs.

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

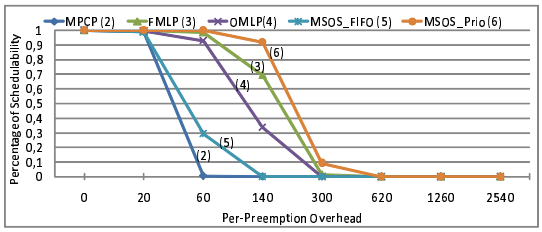


Fig. 4. Performance of synchronization protocols as the preemption overhead increases. Number of processors=12, utilization cap=0.3, number of critical sections per task=3, length of critical sections=25 μs.

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 decreases as the preemption overhead is increased (Figure 4). Despite of the value of per-preemption overhead, MSOS-Priority almost always outperforms all other suspend-based protocols. Only in some cases where the preemption overhead is ignored, MSOS-Priority performs similar to MPCP. For lower per-preemption overhead, e.g., less than $140\mu s$, in most cases where the number of processors and (or) the length of critical sections is relatively large MSOS-Priority outperforms spin-based protocols as well (Figure 5). However, in this paper we have not considered system dependant overhead, e.g., overhead of queue management. We believe that, similar to the preemption overhead, the system overhead will favor spin-based protocols significantly, and for relatively large amount of system overhead the suspend-based protocols may hardly (if not at all) outperform spin-based protocols, specially when the lengths of critical sections are relatively short.

Among suspend-based protocols MPCP drops sharply against preemption overhead already from very low per-preemption overhead followed by MSOS-FIFO. The reason that MPCP and MSOS-FIFO are more sensitive to preemption overhead is that they are the only protocols that allow preemption of a task while it is using a global resource, i.e., the task is within a gcs. 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 $60\mu s$ and MSOS-FIFO does not survive either as the preemption overhead reaches $140\mu s$. For per-preemption overhead around $300\mu s$ only MSOS-Priority survives and when the per-preemption overhead reaches $620\mu s$ 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 VI) is and how much it helps 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

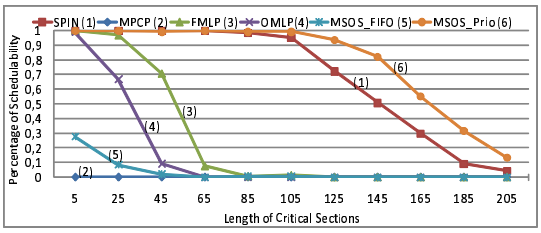


Fig. 5. MSOS-Priority outperforms spin-based protocols in many cases for lower per-preemption overhead, e.g., as the length of critical sections is increased. Number of processors=16, utilization cap=0.3, number of critical sections per task=2, per-preemption overhead=140μs.

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 6, the priority assignment algorithm boosts the performance of MSOS-Priority significantly specially when the number of applications (processors) is increased. The reason is that larger number of applications gives the priority assignment algorithm more flexibility when it assigns priorities to the applications.

To further illustrate an overview of relationship between the performance of protocols and different parameters, we have used a bilinear modeling method called Principal Component Analysis (PCA) [4]. PCA can be used to visualize and interpret relationships and insights when investigating an output against multiple variables. We have used PCA to observe which parameters and how strong they contribute to the differences among the synchronization protocols. Figure 7 illustrates the effect of different parameters on the synchronization protocols using PCA. P, UC, CN, CL, and O denote the number of processors, utilization cap per processor, the number of critical sections per task, length of each critical section and per-preemption overhead respectively. The closer the angle between a parameter and a protocol to 0 or 180 the more correlated the protocol is to the parameter positively or negatively respectively. Besides, the longer the vector of a parameter is the stronger the correlation is.

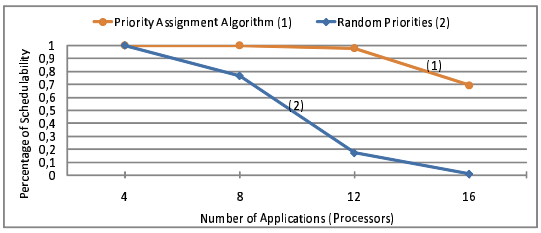


Fig. 6. Performance of MSOS-Priority where priorities of the applications are assigned by the priority assignment algorithm against its performance where the priorities are assigned randomly. Utilization cap=0.3, number of critical sections per task=3, length of critical sections=85 μs.

An interesting interpretation illustrated in Figure 7, is that the suspend-based protocols are most negatively correlated to the preemption overhead, i.e., among other parameters the preemption overhead affects negatively the suspend-based protocols the most. Among suspend-based protocols, MPCP is affected the most followed by MSOS-FIFO. On the other hand the spin-based protocols are mostly affected by the length of the critical sections and the number of processors followed by the number of critical sections and the utilization cap. Briefly speaking, the preemption overhead favors spin-based protocols while the length of critical sections and the number of processors favor the suspend-based protocols.

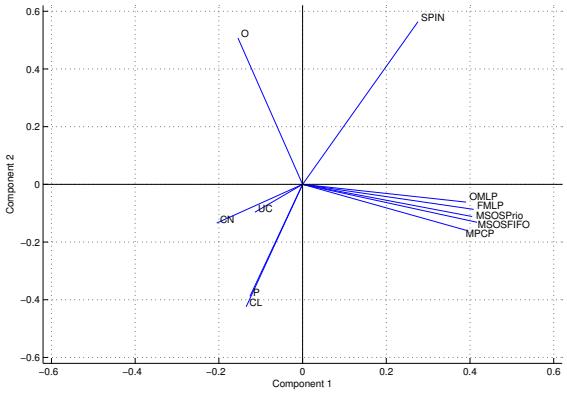


Fig. 7. Investigate the sensitivity of the synchronization protocols against all factors using PCA.

IX. 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 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 scheduling 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 can further , e.g., by using actual

critical section lengths rather than using multiple of the longest critical sections. As a future can be the work on tightening the blocking terms. All the existing locking protocols mentioned in this paper require shared memory platforms. An interesting future work is to develop synchronization protocols for real-time applications on multi-cores by means of message passing instead of shared memory synchronization.

REFERENCES

[1] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

[2] U. Devi. Soft real-time scheduling on multiprocessors. In *PhD thesis, available at www.cs.unc.edu/˜anderson/diss/devidiss.pdf*, 2006.

[3] F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *Proceedings of the 23th IEEE Euromicro Conference on Real-time Systems (ECRTS'11)*, pages 251–261, 2011.

[4] K.H. Esbensen. *Multivariate Data Analysis - in practice (5th Edition)*. CAMO ASA, Oslo, 2010.

[5] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[6] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Proceedings of 30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 469–478, 2009.

[7] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the janus multiple processor on a chip platform. In *Proceedings of 9th IEEE Real-Time And Embedded Technology Application Symposium (RTAS'03)*, pages 189–198, 2003.

[8] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07)*, pages 47–56, 2007.

[9] B. Brandenburg and J. Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS. In *Proceedings of 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'08)*, pages 185–194, 2008.

[10] B. B. Brandenburg and J. H. Anderson. A comparison of the M-PCP , D-PCP , and FMLP on LITMUS. In *Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS'08)*, pages 105–124, 2008.

[11] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of 31th IEEE Real-Time Systems Symposiumst (RTSS'10)*, pages 49–60, 2010.

[12] N.C. Audsley. Optimal Priority Assignment And Feasibility Of Static Priority Tasks With Arbitrary Start. Technical report, 1991.

[13] R. I. Davis and A. Burns. Priority Assignment for Global Fixed Priority Pre-Emptive Scheduling in Multiprocessor Real-Time Systems. In *Proceedings of the 30th IEEE International Real-Time Systems Symposium (RTSS'09)*, pages 398–409, 2009.

[14] A. Bastoni, B.B. Brandenburg, and J.H. Anderson. Is semi-partitioned scheduling practical? In *Proceedings of 23rd IEEE Euromicro Conference on Real-time Systems (ECRTS'11)*, pages 125–135, 2011.

[15] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.