2011 23rd Euromicro Conference on Real-Time Systems

Independently-developed Real-Time Systems on Multi-cores with Shared Resources*

Farhang Nemati, Moris Behnam and Thomas Nolte
Mälardalen Real-Time Research Centre
Mälardalen University, Västerås, Sweden
Email: {farhang.nemati, moris.behnam, thomas.nolte}@mdh.se

***Abstract*—In this paper we propose a synchronization protocol for resource sharing among independently-developed real-time systems on multi-core platforms. The systems may use different scheduling policies and they may have their own local priority settings. Each system is allocated on a dedicated processor (core).**

In the proposed synchronization protocol, each system is abstracted by an interface which abstracts the information needed for supporting global resources. The protocol facilitates the composability of various real-time systems with different scheduling and priority settings on a multi-core platform.

We have performed experimental evaluations and compared the performance of our proposed protocol (MSOS) against the two existing synchronization protocols MPCP and FMLP. The results show that the new synchronization protocol enables composability without any significant loss of performance. In fact, in most cases the new protocol performs better than at least one of the other two synchronization protocols. Hence, we believe that the proposed protocol is a viable solution for synchronization among independently-developed real-time systems executing on a multi-core platform.

I. Introduction

The availability of multi-core platforms has attracted a lot of attention in multiprocessor embedded software analysis and runtime policies, protocols and techniques. As the multi-core platforms are to be the defacto processors, the industry must cope with a potential migration towards multi-core platforms.

An important issue for industry when it comes to migration to multi-cores is the *existing* systems. When migrating to multi-cores it should be possible that several of these systems co-execute on a shared multi-core platform. The (often independently-developed) systems may have been developed with different techniques, e.g., several real-time systems that will co-execute on a multi-core may have different scheduling policies. However, when the systems co-execute on the same multi-core platform they may share resources that require mutual exclusive access. Two challenges to overcome when migrating existing systems to multi-cores are how to migrate the independently-developed systems with minor changes, and how to abstract systems sufficiently, such that the developer of one system does not need to be aware of particular techniques used in other systems.

On the other hand, looking at industrial systems, to speed up their development, it is not uncommon that large and complex systems are divided into several semi-independent subsystems each of which is developed independently. The subsystems which may share resources will eventually be integrated and coexist on the same platform. This issue has got attention and has been studied in the uniprocessor domain [1], [2], [3]. However, new techniques are sought for scheduling semi-independent subsystems on multi-cores.

Looking at current state-of-the-art, two main approaches for scheduling real-time systems on multiprocessors (multi-cores) exist; global and partitioned scheduling [4], [5], [6]. 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). Partitioned scheduling policies have been used more often in industry and are they supported widely by commercial real-time operating systems [7], inherent in their simplicity, efficiency and predictability.

The work presented in this paper is inspired by our initial ideas presented in [8], where we focus on the partitioned scheduling policy and synchronization protocols. Allocation (partitioning) of independently-developed systems on a multi-core architecture may have the following alternatives: (i) one processor includes only one system, (ii) one processor may contain several systems, (iii) a system may be distributed over more than one processor.

In this paper, we focus on the first alternative in which each a system is allocated on a dedicated processor (core). For the second alternative, the well studied techniques for integrating independently-developed systems on uniprocessors can be used, e.g., the methods presented in [9] and [1]. These techniques usually abstract the timing requirements of the internal tasks of each system and using this each system is abstracted as one (artificial) task, hence from outside of the containing processor there will be one system (task set) on the processor. Thus by reusing uniprocessor techniques in this area the second alternative becomes similar to the first alternative. However, extension to the third alternative remains as a future work.

* 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.

1068-3070/11 $26.00 © 2011 IEEE
DOI 10.1109/ECRTS.2011.31

A. Contributions

The contributions of this paper are as follows:

- We propose a *synchronization protocol* for resource sharing among independently-developed real-time systems (open real-time systems) on a multi-core platform, each of which is allocated on a dedicated core. We have named the protocol as Multiprocessors Synchronization protocol for real-time Open Systems (MSOS).
- We derive an *interface-based schedulability condition* for MSOS. The interface abstracts the global resource sharing of a system in one processor through a set of requirements that should be satisfied to guarantee the schedulability of the system in the processor. A global resource is a resource that is shared across processors. A requirement is a function of resource maximum wait times of global resources (i.e., the worst-case time that a processor may wait for a global resource to be available) which should not exceed a certain value. Thus, the requirements in the interface only depend on the maximum wait times of global resources. Hence we do not need any information from other processors, e.g., scheduling protocol or priority setting policy on other processors, in calculating the interface of a processor.
- We have *evaluated the performance* of MSOS by means of experimental evaluation. In the experiments we compared MSOS against MPCP and FMLP. The obtained results show that the composability offered by MSOS does not introduce any significant loss of performance and in most cases it even performs better than at least one of the two other protocols. Thus we believe MSOS can be an appropriate synchronization protocol for handling resource sharing among independently-developed systems on a multi-core platform.

B. Related Work

In the context of independently-developed real-time systems in a shared open environment on uniprocessors, a considerable amount of work has been done. A non-exhaustive list of works in this domain includes [10], [11], [12], [13], [14], [9], [15]. Hierarchical scheduling has been studied and developed as a solution for these systems.

Hierarchical scheduling techniques have also been developed for multiprocessors (multi-cores) [16], [17]. However, the systems (called clusters in the mentioned papers) are assumed to be independent and do not allow for sharing of mutually exclusive resources.

In the context of the synchronization protocols, PCP (Priority Ceiling Protocol) [18] and SRP (Stack-based Resource allocation Protocol) [19] are two of the best known methods for synchronization in uniprocessor systems.

For multiprocessor synchronization, Rajkumar et al. proposed a synchronization protocol [20] which later was called Distributed Priority Ceiling Protocol (DPCP) [21]. DPCP extends PCP to distributed systems and it can be used with shared memory multiprocessors. Rajkumar presented MPCP [21], which extends PCP to multiprocessors hence allowing for synchronization of tasks sharing mutually exclusive resources using partitioned FPS. Lakshmanan et al. [7] investigated and analyzed two alternatives of execution control policies (suspend-based and spin-based remote blocking) under MPCP. However, MPCP can be used for one single system whose tasks are distributed on different processors. Furthermore for schedulability analysis of each processor, detailed information of tasks allocated on other processors (e.g., priority, the number of global critical section, etc) may be required. Under MSOS the schedulability test of a system on a processor is represented as requirements in its interface which can be obtained without any information from other systems (even before these systems are developed) which will be allocated on other processor.

Gai et al. [22], [23] presented MSRP (Multiprocessor SRP), which is a P-EDF (Partitioned EDF) based synchronization protocol for multiprocessors and is an extension of SRP to multiprocessors. Lopez et al. [24] presented an implementation of SRP under P-EDF. Devi et al. [25] presented a synchronization technique under G-EDF. The work is restricted to synchronization of non-nested accesses to short and simple objects, e.g., stacks, linked lists, and queues. In addition, the main focus of the method is on soft real-time systems.

Block et al. [26] presented Flexible Multiprocessor Locking Protocol (FMLP) which is the first synchronization protocol for multiprocessors that can be applied to both partitioned and global scheduling algorithms, i.e., P-EDF and G-EDF. An implementation of FMLP has been described in [27]. Brandenburg and Anderson in [28] have extended partitioned FMLP to the fixed priority scheduling policy and derived a schedulability test for it. In a later work [29], the same authors compared DPCP, MPCP and FMLP.

Easwaran and Andersson proposed a synchronization protocol [30] under the global fixed priority scheduling protocol. In this paper, the authors have derived schedulability analysis of the Priority Inheritance Protocol (PIP) under global scheduling algorithms.

Recently, Brandenburg and Anderson [31] presented a new suspension-based locking protocol, called O(m) Locking Protocol (OMLP), which has variations for both global and partitioned scheduling. The OMLP (both variations) 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. On the other hand, OMLP is an *suspension-oblivious* protocol. Under a suspension-oblivious locking protocol, the suspended jobs are assumed to occupy processors and thus blocking is counted as demand. To test the schedulability, the worst-case execution times of tasks are inflated with blocking times. This means that blocking time of any task is introduced to all lower priority tasks. In this paper we focus on *suspension-aware* locking synchronization in which suspended jobs are not assumed to occupy processors. In addition Brandenburg and Anderson have also proposed an asymptotically optimal

suspension-aware protocol in [31], called Simple Partitioned FIFO Locking Protocol (SPFP) (under partitioned scheduling), however it uses one single global FIFO queue in which all requests to all global resources are enqueued. However, the drawback of using one single FIFO queue is that it prevents parallelization in accessing resources.

In all the aforementioned existing synchronization protocols (under partitioned scheduling) on multi-cores it is assumed that the tasks of a system are distributed among processors and all processors use the same scheduling policy, e.g., EDF or RM. Furthermore, in the schedulability analysis of the existing protocols (e.g., MPCP and FMLP) a processor needs timing attributes of tasks allocated on other processors that share resources with its tasks. MSOS, however, allows each system in a processor to use its own scheduling policy and it abstracts the timing requirements regarding global resources shared by the system in its interface, hence, it is not required to reveal its task attributes to other processors which it shares resources with. Recently, in industry, co-existing of several separated systems on a multi-core platform (called virtualization) has been considered to reduce the hardware costs [32]. MSOS seems to be a natural fit for synchronization under virtualization of real-time systems on multi-cores.

II. Task and Platform Model

In this paper, we assume that the multiprocessor (multi-core) platform is composed of identical, unit-capacity processors (cores) with shared memory. Each processor contains a different task set (system). The scheduling techniques used on each processor may differ from other processors, e.g., one processor can be scheduled by fixed priority scheduling (e.g., RM) while another processor is scheduled by dynamic priority scheduling (e.g., EDF), which means the priority of tasks are local to each processor. However, for the sake of presentation clarity, in this paper we focus on schedulability analysis of processors with fixed priority scheduling. A task set allocated on a processor, P_k, is denoted by τ_{P_k} and 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 between two successive jobs of task τ_i with worst-case execution time C_i and ρ_i as its priority. The tasks have implicit deadlines, i.e., the relative deadline of any job of τ_i is equal to T_i. A task, τ_h, has a higher priority than another task, τ_l, if $\rho_h > \rho_l$. For the sake of simplifying presentation we assume that each task has a unique priority. The tasks on processor P_k share a set of resources, R_{P_k}, which are protected using semaphores. The set of shared resources (R_{P_k}) consists of two subsets of different types of resources; *local* and *global* resources. A local resource is only shared by tasks on the same processor while a global resource is shared by tasks on more than one processor. The sets of local and global resources accessed by tasks on processor P_k are denoted by $R^L_{P_k}$ and $R^G_{P_k}$ respectively. The set of critical sections, in which task τ_i requests resources in R_{P_k} is denoted by $\{Cs_{i,p,q}\}$, 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 $R_q \in R_{P_k}$. We denote $C_{i,q}$ as the worst-case execution time of the longest critical section in which τ_i requests R_q. In this paper, we focus on non-nested critical sections (the common case). A job of task τ_i, is specified by J_i.

III. The Multiprocessors Synchronization Protocol for Real-time Open Systems (MSOS)

A. Assumptions and terminology

We assume that systems are already allocated on processors and that each processor may use a different scheduling policy. The tasks within a system allocated on a processor do not need any information about the tasks within other systems allocated on other processors, neither do they need to be aware of the scheduling policies on other processors, when performing schedulability analysis of the system.

Definition 1: *Resource Hold Time* of a global resource R_q by task τ_i on processor P_k is denoted by $\mathrm{RHT}_{q,k,i}$ and is the maximum duration of time the global resource R_q can be locked by τ_i. In other words, $\mathrm{RHT}_{q,k,i}$ is the 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, which includes the longest critical section in which τ_i accesses R_q as well as the possible interference form other tasks accessing global resources other than R_q. Consequently, the resource hold time of a global resource, R_q, by processor P_k (i.e., the maximum duration of time R_q is locked by any task on P_k) denoted by $\mathrm{RHT}_{q,k}$, is as follows:

$$\mathrm{RHT}_{q,k} = \max_{\tau_i \in \tau_{q,k}} \{\mathrm{RHT}_{q,k,i}\} \quad (1)$$

where $\tau_{q,k}$ is the set of tasks on processor P_k sharing R_q.

The concept of resource hold times for composing multiple independently-developed real-time applications on uniprocessors has been studied previously [33], [34], however, on a multi-core (multiprocessor) platform we compute resource hold times for global resources in a different way (Section IV-A).

Definition 2: *Maximum Resource Wait Time* for a global resource R_q on processor P_k, denoted as $\mathrm{RWT}_{q,k}$, is the worst-case time that any task, τ_i, within P_k may wait for R_q (R_q is held by other processors) each time τ_i requests R_q. Processors waiting for a global resources are enqueued in a corresponding FIFO queue (Section III-B), hence the worst case occurs when all tasks within other processors have requested R_q before τ_i.

Definition 3: A processor, P_k, is abstracted and represented by an *interface* $I_k(Q_k, Z_k)$. In the interface, Q_k represents a set of l requirements where l is the number of tasks on P_k that request at least one global resource, i.e., each requirement is extracted from a task requesting one or more global resources (Section V). For a processor, P_k, to be schedulable all requirements in Q_k should be satisfied. A requirement, $r_s \in Q_k$, is an expression of the maximum resource wait times of one or more global resources, e.g., $r_1 \equiv \mathrm{RWT}_{1,k} + \mathrm{RWT}_{3,k} \leq 10$ indicates that the maximum waiting time for both global

resources R_1 and R_3 should not exceed 10 time units. The requirements (Q_k) of each processor is extracted from the schedulability analysis of the processor independently. Z_k in the interface is a set; $Z_k = \{\ldots, Z_{q,k}, \ldots\}$, where $Z_{q,k}$ is the Maximum Processor Locking Time ($MPLT$) which represents the maximum duration of time that any task τ_x on any other processor P_l ($l \neq k$) may be blocked by (tasks from) P_k each time τ_x requests R_q. I.e., whenever a task, τ_x, on a processor, P_l issues a request to a global resource, R_q, the maximum (collective) time that τ_x can be blocked on resource R_q by tasks on P_k, ($k \neq l$) is indicated by $Z_{q,k}$.

B. General Description of MSOS

The MSOS manages intra-processor and inter-processor global resource requests. Each global resource is associated with a global queue in which processors requesting the resource are enqueued. The processors are granted the resource in FIFO manner. For the global queue, FIFO fits well because prioritizing the systems on processors may not be the case, since during the development of a system, the priority of other systems may not be known. Within a processor the tasks requesting the global resource are enqueued in a local queue. We have studied and developed both priority-based and FIFO-based queues for handling intra-processor global resource requests.

Figure 1 shows an overview of the protocol. When the resource becomes available to the processor at the head of the global queue the eligible task (e.g., at the top of queue if FIFO is used) from the local queue within the processor can hold the resource.

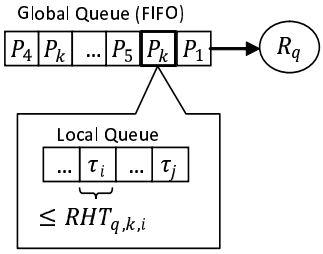


Fig. 1. An overview of MSOS. The processor at the head of a global queue receives the corresponding global resource and within the processor the eligible task in the local queue is granted to access the resource.

Considering that a processor, P_l, can block another processor, P_k, on a global resource, R_q, up to $Z_{q,l}$ time units each time (any task within) P_k requests the resource, the worst-case waiting time ($\mathrm{RWT}_{q,k}$) for P_k to wait until R_q becomes available is bounded by the sum of all $MPLT$'s of other processors on R_q:

$$\mathrm{RWT}_{q,k} = \sum_{P_l \neq P_k} Z_{q,l} \quad (2)$$

C. MSOS Rules

The MSOS rules are as follows:

Rule 1: Access to local resources is controlled by a uniprocessor synchronization protocol, e.g. PCP or SRP.

Rule 2: When a task, τ_i, within a processor, P_k, requests a global resource, R_q, the priority of τ_i is increased immediately to $\rho_i + \rho^{max}(P_k)$, where $\rho^{max}(P_k) = \max\{\rho_i | \tau_i \in P_k\}$. This means a task, τ_i, which is granted to access a global resource can only be delayed or preempted by higher priority tasks executing within a *global critical section* (gcs), in which they accesses a global resource, e.g., when a higher priority task, τ_x, which is blocked on a global resource R_l ($\neq R_q$) is granted to access R_l, it resumes and preempts τ_i while τ_i is accessing R_q. This bounds blocking times on a global resource as a function of only global critical sections. The concept that the blocking time on global resources should only depend on the duration of global critical sections is one of the principles in the existing multiprocessor synchronization protocols, e.g., MPCP, MSRP [21], [23].

Rule 3: When a task, τ_i, within a processor, P_k, requests a global resource, R_q, if R_q is not locked (i.e., both local and global queues are empty), τ_i accesses R_q. If R_q is locked, a placeholder for P_k is located in the global FIFO queue of R_q and τ_i is located in the local queue of R_q and then τ_i suspends itself.

Rule 4: When global resource R_q becomes available to processor P_k the eligible task within the local queue of R_q is resumed and granted the access to R_q. Depending on the type of local queue, the eligible task would be the one at the top of the local queue if FIFO is used and if the local queue is a prioritized queue the eligible task would be the highest priority task blocked on R_q. Note that using FIFO local queues, a task τ_i will always access a global resource R_q when the corresponding placeholder in the global queue (which is added by τ_i) is at the top of R_q's global queue. However, for a prioritized queue, it may not be the case because when a higher priority task is released, it will locate a placeholder in the global FIFO, but it may use the earlier placeholders added by lower priority tasks. This means, the lower priority task may use a later placeholder added by the higher priority tasks from P_k (Figure 2 shows an example of such case).

Rule 5: When a task, τ_i, on processor P_k releases a global resource, R_q, the placeholder of P_k from the top of the global FIFO queue will be removed and the resource becomes available to the processor whose placeholder is at the top of R_q's global queue.

IV. Schedulability Analysis

A. Computing Resource Hold Times

We now describe how to compute the global resource hold time by a task and consequently by a processor.

Lemma 1. *On a processor, P_k, any task, τ_i, that is granted to access a global resource, R_q, can be interfered (either delayed at the beginning or preempted) by at most one gcs per each higher priority task, τ_j (on P_k) in which τ_j accesses a global resource other than R_q.*

Proof: For τ_i, that is granted the access to a global resource, to be interfered by two gcs's (and more) of a higher priority task, τ_j (from the same processor), τ_j needs to enter

a non-critical section before entering the second gcs. On the other hand τ_i, which has been granted the access to a global resource, has a priority higher than any task that is not accessing a global resource (Rule 3). Considering that τ_i (granted to access a global resource) can only be preempted by other tasks within gcs's, τ_j will be preempted after exiting the first gcs and will not have any chance to enter the second gcs as long as τ_i has not exited its gcs. ■

Based on Lemma 1, the maximum interference to any gcs of task τ_i in which it accesses a global resource R_q, from the higher priority tasks located on the same processor, P_k, executing within their gcs's is denoted as $H_{i,q,k}$ and is computed as follows:

$$H_{i,q,k} = \sum_{\substack{\rho_i < \rho_j \\ \wedge\ R_l \in R^G_{P_k},\ l \neq q}} Cs_{j,l}$$

Consequently the resource hold time of global resource R_q by task τ_i is computed as follows:

$$\mathrm{RHT}_{q,k,i} = Cs_{i,q} + H_{i,q,k} \tag{3}$$

B. Blocking times under MSOS

In this section we describe the possible situations that a task τ_i can be blocked by other tasks on the same processor as well as by other processors. Each processor may contain a different system and may have a different scheduling policy. Thus the worst case blocking overhead from other processors on a global resource, R_q, introduced to any task, τ_i (each time τ_i requests R_q), within a processor, P_k, is abstracted by $\mathrm{RWT}_{q,k}$ (Definition 1). As shown in Equation 2, $\mathrm{RWT}_{q,k}$ depends on the $MPLT$'s of other processors on R_q. The value of $MPLT$ of a processor on each global resource is included in the interface. However, depending on the type of the local queues i.e., FIFO or prioritized, the $MPLT$ on a global resource is calculated differently:

a) FIFO-based local queues: In this case queueing on global resources are handled by FIFO queues. The maximum blocking time on a global resource, R_q, that tasks from P_k can introduce to any task, τ_x, located in a different processor each time τ_x requests R_q will happen when all tasks within P_k, sharing R_q, request R_q earlier than τ_x. Note that at any time instant there will be at most one placeholder per requesting task for P_k in the R_q's global FIFO queue because each task can add at most one placeholder and it cannot add another placeholder before releasing the previous one (i.e., a task cannot be in two critical sections at the same time). On the other hand the longest time that R_q can be locked by any task, τ_i, is $\mathrm{RHT}_{q,k,i}$ time units. Thus the $MPLT$ of P_k on R_q is calculated as follows:

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

b) Priority-based local queues: In this case queueing on global resources within processors is handled by prioritized queues; when a global resource, R_q, becomes available to a processor, P_k, the highest priority task, τ_h, is eligible to access R_q. Since FIFO is used for the global queue, similarly to the FIFO-based local queuing, the maximum blocking time that tasks from P_k on a global resource, R_q, can introduce to any task, τ_x, from a different processor each time τ_x requests R_q, will happen when all tasks within P_k, sharing R_q, request R_q earlier than τ_x. This means that the number of P_k's placeholders in R_q's global FIFO is equal to the number of the tasks within P_k sharing R_q. However, a higher priority task that requests R_q may use all these placeholders (as explained in Rule 4) Thus the the upper bound for $MPLT$ of P_k on R_q is calculated as follows:

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

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

Combining Equations 5 and 1, the result becomes as follows:

$$Z_{q,k} = |\tau_{q,k}|\, \mathrm{RHT}_{q,k} \tag{6}$$

The possible blocking terms that a task τ_i on a processor P_k may experience are as follows:

1) Local blocking due to local resources: Suppose n_i^G is the number of gcs's of τ_i. Each time τ_i is blocked on a global resource and suspended, a lower priority task τ_j may arrive and lock a local resource and may block τ_i when it resumes and after it releases the global resource. This scenario can happen up to n_i^G times. In addition, according to PCP (and SRP), task τ_i can be blocked on a local resource by at most one critical section of a lower priority task which has arrived before τ_i. On the other hand, τ_j can release at most $\lceil T_i/T_j \rceil$ jobs before the current job of τ_i finishes and each job can block τ_i's current job at most $n_j^L(\tau_i)$ times where $n_j^L(\tau_i)$ is the number of the critical sections in which τ_j requests local resources with ceiling higher than the priority of τ_i. This means τ_i can be blocked at most $\min\{n_i^G + 1, \sum_{\rho_j \leq \rho_i} \lceil T_i/T_j \rceil n_j^L(\tau_i)\}$ times by τ_j, on local resources. Thus, the upper bound blocking time on local resources (denoted by $B_{i,1}$) is calculated as follows:

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

where $ceil(R_l) = \max\{\rho_i |\ \tau_i\ \in\ \tau_{l,k}\}$

2) Local blocking due to global resources: Before τ_i arrives or each time it suspends on a global resource, a lower priority task τ_j may access a global resource (enters a gcs) and preempt τ_i in its non-gcs sections after it arrives or resumes. Since τ_i can suspend on global resources up to n_i^G times,

this type of preemption can occur at most $n_i^G + 1$ times (the additional preemption can happen by τ_j arriving and entering a gcs before τ_i arrives). On the other hand and similar to the case of local resources described above, τ_j can release at most $\lceil T_i/T_j \rceil$ jobs before the current job of τ_i finishes and each job can preempt τ_i's current job at most n_j^G times. Hence preemption from τ_j can happen at most $\min\{n_i^G + 1, \lceil T_i/T_j \rceil n_j^G\}$ times and thus the upper bound blocking time of this type, denoted by $B_{i,2}$ introduced by lower priority tasks is calculated as follows:

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

Equation 8 contains all the possible interference introduced to task τ_i from all gcs's of lower priority tasks including gcs's of tasks in which they share a global resource with τ_i.

Note that in both Equations 7 and 8, for simplicity we assume that the maximum blocking time (max function) will be introduced to τ_i each time it is blocked by a lower priority task. This may make the results of these equations pessimistic. More complex analysis can give tighter upper bounds.

3) Remote Blocking: This type of blocking occurs when task τ_i within processor P_k requests a global resource, R_q. Depending on the type of the local queues (FIFO-based or priority-based), the remote blocking is calculated differently:

a) Remote blocking with FIFO-based local queues: In this case, when τ_i is blocked on a global resource, R_q, it is added to the local FIFO of R_q. In the worst case, all tasks within P_k sharing R_q have requested R_q before τ_i and are already in the local FIFO. However, if the tasks that requested R_q before τ_i have priority lower than that of τ_i then their effect has been included in Equation 8. Otherwise if the tasks requesting R_q before τ_i have priority higher than that of τ_i the interference from these tasks to τ_i is considered as the normal preemption.

On the other hand, to compute the maximum remote blocking from other processors we assume that each time τ_i requests R_q, all tasks on other processors sharing R_q have requested R_q before τ_i. Since we use FIFO global queue, each task from a different processor can lock R_q at most once before τ_i accesses R_q, this means τ_i can be blocked up to $\mathrm{RWT}_{q,k}$ time units.

This scenario can happen each time τ_i requests R_q, i.e. up to $n_{i,q}^G$ times, where $n_{i,q}^G$ is the number of τ_i's global critical sections in which it requests R_q. Thus the remote blocking with FIFO local queues is calculated as follows:

$$B_{i,3} = \sum_{\substack{R_q \in R_{P_k}^G \\ \wedge\ \tau_i \in \tau_{q,k}}} n_{i,q}^G \, \mathrm{RWT}_{q,k} \tag{9}$$

In order to uniform the equation used to calculate the blocking independently on the type of local queue (Section V), we rewrite Equation 9 as follows:

$$B_{i,3} = \sum_{\substack{R_q \in R_{P_k}^G \\ \wedge\ \tau_i \in \tau_{q,k}}} \alpha_{i,q} \, \mathrm{RWT}_{q,k} \tag{10}$$

where $\alpha_{i,q} = n_{i,q}^G$.

b) Remote blocking with Priority-based local queues: In the case of using priority-based local queues, when processor P_k gets the resource R_q the highest priority task within processor P_k accesses R_q.

To calculate the remote blocking, first we derive an upper bound for the amount of remote blocking each local higher priority task, τ_x, can introduce to task τ_i (a task is called as a local task to τ_i if it is allocated on the same processor as τ_i). Figure 2 illustrates (in three stages) how τ_x may introduce remote blocking into τ_i. In the first stage, τ_i requests R_q and a placeholder for processor P_k is added to the end of global FIFO queue of R_q. τ_i would wait up to $\mathrm{RWT}_{q,k}$ time units to be eligible to access R_q if there was not any other task within P_k requesting R_q. However, as shown in the second stage, just before P_k gets R_q, task τ_x also requests R_q and another placeholder for P_k is added to the global queue. As shown in the third stage, when it becomes P_k's turn to get R_q, τ_x will access it (because it has a higher priority than τ_i) and τ_i's request is postponed to the second placeholder. This makes τ_i to wait additional $\mathrm{RWT}_{q,k}$ time units.

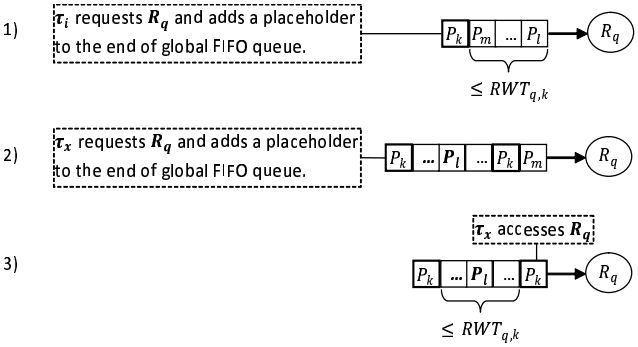


Fig. 2. Priority-based local queue.

Thus, each gcs of τ_x in which τ_x requests R_q adds up to $\mathrm{RWT}_{q,k}$ time units blocking to τ_i. On the other hand, the higher priority task τ_x may request R_q up to $\lceil T_i/T_x \rceil \, n_{x,q}^G$ times before τ_i finishes. Each time τ_x may hold R_q up to $\mathrm{RHT}_{q,k,x}$ time units, however, as all tasks contributing to $\mathrm{RHT}_{q,k,x}$ have higher priority than τ_i, the time during which τ_x holds R_q is considered as normal preemption and not blocking. Hence, the remote blocking of τ_i on R_q introduced by higher priority task τ_x, denoted by $\mathrm{RB}_i(R_q, \tau_x)$ is calculated as follows:

$$\mathrm{RB}_i(R_q, \tau_x) = \lceil T_i/T_x \rceil \, n_{x,q}^G \, \mathrm{RWT}_{q,k} \tag{11}$$

where $\tau_i \in \tau_{q,k}$.

The total blocking time of τ_i on R_q introduced by all higher priority tasks sharing R_q (denoted by $\mathrm{RB}_i(R_q)$) is calculated as follows:

$$\mathrm{RB}_i(R_q) = \sum_{\substack{\rho_i < \rho_x \\ \wedge\ \{\tau_x, \tau_i\} \subseteq \tau_{q,k}}} \mathrm{RB}_i(R_q, \tau_x) \tag{12}$$

In addition to the remote blocking on R_q that τ_i incurs because of higher priority tasks, each gcs of τ_i may wait up to $\mathrm{RWT}_{q,k}$ time units to be granted to access R_q, i.e., τ_i may incur this blocking even if no higher priority task (on τ_i's processor) requests R_q while τ_i is waiting for R_q. The upper bound for this blocking time is $n^G_{i,q}\,\mathrm{RWT}_{q,k}$ time units. Finally, the total remote blocking time of τ_i is as follows:

$$B_{i,3} = \sum_{\substack{R_q \in R^G_{P_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} (\mathrm{RB}_i(R_q) + n^G_{i,q}\,\mathrm{RWT}_{q,k}) \tag{13}$$

and by replacing Equations 11 and 12:

$$B_{i,3} = \sum_{\substack{R_q \in R^G_{P_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} \alpha_{i,q}\mathrm{RWT}_{q,k} \tag{14}$$

$$\text{where } \alpha_{i,q} = \sum_{\substack{\rho_i < \rho_x \\ \wedge\ \{\tau_x, \tau_i\} \subseteq \tau_{q,k}}} (\lceil T_i/T_x \rceil\, n^G_{x,q}) + n^G_{i,q}$$

Looking at the remote blocking for FIFO-based and priority-based local queues (Equations 9 and 13 respectively), it is shown that for the priority-based queues $B_{i,3}$ is always greater than that for FIFO-based queues. This means, using local FIFO queues (combined with global FIFO queues) always gives lower upper bounds for remote blocking time compared to using priority-based local queues. If the remote blocking is low (i.e., maximum resource wait times are small), it may seem that using local FIFO queues, whenever a higher priority task, τ_h, requests a global resource, R_q, τ_h can be delayed by all lower priority tasks that have requested the resource before τ_h which is not the case when using priority-based local queues. However, since the priority of these lower priority tasks requesting R_q is boosted (is higher than the base priority of τ_h) they will delay the execution of τ_h when τ_h is in its non-critical sections anyway, thus from the analysis point of view using priority-based local queues does not benefit higher priority tasks even though the remote blocking is very low.

C. Total blocking time

The total blocking time of τ_i is the summation of the three blocking terms:

$$B_i = B_{i,1} + B_{i,2} + B_{i,3} \tag{15}$$

Equations 10 and 14 show that $B_{i,3}$ is a function of maximum resource wait times (e.g., $\mathrm{RWT}_{q,k}$) of the global resources. Consequently B_i will also be a function of maximum resource wait times of global resources. Considering that $B_{i,1}$ and $B_{i,2}$ are constant numbers (i.e., they only depend on internal parameters), we can rewrite Equation 15 as follows:

$$B_i = \gamma_i + \sum_{\substack{R_q \in R^G_{P_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} \alpha_{i,q}\mathrm{RWT}_{q,k} \tag{16}$$

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

V. Extracting the requirements in the Interface

In this section we describe how to extract the requirements Q_k in the interface of a processor P_k from the schedulability analysis.

Each requirement in Q_k specifies a criteria on maximum resource wait times (Definition 2) of one or more global resources. We will show how to evaluate the requirement of each task τ_i accessing global shared resources.

Starting from the schedulability condition of τ_i, the maximum value of blocking time $mtbt_i$ that τ_i can tolerate without missing its deadline can be evaluated as follows.

τ_i is schedulable, using the fixed priority scheduling policy and executed in a single processor, if

$$0 < \exists t \leq T_i \quad \mathtt{rbf_{FP}}(i,t) \leq t, \tag{17}$$

where $\mathtt{rbf_{FP}}(i,t)$ 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:

$$\mathtt{rbf_{FP}}(i,t) = C_i + B_i + \sum_{\rho_i < \rho_j} (\lceil t/T_j \rceil C_j) \tag{18}$$

By substituting B_i by $mtbt_i$ in Equations 17 and 18, we can compute $mtbt_i$ as follows:

$$mtbt_i = \max_{0<t\leq T_i} (t - (C_i + \sum_{\rho_i < \rho_j} (\lceil t/T_j \rceil C_j))) \tag{19}$$

Note that it is not required to test all possible values of t in Equation 19, and only a bounded number of values of t that change $\mathtt{rbf_{FP}}(i,t)$ should be considered (see [35] for more details).

Equation 16 shows that the total blocking time of task τ_i is a function of maximum resource wait times of the global resources accessed by tasks on P_k. With the achieved $mtbt_i$ and Equation 16 we extract a requirement:

$$\gamma_i + \sum_{\substack{R_q \in R^G_{P_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} \alpha_{i,q}\mathrm{RWT}_{q,k} \leq mtbt_i \tag{20}$$

and

$$r_i \equiv \sum_{\substack{R_q \in R^G_{P_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} \alpha_{i,q}\mathrm{RWT}_{q,k} \leq mtbt_i - \gamma_i \tag{21}$$

Note that, Equation 15 can be used to evaluate B_i for both the tasks that share global resources and the tasks that do not share any global resource. For a task that does not share global resources, $B_{i,3} = 0$ and Equations 7 and 8 can be used to evaluate $B_{i,1}$ and $B_{i,2}$ assigning $n^G_i = 0$. Since the remote

blocking does not affect the schedulability of the tasks that do not share global resources, the schedulability test for these tasks can be done during the requirement extraction phase.

The schedulability of each processor is tested by its requirements. A processor P_k is schedulable if all the requirements in Q_k are satisfied and assuming that all tasks in P_k that do not share global resources are schedulable. To test the requirements in Q_k we need maximum resource wait times (e.g., $RWT_{q,k}$) of global resources accessed by tasks within P_k which are calculated using Equation 2.

VI. Experimental Evaluation

In this section we present our experimental results of the synchronization protocol (MSOS) together with MPCP and a variant of partitioned FMLP (called long FMLP). In fact, under FMLP global resources are divided into two groups; long resources and short resources, where tasks blocked on long resources suspend while tasks blocked on short resources spin (busy-wait). Dividing of global resources into long and short resources is user-defied and there is no method to efficiently decide which resources should be long or short. In a comparison of FMLP to other synchronization protocols (e.g., MPCP) in [29], FMLP is divided into two variants, i.e., long FMLP (all global resources are long) and short FMLP (all global resources are short). In this paper we consider only long FMLP. Note that MSOS can easily be extended to support spin blocking on global resources as well. However, since under spin blocking, a task blocked on a global resource executes non-preemptively, the worst case blocking times practically would be the same as short FMLP as well as the spin-based variant of MPCP [7].

As shown in Section IV priority-based local queues will always introduce more blocking overheads than FIFO-based local queues and hence FIFO-based local queues will always perform better. Therefore, we have only evaluated the MSOS with FIFO-based local queues.

We have developed MSOS as a synchronization protocol for independently-developed systems on multi-cores and thus we have abstracted overhead introduced to a processor from other resources due to resource sharing. However, to evaluate our protocol and investigate the performance loss due to the abstractions, we have performed experimental evaluations and compared the performance of MSOS to MPCP and FMLP.

It turned out that MSOS, that compared to MPCP and FMLP additionally supports independently-developed systems, does not come with any significant reduction in performance due to composability. In fact MSOS, depending on the settings of the systems under analysis, may even perform better than either one or both of the alternative synchronization protocols. Hence, besides offering the possibility of composability of independently-developed systems, MSOS can be used as a regular synchronization protocol for one single system distributed over processors as it offers relatively a simple schedulability analysis method compared to FMLP and MPCP. The schedulability analysis is simple in the sense that the local schedulability analysis for each processor is performed only once and in the case of changing a system allocated on a processor or introducing a new system (on a new processor), the schedulability analysis of other processors does not to be redone. The only test to be performed is to check that the requirements of all processors are still valid which is much simpler than performing the whole schedulability analysis for every processor.

A. Experiment Setup

To determine the performance of MSOS compared to other synchronization protocols, we tested the schedulability for each protocol using randomly generated systems. For schedulability analysis of MPCP and FMLP we used the methods described in [21] and [28] respectively to perform the analysis. The tasks within each system 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 system (processor), tasks were generated until the utilization of the system reached a cap or a maximum number of 40 tasks generated. The utilization cap ranged from $\{0.1,\ 0.2,\ 0,3,\ 0.4,\ 0.5,\ 0.6\}$. In both MPCP and FMLP, calculation of blocking times of each task (allocated on a processor) that shares global resources, depends on the timing attributes of remote tasks (allocated on other processor), e.g., task priorities. For each set of parameters, to achieve a global priority setting, all tasks from all systems were put in a single list and priorities were assigned to them based on Rate Monotonic (RM).

The resource sharing parameters were chosen as follows. The number of resources shared among all tasks (within all systems) was chosen from $\{5, 10, 15, 20\}$. The number of requests each task issued (i.e., the number of critical sections) was randomly chosen from $[0, \mathrm{CsNum}]$ where CsNum ranged from 1 to 6. The length of each critical section was randomly chosen from $[\mathrm{minL}, \mathrm{maxL}]$ which ranged from $\{[5\mu s, 10\mu s],\ [10\mu s, 20\mu s],\ [20\mu s, 40\mu s],\ [40\mu s, 80\mu s],\ [80\mu s, 160\mu s],\ [160\mu s, 320\mu s]\}$.

For each setting point we generated 1000 samples. For some settings we repeated the experiments 10 times which always yielded the same results, confirming that 1000 samples per each setting can be representative.

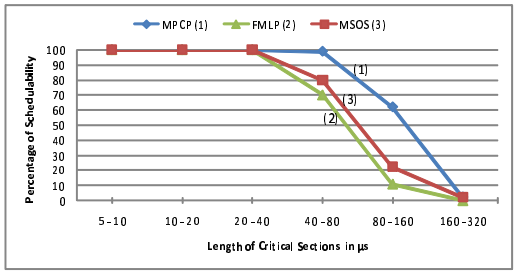
Preemption overhead: Preemption overhead which includes cache state loss as well as context switches, is a platform dependant and may differ in different platforms significantly. However, to determine the performance of different synchronization protocols and considering preemption overhead, besides running our experiments under no preemption overhead we ran them considering various preemption overheads (i.e., $10\mu s$, $20\mu s$, $40\mu s$ and $80\mu s$) as well. In all three protocols, tasks within their non-critical sections suffer from the same preemptions (i.e., from higher priority tasks and from lower priority tasks within gcs's), hence, we assume those overheads are counted for in the worst-case execution times of tasks. The difference is preemptions within gcs's, from which FMLP does

not suffer as tasks within their gcs's execute non-preemptively. Both MSOS and MPCP suffer from preemptions within gcs's. Preemption overhead makes a gcs longer which consequently punishes (blocks more) tasks (from other processors) requesting the same global resource as is accessed in the gcs. Under MPCP, a task within a gcs can be preempted by gcs's from both lower priority and higher priority tasks (for more details see [21]). Under MSOS, on the other hand, a task within a gcs can only be preempted by higher priority tasks within their gcs's.

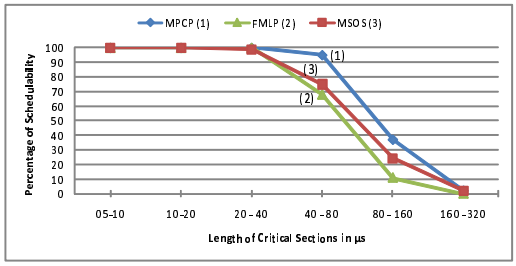
B. Results

In this section we present the evaluation results. Overall results show that in most cases MSOS performs better than at least one of the other protocols. We have performed experiments according to all different parameters (combining all the parameters, we achieve 2880 different settings), however it is not feasible to present all results in this paper. Thus, we present our observations according to three important factors that affected the protocols differently; the length of critical sections, the number of critical sections, and the preemption overhead. Regarding the length of critical sections, under no preemption overhead, as shown in Figure 3(a), for shorter critical sections all three protocols perform the same. However, as the length of critical sections is increased, MSOS mostly performs better than FMLP while MPCP exhibit the best performance. The reason is inherent in the different ways of handling queues for blocked tasks on global resources in each protocol. Under MPCP, tasks (from all processors) blocked on a global resource are enqueued in a priority-based queue while FMLP and MSOS use FIFO-based queues for blocked tasks on global resources. When the critical sections are relatively long, using FIFO queues may lead to starvation of higher priority tasks. This is why MPCP performs better than MSOS and FMLP for longer critical sections. FMLP, uses FIFO's more than MSOS; tasks waiting for a global resource are enqueued in a FIFO, and within a processor the tasks blocked on different global resources are also enqueued in FIFO manner. On the other hand in MSOS, within a processor tasks requesting different global resources are enqueued in a priority-based manner, i.e., a task that becomes eligible to access a global resource can preempt a lower priority task holding another global resource. This leads to less starvation of higher priority tasks requesting global resources specially for longer critical sections. Thus MSOS is affected by FIFO's less than FMLP but more than MPCP (MPCP does not use FIFO's at all).

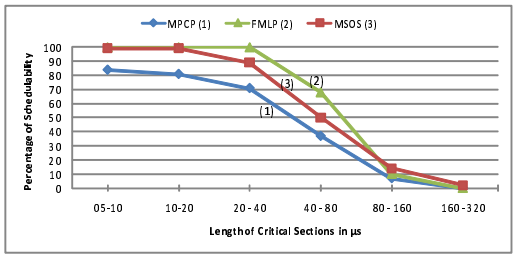
However, when considering preemption overhead as illustrated in Figures 3(b) and 3(c), the performance of MSOS and MPCP drops; with $40\mu s$ per preemption overhead (Figure 3(c)), FMLP mostly exhibits the best performance and MSOS performs better than MPCP. The rationale behind this is that the global critical sections under MPCP and MSOS suffer from preemption by other global critical sections which introduces preemption overhead into gcs's. This makes gcs's longer which leads to more blocking overhead for remote tasks. As



(a) Preemption overhead=0μs



(b) Preemption overhead=10μs

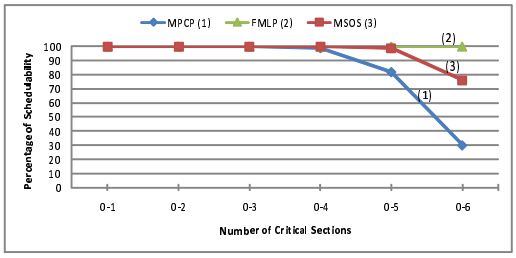


(c) Preemption overhead=40μs

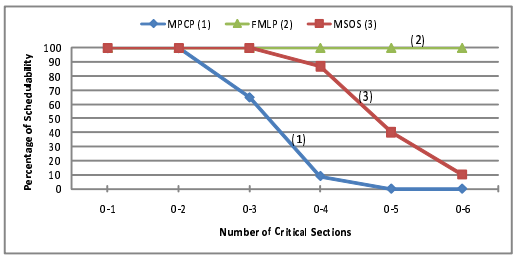
Fig. 3. Performance of synchronization protocols as the length of critical sections increases. Number of processors=8, utilization cap=0.3, number of resources=10, maximum number of critical sections per task=6.

confirmed by experiments, MPCP introduces more preemption overhead and thus leads to a lower schedulability of tasks. Increasing the number of critical sections when neglecting preemption overhead had similar effect as the length of critical sections on the performance of the protocols, i.e., for lower number of critical sections all three protocols perform almost the same and as the number of critical sections are incremented the performance of FMLP and MSOS drops faster than of MPCP, although MSOS performs better than FMLP. This is also the negative effect of using FIFO's when the number of critical sections becomes higher. However, with presence of preemption overhead (Figure 4), when the number of critical sections is increased the performance of MPCP and MSOS drops significantly fast. The reason is that the higher number of critical sections leads to higher number of preemptions within gcs's, hence more preemption overhead within the gcs's. This increases the length of gcs's and consequently causes longer blocking times on global resources. MSOS performs better than MPCP since MSOS suffers from less preemptions than MPCP.

Interestingly, in almost all cases MSOS performs better than

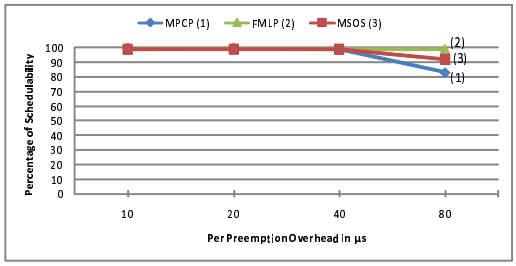


(a) Preemption overhead=10μs

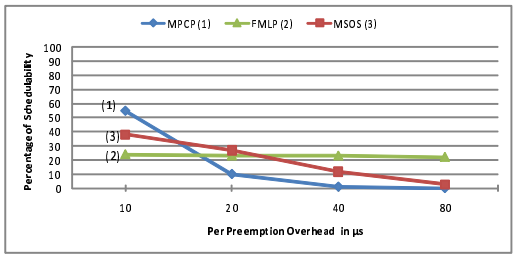


(b) Preemption overhead=40μs

Fig. 4. Performance of synchronization protocols as the number of critical sections increases. Number of processors=8, utilization cap=0.5, number of resources=10, length of critical sections =10-20 μs.



(a) utilization cap=0.3



(b) utilization cap=0.5

Fig. 5. Performance of synchronization protocols as the per preemption overhead increases. Number of processors=8, , number of resources=10, maximum number of critical sections=4, length of critical sections =40-80 μs.

at least one of the other two protocols. This is because MSOS uses a combination of FIFO-based and priority-based global resource accessing (i.e., FIFO's for global and local queues of global resources, and priority-based accessing different global resources) while FMLP and MPCP use either of them. Consequently, MSOS performs better than FMLP when the preemption overhead is low, and it performs better than MPCP when the preemption overhead is higher. Figure 5 shows the performance of the three protocols regarding different values of per preemption overhead. In our experiments, we also explored the performance of the protocols regarding other parameters, e.g., various number of processors and various utilization cap per each processor. Similar to the aforementioned results, under no preemption overhead, MSOS performs better than FMLP, and MPCP performs better than both as the number of processors and/or the utilization cap is increased. However, by increasing preemption overhead, although MSOS performs better than MPCP, the performance of both MSOS and MPCP drops significantly fast.

VII. Conclusion

In this paper, we have proposed a synchronization protocol which manages resource sharing among independently-developed systems on a multi-core platform where each system is allocated on a dedicated core.

In our protocol, each system is presented by an interface which abstracts the sharing of global resources in the system. Furthermore, we have derived schedulability analysis under our synchronization protocol. The systems within each processor may use a different scheduling policy and priority setting, however this does not affect the schedulability analysis of a system as these design decisions are abstracted by the interfaces. This offers the possibility of different systems to be developed independently and their schedulability analysis to be performed and abstracted in their interfaces. Hence, the protocol also simplifies migration of legacy real-time systems to multi-core architectures.

In this paper we focused on the common case of non-nested critical sections. However, MSOS can support properly-nested global critical sections by means of grouping global resources whose requests are nested (similar to FMLP). In this case a joint global FIFO queue will be used for all the resources of each group. However, this may introduce very large amount of blocking overhead.

We have performed experimental evaluations of our proposed synchronization protocol, MSOS, by means of comparing its performance against two existing synchronization protocols, i.e., MPCP and FMLP. The results show that in most cases MSOS performs better than at least one of other two protocols. Therefore, we believe that MSOS is a viable protocol for independently-developed systems on a multi-core platform.

In the future we plan to implement MSOS under real-time operating systems (RTOS) and investigate its performance. Another interesting future work is to study the multiprocessor hierarchical scheduling protocols for independent/semi-independent systems with presence of shared resources.

References

[1] M. Behnam, I. Shin, T. Nolte, and M. Nolin. SIRAP: a synchronization protocol for hierarchical resource sharing in real-time open systems. In *ACM & IEEE conference on Embedded software (EMSOFT'07)*, pages 279–288, 2007.

[2] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *IEEE Real-Time Systems Symposium (RTSS'06)*, pages 389–398, 2006.

[3] N. Fisher, M. Bertogna, and S. Baruah. The design of an edf-scheduled resource-sharing open environment. In *IEEE Real-Time Systems Symposium (RTSS'07)*, pages 83–92, 2004.

[4] 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.

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

[6] T. Baker. A comparison of global and partitioned EDF schedulability test for multiprocessors. Technical report, 2005.

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

[8] F. Nemati, M. Behnam, and T. Nolte. Sharing resources among independently-developed systems on multi-cores. *ACM SIGBED Review*, 8(1), 2011.

[9] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *IEEE Real-Time Systems Symposium (RTSS'03)*, pages 2–13, 2003.

[10] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *ACM Intl. Conference on Embedded Software (EMSOFT'04)*, pages 95–103, 2004.

[11] X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *IEEE Real-Time Systems Symposium (RTSS'02)*, pages 26–35, 2002.

[12] G. Lipari and S. K. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS'00)*, pages 166–175, 2000.

[13] S. Matic and T. A. Henzinger. Trading end-to-end latency for composability. In *IEEE Real-Time Systems Symposium (RTSS'05)*, pages 99–110, 2005.

[14] A. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS'01)*, pages 75–84, 2001.

[15] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *IEEE Real-Time Systems Symposium (RTSS'05)*, pages 389–398, 2005.

[16] J. M. Calandrino, J. H. Anderson, and D. P. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *IEEE Euromicro Conference on Real-time Systems (ECRTS'07)*, pages 247–258, 2007.

[17] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *IEEE Euromicro Conference on Real-time Systems (ECRTS'08)*, pages 181–190, 2008.

[18] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *Journal of IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[19] T. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, 1991.

[20] R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *IEEE Real-Time Systems Symposium (RTSS'88)*, pages 259–269, 1988.

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

[22] 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 *IEEE Real-Time and Embedded Technology Application Symposium (RTAS'03)*, pages 189–198, 2003.

[23] P. Gai, G. Lipari, and M. D. Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *IEEE Real-Time Systems Symposium (RTSS'01)*, pages 73–83, 2001.

[24] J. M. López, J. L. Díaz, and D. F. García. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Journal of Real-Time Systems*, 28(1):39–68, 2004.

[25] U. Devi, H. Leontyev, and J. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *IEEE Euromicro Conference on Real-time Systems (ECRTS'06)*, pages 75–84, 2006.

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

[27] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Synchronization on multiprocessors: To block or not to block, to suspend or spin? In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*, pages 342–353, 2008.

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

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

[30] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *IEEE Real-Time Systems Symposium (RTSS'09)*, pages 377–386, 2009.

[31] 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.

[32] C. Bialowas. Achieving Business Goals with Wind Rivers Multicore Software Solution. *Wind River white paper*.

[33] N. Fisher, M. Bertogna, and S. Baruah. Resource-locking durations in edf-scheduled systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'07)*, pages 91–100, 2007.

[34] M. Bertogna, N. Fisher, and S. Baruah. Static-priority scheduling and resource hold times. In *IEEE Parallel and Distributed Processing Symposium (IPDPS'07) Workshops*, pages 1–8, 2007.

[35] Enrico Bini and Giorgio C. Buttazzo. The space of rate monotonic schedulability. In *IEEE Real-Time Systems Symposium (RTSS'02)*, pages 169–178, 2002.