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 arbitrary 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 data needed for sharing 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 FMMP. 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 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 \textit{existing} systems. When migrating to multi-cores it should be possible that several of these systems coexist on a now shared multi-core platform. The (often independently developed) systems may have been developed with different techniques, e.g., several real-time systems that will coexist on a multi-core may perform 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.

The contributions of this paper are as follows.

A. Contributions

The contributions of this paper are as follows.

* 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.
• 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 in order for the processor to be schedulable. A global resource is a resource that is shared across processors. A requirement indicates that an expression (e.g., summation) of resource maximum wait times (i.e., the worst-case time that a processor may wait for a global resource to be available) of one or more global resources 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 observed results show that the compositability 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 (real-time open systems) 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], [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 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 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 [31] Brandenburg and Anderson have also proposed an asymptotically optimal suspension-aware protocol, 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. This will prevent 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 available schedulability analysis of the existing protocols (e.g., MPCP and FMLP) a processor needs timing attributes of tasks allocated on other processors which it shares resources with. MSOS, however, allows each system in a processor to use its own scheduling policy and it abstracts the resource sharing of 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 \( \tau_{P_k} \) and consists of \( n \) sporadic tasks, \( \tau_i(T_i, C_i, \rho_i, \{C_{S_i,q,p}\}) \) where \( T_i \) denotes the minimum inter-arrival time between two successive jobs of task \( \tau_i \) with worst-case execution time \( C_i \) and \( \rho_i \) as its priority. The tasks have implicit deadlines (deadlines are equal to periods), i.e., the deadline of any job of \( \tau_i \) is equal to \( T_i \). A task, \( \tau_i \), has a higher priority than another task, \( \tau_l \), if \( \rho_i > \rho_l \). 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 \( \tau_i \) requests resources in \( R_{P_k} \) is denoted by \( \{C_{S_i,q,p}\} \), where \( C_{S_i,q,p} \) is the worst-case execution time of the \( p \)th critical section of task \( \tau_i \) 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 \( \tau_i \) requests \( R_q \). In this paper, we focus on non-nested critical sections (the common case). A job of task \( \tau_i \), 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.

**Definition 1: Resource Hold Time of a global resource \( R_q \)** by task \( \tau_i \) on processor \( P_k \) is denoted by \( \text{RHT}_{q,k,i} \) and is the maximum duration of time the global resource \( R_q \) can be locked by \( \tau_i \). In other words, \( \text{RHT}_{q,k,i} \) is the the maximum time interval starting from the time instant \( \tau_i \) locks \( R_q \) and ending at the time instant \( \tau_i \) releases \( R_q \), which includes the time intervals \( \tau_i \) accesses \( R_q \) as well as the 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 \( \text{RHT}_{q,k} \), is as follows:

\[
\text{RHT}_{q,k} = \max_{\tau_i \in \tau_{P_k}} \{\text{RHT}_{q,k,i}\}
\]

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 \( \text{RWTT}_{q,k} \), is the worst-case time that any task, \( \tau_i \), within \( P_k \) may wait for \( R_q \) (\( R_q \) is held by other processors) each time \( \tau_i \) requests \( R_q \). Processors request global resources in FIFO queues (Section III-B), hence the worst case occurs when all tasks within other processors have requested \( R_q \) before \( \tau_i \).

**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., \( r_s \equiv \text{RWTT}_{1,k} + \text{RWTT}_{3,k} \leq 10 \) 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 \( \tau_{q,k} \) on any processor \( P_l \) \( (l \neq k) \) may be blocked by (tasks from)
$P_k$ each time $\tau_k$ requests $R_q$. I.e., whenever any task $\tau_k$ on any processor $P_l$ issues a request to global resource $R_q$, the maximum time $\tau_k$ may be blocked on resource $R_q$ by tasks on $P_k$, ($k \neq l$) is indicated by $Z_{q,k,l}$.

**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 since the eligible task (e.g., at the top of queue if resource becomes available to the processor at the head of queue). We have studied and developed both priority-based rules.

**Rule 2:** Tasks requesting the global resource are enqueued in a local processor synchronization protocol, e.g., PCP or SRP. Access to local resources is controlled by a uniprocessor.

**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, $\tau_i$, within a processor, $P_k$, requests 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 ($RWT_{q,k}$) for $P_k$ to wait until $R_q$ becomes available is bounded by the sum of all MPLT’s of other processors on $R_q$:

$$RWT_{q,k} = \sum_{P_l \neq P_k} Z_{q,l}$$

**Rule 3:** When a task, $\tau_i$, 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), $\tau_i$ accesses $R_q$. If $R_q$ is locked, a placeholder for $P_k$ is located in the local FIFO queue of $R_q$ and $\tau_i$ is located in the local queue of $R_q$ and then $\tau_i$ 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 to access $R_q$. 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 $\tau_i$ will always access a global resource $R_q$ when the corresponding placeholder in the global queue (which is added by $\tau_i$) 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, $\tau_i$, 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, $\tau_i$, 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, $\tau_j$ (on $P_k$) in which $\tau_j$ accesses a global resource other than $R_q$.

**Proof:** For $\tau_i$, that is granted to access a global resource, to be interfered by two gcs’s (and more) of a higher priority task, $\tau_j$ (from the same processor), $\tau_j$ needs to enter a non-critical section before entering the second gcs. On the other hand $\tau_i$, which has been granted to access a global resource, has a priority higher than any task within a non-critical section (Rule 3). Considering that $\tau_i$ (granted to access a global resource) can only be preempted by other tasks within gcs’s, $\tau_j$ will be preempted after exiting the first gcs and will not
have any chance to enter the second gcs as long as $\tau_i$ has not

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

$$H_{i,q,k} = \sum_{\rho_i < \rho_j} C_{s_{j,l}}$$

Consequently the resource hold time of global resource $R_q$ by task $\tau_i$ is computed as follows:

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

B. Blocking times under MSOS

In this section we describe the possible situations that a task $\tau_i$ can be blocked by other tasks on the same processor as well as by other processors. Each processor can 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, $\tau_i$ (each time $\tau_i$ requests $R_q$), within a processor, $P_k$, is abstracted by $RWT_{q,k}$ (Definition 1). As shown in Equation 2, $RWT_{q,k}$ 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, $\tau_x$, located in a different processor each time $\tau_x$ requests $R_q$ will happen when all tasks within $P_k$, sharing $R_q$, request $R_q$ earlier than $\tau_x$. 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., 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, $\tau_i$, is $RHT_{q,k,i}$ time units. Thus the $MPLT$ of $P_k$ on $R_q$ is calculated as follows:

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

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, $\tau_h$, is eligible to access $R_q$. Since FIFO is used for the local 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, $\tau_x$, from a different processor each time $\tau_x$ requests $R_q$, will happen when all tasks within $P_k$, sharing $R_q$, request $R_q$ earlier than $\tau_x$. 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:

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

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:

$$Z_{q,k} = |\tau_{q,k}| RHT_{q,k}$$

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

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

$$B_{i,1} = \min \{n^G_i + 1, \sum_{\rho_j < \rho_i} [T_i/T_j] n^G_j(\tau_i) \} \max_{\rho_i < \rho_j} \{ C_{s_{j,l}} \}$$

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

2) Local blocking due to global resources: Before $\tau_i$ arrives or each time it suspends on a global resource, a lower priority task $\tau_j$ may access a global resource (enters a gcs) and preempt $\tau_i$ in its non-gcs sections after it arrives or resumes. Since $\tau_i$ can suspend on global resources up to $n^G_i$ times, this type of preemption can occur at most $n^G_i + 1$ times (the additional preemption can happen by $\tau_j$ arriving and entering a gcs before $\tau_i$ arrives). On the other hand and similar to the case of local resources described above, $\tau_j$ can release at most $[T_i/T_j]$ jobs before the current job of $\tau_i$ finishes and each job can preempt $\tau_i$’s current job at most $n^G_j$ times. Hence preemption from $\tau_j$ can happen at most min $\{n^G_i + 1, [T_i/T_j] n^G_j \}$ 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:

$$B_{i,2} = \sum_{\rho_j < \rho_i \land \{\tau_i, \tau_j\} \subseteq \tau_{\rho_k}} \left( \min \{n_{i,j}^G + 1, [T_i/T_j]n_{j}^G\} \max_{R_k \in R_{\rho_k}} \{C_{s,j,q}\} \right)$$

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

Note that in both Equations 7 and 8, for simplicity we assume that the maximum blocking time (max function) will be introduced to $\tau_i$ 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 $\tau_i$ 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 $\tau_i$ 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 $\tau_i$ and are already in the local FIFO. However, if the tasks that requested $R_q$ before $\tau_i$ have priority lower than that of $\tau_i$, then their effect has been included in Equation 8. Otherwise if the tasks requesting $R_q$ before $\tau_i$ have priority higher than that of $\tau_i$, the interference from these tasks to $\tau_i$ is considered as the normal preemption.

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

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

$$B_{i,3} = \sum_{R_q \in R_{\rho_k} \land \{\tau_i, \tau_q\} \subseteq \tau_{\rho_k}} n_{i,q}^G \cdot RWT_{q,k}$$

In order that calculations in Section V can be applied to both priority-based and FIFO-based local queues, we rewrite Equation 9 as follows:

$$B_{i,3} = \sum_{R_q \in R_{\rho_k} \land \{\tau_i, \tau_q\} \subseteq \tau_{\rho_k}} \alpha_{i,q} \cdot RWT_{q,k}$$

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, $\tau_x$, can introduce to task $\tau_i$ (a task is called as a local task to $\tau_i$ if it is allocated on the same processor as $\tau_i$). Figure 2 illustrates (in three stages) how $\tau_x$ may introduce remote blocking into $\tau_i$. In the first stage, $\tau_i$ requests $R_q$ and a placeholder for processor $P_k$ is added to the end of global FIFO queue of $R_q$. $\tau_i$ would wait up to $RWT_{q,k}$ 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 $\tau_x$ 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 $R_q$, $\tau_x$ will access it (because it has a higher priority than $\tau_i$) and $\tau_i$’s request is postponed to the second placeholder. This makes $\tau_i$ to wait additional $RWT_{q,k}$ time units.

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

$$RB_{i}(R_q, \tau_x) = [T_i/T_x]n_{x,q}^G \cdot RWT_{q,k}$$

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

The total blocking time of $\tau_i$ on $R_q$ introduced by all higher priority tasks sharing $R_q$ (denoted by $RB_i(R_q)$) is calculated as follows:

$$RB_i(R_q) = \sum_{\rho_j < \rho_i \land \{\tau_x, \tau_i\} \subseteq \tau_{q,k}} RB_{i}(R_q, \tau_x)$$
In addition to the remote blocking on \( R_q \) that \( \tau_i \) incurs because of higher priority tasks, each \( g_c \) of \( \tau_i \) may wait up to \( \text{RWT}_{q,k} \) time units to be granted to access \( R_q \), i.e., \( \tau_i \) may incur this blocking even if no higher priority task (on \( \tau_i \)’s processor) requests \( R_q \) while \( \tau_i \) is waiting for \( R_q \). The upper bound for this blocking time is \( n^G_{i,q} \text{RWT}_{q,k} \) time units. Finally, the total remote blocking time of \( \tau_i \) is as follows:

\[
B_{i,3} = \sum_{R_q \in R^G_{P_k}} (\text{RB}_i(R_q) + n^G_{i,q} \text{RWT}_{q,k})
\]  

(13)

and by replacing Equations 11 and 12:

\[
B_{i,3} = \sum_{R_q \in R^G_{P_k}} \alpha_{i,q} \text{RWT}_{q,k}
\]  

(14)

where \( \alpha_{i,q} = \sum_{\rho_i < \rho_k \land \{\tau_{i,\tau_k}\} \subseteq \tau_{q,k}} ([T_i/T_x] n^G_{x,q} + n^G_{i,q}) \).

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, \( \tau_{h} \), requests a global resource, \( R_q \), \( \tau_{q} \) can be delayed by all lower priority tasks that have requested the resource before \( \tau_{h} \) 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 \( \tau_{h} \)), they will interfere to \( \tau_{h} \) when \( \tau_{h} \) 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 \( \tau_i \) is the summation of the three blocking terms:

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

(15)

Equations 10 and 14 show that \( B_{i,3} \) is a function of maximum resource wait times (e.g., \( \text{RWT}_{q,k} \)) 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:

\[
B_i = \gamma_i + \sum_{R_q \in R^G_{P_k}} \alpha_{i,q} \text{RWT}_{q,k}
\]  

(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 \( \tau_i \) accessing global shared resources.

Starting from schedulability condition of \( \tau_i \), the maximum value of blocking time \( \text{mtbt}_i \) that \( \tau_i \) can tolerate without missing its deadline can be evaluated as follows.

\[
\tau_i \text{ is schedulable, using the fixed priority scheduling policy and executed in a single processor, if}
0 < \exists t_i \leq T_i \ rbf_{FP}(i,t) \leq t_i
\]  

(17)

where \( rbf_{FP}(i,t) \) denotes request bound function of \( \tau_i \) which computes the maximum cumulative execution requests that could be generated from the time that \( \tau_i \) is released up to time \( t \), and is computed as follows.

\[
rbf_{FP}(i,t) = C_i + B_i + \sum_{\rho_i < \rho_j} ([t/T_j] C_j)
\]  

(18)

By substituting \( B_i \) by \( \text{mtbt}_i \) in Equations 17 and 18, we can compute \( \text{mtbt}_i \) as follows.

\[
\text{mtbt}_i = \max_{0 < t_i \leq T_i} (t - (C_i + \sum_{\rho_i < \rho_j} ([t/T_j] C_j)))
\]  

(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 \( rbf_{FP}(i,t) \) should be considered (see [35] for more details).

Equation 16 shows that the total blocking time of task \( \tau_i \) is a function of maximum resource wait times of the global resources accessed by tasks on \( P_k \). With the achieved \( \text{mtbt}_i \) and Equation 16 we extract a requirement:

\[
\gamma_i + \sum_{R_q \in R^G_{P_k}} \alpha_{i,q} \text{RWT}_{q,k} \leq \text{mtbt}_i
\]  

(20)

and

\[
\rho_i = \sum_{R_q \in R^G_{P_k}} \alpha_{i,q} \text{RWT}_{q,k} \leq \text{mtbt}_i - \gamma_i
\]  

(21)

Note that, Equation 16 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 \( n^G_{i,q} = 0 \). 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...
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 whereas tasks blocked on short resources spin (busy-wait). Dividing of global resources into long and short resources is user-defined 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 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 need 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 calculate the blocking times. 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 10\(\mu s\) and 100\(\mu s\). 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 \([0, \text{CsNum}]\) where \text{CsNum} ranged from 1 to 6. The length of each critical section was randomly chosen from \([\minL, \maxL]\) which ranged from \{5\(\mu s\), 10\(\mu s\), 10\(\mu s\), 20\(\mu s\), 40\(\mu s\), 40\(\mu s\), 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 gc's can only be preempted by higher priority tasks within their gc'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 μs 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 gc's. This makes gc's longer which leads to more blocking overhead for remote tasks. As 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 increases. Number of processors=8, utilization cap=0.3, number of resources=10, maximum number of critical sections per task=6.
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 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 to 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
