Sharing Resources among Independently-developed Systems on Multi-cores [*]

Farhang Nemati, Moris Behnam, Thomas Nolte
Mälardalen Real-Time Research Centre, Västerås, Sweden
{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 arbitrary priority settings. When using this synchronization protocol each processor is abstracted by an interface which consists of a set of requirements. A requirement depends only on the worst-case time the processor may wait for resources, i.e., the maximum number of times that the resources can be blocked by other processors. We have derived schedulability conditions for each processor and based on the analysis we extract the interface of the processor. In this paper, we focus on the cases when each system is allocated on a dedicated processor.

1 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 *existing* systems. When migrating to multi-cores it should be possible that several of these systems coexist 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 coexist on a multi-core may have different scheduling policies. However, when the systems coexist on the same multi-core platform they may share resources. 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 systems do not need to be aware of techniques used in other systems.

[*]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.

On the other hand, looking at industrial systems, to speed up their development, it is not uncommon that the 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 [4, 15, 21]. An interesting challenge is to extend this issue to multi-cores. Hence, new techniques are sought for scheduling semi-independent subsystems.

Looking at current state-of-the-art, two main approaches for scheduling real-time systems on multiprocessors (multi-cores) exist; global and partitioned scheduling [2, 3, 13, 17]. 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) and Earliest Deadline First (EDF). Partitioned scheduling policies have been used more often and are supported widely by commercial real-time operating systems [26], inherent in their simplicity, efficiency and predictability. Besides, the well studied uniprocessor scheduling and synchronization techniques can be reused for multiprocessors with fewer changes (or no changes).

In this paper, we focus on the partitioned scheduling policy and synchronization protocols. Allocation of independently developed systems on a multi-core architecture may have 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 concentrate on the first alternative in which each 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. These techniques usually transform each system into the abstraction of a 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 the same as the first alternative. However, extension to the third alternative remains as a future work.

1.1 Contributions

The contributions of this paper are as follows.

- We propose a *synchronization protocol* for resource sharing among independent systems on a multi-core system, each of which allocated on a dedicated core. We call the protocol as Multiprocessors Synchronization Protocol for Independent Systems (MSPIS).

- For a processor we derive the *resource hold time* of a global resource (i.e., a resource shared across processor) which is the maximum time that a resource can be held by a any task on the processor. We also derive the maximum *resource wait time* for a resource which is the worst-case time that a processor may wait for a resource to be available.

- We derive the *schedulability conditions* and based on that we extract an *interface* for each processor which abstracts the system on the processor. The interface is a set of requirements that should be satisfied for the processor to be schedulable. A requirement indicates that an expression (e.g., summation) of resource wait times of one or more global resources should not exceed a certain value. Thus, the requirements in the interface only depend on the resource maximum wait times and hence to obtain the interface of a system, the processor will not need any information from other processors, e.g., scheduling protocol or priority setting policy on other processors.

1.2 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 [1, 14, 16, 20, 25, 27, 28, 30, 31, 34, 38, 37]. Hierarchical scheduling has been studied and developed as a solution for these systems.

Hierarchical scheduling techniques have also been developed for multiprocessors (multi-cores) [12, 36]. However, the systems (called clusters in the mentioned papers) are assumed to be independent and do not share resources.

In the context of the synchronization protocols, PCP (Priority Ceiling Protocol) [35] and SRP (Stack-based Resource allocation Protocol) [2] are two of the best known methods for synchronization in uniprocessor systems.

For multiprocessor synchronization, Rajkumar et al. for the first time proposed a synchronization protocol in [33] which later [32] was called Distributed Priority Ceiling Protocol (DPCP). DPCP extends PCP to distributed systems and it can be used with shared memory multiprocessors. Rajkumar in [32] presented MPCP, which extends PCP to multiprocessors hence allowing for synchronization of tasks sharing mutually exclusive resources using partitioned FPS. Lakshmanan et al. [26] investigate and analyze 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 MSPIS the schedulability test of a system on a processor is represented as its interface (requirements) which can be obtained without any information from other systems (even before the systems are developed) which will be allocated on other processor.

Gai et al. [23, 24] present 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. [29] present an implementation of SRP under P-EDF. Devi et al. [18] present 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. [8] present 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 [10]. Brandenburg and Anderson in [9] have extended partitioned FMLP to the fixed priority scheduling policy and derived a schedulability test for it. In a later work [11], the same authors have compared DPCP, MPCP and FMLP.

In all the aforementioned existing synchronization protocols on multi-cores (multiprocessors) 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) is used. MSPIS, however, allows each processor use its own scheduling policy. Recently, in industry, co-existing of several separated systems on a multi-core platform (called virtualization) has been considered to reduce the hardware costs [6]. MSPIS seems to be a natural fit for synchronization under virtualization of real-time systems on multi-cores.

Recently, Easwaran and Andersson have proposed a synchronization protocol [19] under the global fixed priority scheduling protocol. In this paper, for the first time, the authors have derived schedulability analysis of the Priority Inheritance Protocol (PIP) under global scheduling algorithms.

2 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., a 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.

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. A task, τ_i has a higher priority than another task, τ_j, if $\rho_i > \rho_j$. 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 sets 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 p^{th} critical section of task τ_i in which the task locks resource $R_q \in R_{P_k}$ and $|Cs_{i,q,p}|$ indicates the worst case execution time of the critical section. In this paper, we focus on non-nested critical sections (the common case). The deadline of each job is equal to T_i. A job of task τ_i, is specified by J_i. The utilization factor of task τ_i is denoted by u_i where $u_i = C_i/T_i$.

3 The Multiprocessors Synchronization Protocol for Independent Systems (MSPIS)

3.1 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 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. 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:

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

The concept of resource hold times for composing multiple independently-developed real-time applications on uniprocessors has been studied previously [22, 5], however, on a multi-core (multiprocessor) platform we compute resource hold times for global resources in a different way (Section 3.4.1).

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 R_q is held by other processors than P_k, i.e., $\mathrm{RWT}_{q,k}$ is the maximum duration of time in which R_q is not available to any task on P_k.

Definition 3: A processor, P_k, is represented by an *interface* Q_k which is 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 4). For a processor, P_k, to be schedulable all requirements in Q_k should be satisfied. A requirement, $r_s \in Q_k$, is 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 interface (requirements) of each processor is extracted from the schedulability analysis of the processor independently.

3.2 General Description of MSPIS

The MSPIS manages intra-processor and inter-processor global resource requests; the tasks within a processor requesting a global resource are enqueued in a local FIFO queue (intra-processor queuing) and the processors requesting the global resource are enqueued in a global FIFO queue (inter-processor queuing). It is also possible to use a local prioritized queue instead of FIFO, but the schedulability analysis will be more complex. On the other hand no concrete research results have shown which type of queues is absolutely better for queuing on global resources. For the global queue, however, FIFO fits well since prioritizing the systems on processors may not make sense. Besides, the maximum resource wait times may not easily be calculated if the prioritized global queue is used. Figure 1 shows an overview of how the protocol works. Each processor can hold a global resource.

Definition 4: A global resource, R_q, is available to a processor, P_k, for at most $Z_{q,k}$ time units called *budget* which should be greater or equal to $RHT_{q,k}$, i.e., $RHT_{q,k} \leq Z_{q,k}$. The resource is available to the processor at the head of the global queue and the processor holds the resource until the budget is depleted or unless there are no tasks in the local queue. Considering each processor, P_k, has a limited budget ($Z_{q,k}$) on a global resource, R_q, the worst-case waiting time ($\mathrm{RWT}_{q,k}$) for P_k to wait until R_q becomes available is bounded as a summation of R_q budgets of other processors sharing R_q:

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

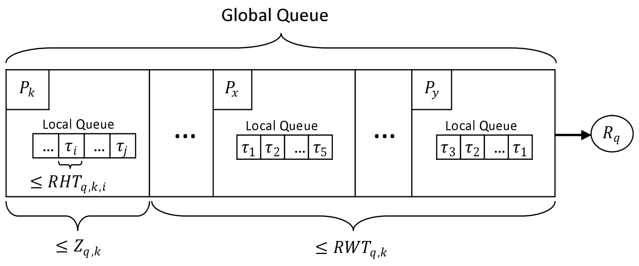

Figure 1: MSPIS

3.3 MSPIS Rules

The MSPIS rules are as follows:
Rule 1: Access to local resources is controlled by a uniprocessor synchronization protocol, e.g. PCP or SRP.
Rule 2: For a processor, P_k, a ceiling is defined as $ceil(P_k) = \max\{\rho_i | \tau_i \in P_k\}$ if PCP is used for local resources (in this paper, we assume that PCP is used). However, in the case of using SRP for local resources the ceiling of the processor is defined as $ceil(P_k) = \max\{\lambda_i | \tau_i \in P_k\}$ where λ_i is the static preemption level of τ_i.
Rule 3: A task, τ_i, within a *global critical section* (gcs) in which τ_i accesses a global resource can only be preempted by another task within a gcs. This bounds blocking times on a global resource as a function of only global critical sections. The concept that the blocking time on global resource should only depend on the duration of global critical sections is a basic issue in the existing multiprocessor synchronization protocols, e.g., MPCP, MSRP [32, 24]. To satisfy this criteria the priority of a task within a gcs has to be greater than $ceil(P_k)$. Thus the priority of a task, τ_i within a gcs in which τ_i accesses a global resource is raised to $\rho_i + ceil(P_k)$. This means that a task within a gcs can only be preempted by a higher priority task within a gcs.
Rule 4: In this paper, for a processor P_k accessing a global resource we assume $Z_{q,k} = RHT_{q,k}$ (how to fairly distribute the budget among processors remains as a future work). The processors requesting a global resource are located in the global FIFO queue of the resource; when a task on a processor, P_k, requests a global resource, R_q, if R_q is not available to P_k it will be added to the global queue (if P_k is not already in the queue). When R_q becomes available to P_k it can lock R_k for at most $Z_{q,k}$ time units.
Rule 5: When a global resource, R_q, is available to processor P_k the task from the top of the local FIFO queue of R_q locks it. The total duration of locking R_q should not exceed $Z_{q,k}$, hence there should exist a runtime mechanism to figure out the *remaining budget* of any global resource, R_q, at any time instant, t. We denote the remaining budget at time instant t by $Z'_{q,k}(t)$. When a global resource, R_q at time instant t becomes available to task τ_i it will be eligible to access the resource if $\mathrm{RHT}_{q,k,i} \leq Z'_{q,k}(t)$ otherwise R_q is released and becomes available to the next processor in the global queue. In this case P_k is deleted from the head and added to the end of the global queue, and τ_i will continue suspending and remains at the top of the local queue until next time R_q becomes available to P_k. Inspired by a similar definition in [4] we call the extra overhead introduced to τ_i by this suspension as *self-blocking time*. When τ_i requests R_q, if it is not available to the processor or if it is locked by another task on the processor, τ_i suspends and is added to the end of R_q's local queue.
Rule 6: When τ_i on P_k releases global resource R_q at time instant t, if there is no more tasks waiting in R_q's local queue, P_k releases R_q and P_k is deleted from R_q's global queue even if the P_k's budget of R_q is not finished (i.e., $Z'_{q,k}(t) > 0$).

3.4 Schedulability Analysis

3.4.1 Computing Resource Hold Times

Supposing a task set on a processor is schedulable, we describe how to compute the global resource hold time by a task and consequently by a processor.

LEMMA 1: A task, τ_i, within a gcs accessing a global resource, R_q, can be interfered with at most one gcs per each higher priority task, τ_j in which τ_j accesses a global resource other than R_q.

Proof: For a gcs of τ_i to be interferenced by two gcses (and more) of a higher priority task, τ_j, τ_j needs to enter a non-critical section before entering the second gcs. On the other hand, τ_i within a gcs has a priority higher than any task within a non-critical section (Rule 3). Considering that τ_i within the gcs can only be preempted by other tasks within gcses, 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 LEMMA1, the maximum interference from the higher priority tasks (within gcses) to any gcs of task τ_i in which it accesses a global resource R_q is denoted as $H_{i,q}$ and is computed as follows.

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

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

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

3.4.2 Blocking times under MSPIS

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 can contain a different system and may have a different scheduling policy.

Thus the worst case blocking overhead (i.e., *remote blocking*) from other processors on a global resource introduced to tasks on a processor, P_k, is abstracted by $\mathrm{RWT}_{q,k}$ (Definition 1).

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

1. Suppose n_i^G is the number of gcses of τ_i. Each time τ_i is blocked on a global resource and suspended, a lower priority task may arrive and lock a local resource and may block τ_i when it resumes. This scenario can happen up to n_i^G times. On the other hand, 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. This means that τ_i can be blocked at most $n_i^G + 1$ times on local resources. Thus, the worst case blocking time on local resources (denoted by $B_{i,1}$) is calculated as follows:

$$B_{i,1} = (n_i^G + 1) \max_{\substack{\rho_j \le \rho_i \\ R_l \in R^L_{P_k},\ \rho_i \le ceil(R_l) \\ \forall p}} \{|Cs_{j,l,p}|\} \tag{4}$$

where $ceil(R_l) = \max\{\rho_i |\ \tau_i \text{ uses } R_l\}$

2. 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 τ_j can preempt τ_i at most n_j^G times. Hence preemption from τ_j can happen at most $\min\{n_i^G + 1, n_j^G\}$ times and thus the worst case blocking time introduced by τ_j is $\min\{n_i^G + 1, n_j^G\} \max_{\substack{R_q \in R^G_{P_k} \\ \forall p}} \{|Cs_{j,q,p}|\}$. Thus, the worst case blocking time of this type, denoted by $B_{i,2}$ introduced by lower priority tasks is calculated as follows:

$$B_{i,2} = \sum_{\rho_j \le \rho_i} (\min\{n_i^G + 1, n_j^G\} \max_{\substack{R_q \in R^G_{P_k} \\ \forall p}} \{|Cs_{j,q,p}|\}) \tag{5}$$

3. When a global resource, R_q, is available to P_k (i.e., $Z'_{q,k}(t) \ge 0$), the task τ_j at the top of R_q's local queue will hold R_q at most for $\mathrm{RHT}_{q,k,j}$ time units if $\mathrm{RHT}_{q,k,j} \le Z'_{q,k}(t)$ otherwise it self-blocks and suspends. In the case of self-blocking, τ_j will remain at the top of the local queue for an extra duration of time (i.e., wastes extra time units of P_k's budget for R_q) up to $\mathrm{RHT}_{q,k,j}$. Thus each task in the local queue may consume up to $2\mathrm{RHT}_{q,k,j}$ of the budget ($Z_{q,k}$). The tasks in the local queue located before τ_i may consume several instances of the budget. Each time the budget is consumed the tasks in the local queue will wait for another $RWT_{q,k}$ time units. When eventually τ_i is at the top of the local queue the budget may not be enough and τ_i has to wait for an additional $RWT_{q,k}$ time units. Thus, the maximum number of budgets needed until τ_i accesses R_q is

$$\lceil \sum_{\substack{\tau_j \in \tau(R_q, P_K), \\ \tau_j \ne \tau_i}} 2\mathrm{RHT}_{q,k,j}/Z_{q,k} \rceil$$

where $\tau(R_q, P_K)$ is the set of tasks on processor P_k sharing R_q. Hence, the worst case blocking time of τ_i each time it requests R_q is upper bounded by

$$\lceil \sum_{\substack{\tau_j \in \tau(R_q, P_K), \\ \tau_j \ne \tau_i}} 2\mathrm{RHT}_{q,k,j}/Z_{q,k} \rceil RWT_{q,k}$$

This scenario may occur each time τ_i requests R_q, hence, the total blocking time of τ_i on R_q, denoted by $B_i(R_q)$ is as follows:

$$B_i(R_q) = n_{i,q}^G \lceil \sum_{\substack{\tau_j \in \tau(R_q, P_K), \\ \tau_j \ne \tau_i}} 2\mathrm{RHT}_{q,k,j}/Z_{q,k} \rceil RWT_{q,k} \tag{6}$$

The total blocking time of this type, denoted by $B_{i,3}$ is calculated as follows:

$$B_{i,3} = \sum_{R_q \in R^G_{P_k}} B_i(R_q)$$

or

$$B_{i,3} = \sum_{R_q \in R^G_{P_k}} \alpha_{q,i} \mathrm{RWT}_{q,k} \tag{7}$$

where $\alpha_{q,i} = n_{i,q}^G \lceil \sum_{\substack{\tau_j \in \tau(R_q, P_K), \\ \tau_j \ne \tau_i}} 2\mathrm{RHT}_{q,k,j}/Z_{q,k} \rceil$ which is a constant number.

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

$$B_i = B_{i,1} + B_{i,2} + B_{i,3} \tag{8}$$

Equation 7 shows that $B_{i,3}$ is a function of maximum resource wait times (e.g., $\mathrm{RWT}_{q,k}$) of the global resources accessed by tasks on P_k. 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, we can rewrite Equation 8 as follows:

$$B_i = \gamma_i + \sum_{R_q \in R^G_{P_k}} \alpha_{q,i} \mathrm{RWT}_{q,k} \tag{9}$$

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

4 Extracting the Interface of a Processor

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

Each requirement in the interface 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 schedulbaility 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

$$0 < \exists t \leq T_i \ \ \mathtt{rbf}_{\mathtt{FP}}(i,t) \leq t, \tag{10}$$

where $\mathtt{rbf}_{\mathtt{FP}}(i,t)$ 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.

$$\mathtt{rbf}_{\mathtt{FP}}(i,t) = C_i + B_i + \sum_{\rho_i \leq \rho_j} (\lceil t/T_j \rceil C_j) \tag{11}$$

By substituting B_i by $mtbt_i$ in Equations 10 and 11, we can compute $mtbt_i$ as follows.

$$mtbt_i = \max_{0<t\leq T_i} (t - (C_i + \sum_{\rho_i \leq \rho_j} (\lceil t/T_j \rceil C_j))) \tag{12}$$

Note that it is not required to test all possible values of t in Equation 12, and only a bounded number of values of t that change $\mathtt{rbf}_{\mathtt{FP}}(i,t)$ should be considered (see [7] for more details).

Equation 9 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 9 we extract a requirement:

$$\gamma_i + \sum_{R_q \in R^G_{P_k}} \alpha_{q,i} \mathrm{RWT}_{q,k} \leq mtbt_i \tag{13}$$

and

$$r_i \equiv \sum_{R_q \in R^G_{P_k}} \alpha_{q,i} \mathrm{RWT}_{q,k} \leq mtbt_i - \gamma_i \tag{14}$$

The schedulability of each processor is tested by its interface. A processor P_k is schedulable if all the requirements in Q_k are satisfied. To test the requirements in Q_k we need maximum resource wait times (e.g., $RWT_{q,k}$) of global resources accessed by tasks on P_k. In this paper, we have assumed $Z_{q,k} = RHT_{q,k}$, hence (according to Equation 2) for each global resource R_q the maximum resource wait time is calculated as follows:

$$\mathrm{RWT}_{q,k} = \sum_{P_l \neq P_k} RHT_{q,l} \tag{15}$$

5 An Example

In this section we present a simple example to illustrate how MSPIS work.

The multiprocessor is comprised of two processors (P_1 and P_2) and each processor contains a system (task set). The systems share two global resources (R_1 and R_2). The tasks within the system on P_1 also share a local resource (R_3). Figure 2 shows the task sets on each processor. A lower index of a task indicates a higher priority, e.g., $\rho_1 > \rho_2$. In this example each task accesses a resource once, i.e., a task has one critical section for each resource it accesses. The length of critical sections are shown in Figure 2.

P_1			
	R_1	R_2	R_3
τ_1	2	-	3
τ_2	3	-	-
τ_3	-	5	-
τ_4	5	-	2
τ_5	-	-	2

P_2		
	R_1	R_2
τ'_1	3	2
τ'_2	2	1
τ'_3	4	-

Figure 2: Task sets

Using Equation 3, resource hold times of global resources R_1 and R_2 by tasks accessing them are as follows:
(1) on processor P_1: $\mathrm{RHT}_{1,1,1} = 2$, $\mathrm{RHT}_{1,1,2} = 7$, $\mathrm{RHT}_{1,1,4} = 14$, $\mathrm{RHT}_{2,1,3} = 10$,
(2) on processor P_2: $\mathrm{RHT}_{1,2,1} = 3$, $\mathrm{RHT}_{1,2,2} = 4$, $\mathrm{RHT}_{1,1,3} = 7$, $\mathrm{RHT}_{2,2,1} = 2$, $\mathrm{RHT}_{2,2,2} = 4$.
Consequently, using Equation 1 resource hold times of the global resources on each processor are as follows: $\mathrm{RHT}_{1,1} = 14$, $\mathrm{RHT}_{2,1} = 10$, $\mathrm{RHT}_{1,2} = 7$, and $\mathrm{RHT}_{2,2} = 4$. Figure 3 illustrates a snapshot of the tasks initial execution on processors. The example shows the interaction between tasks and their corresponding blocking on resources:
At time instant 1, τ'_2 requests R_1, and since R_1 is free it becomes available to P_2 and τ'_2 will access the resource. At this time instant τ_2 on P_1 requests R_1 and P_1 is put into the global queue of R_1. Since R_1 is not available to P_1, τ_2 is blocked, is put into the local queue of R_1 and suspends. At time instant 3, on P_2, τ'_2 releases R_1. At the same time instant τ'_1 requests and accesses R_1 because R_1 is still available to P_2 (the budget is not finished) and τ'_1 is eligible to access R_1, i.e., at time instant 3, $Z'_{1,2}(3) = 7 - 2 = 5$ and $\mathrm{RHT}_{1,2,1} \leq Z'_{1,2}(3)$. At this time instant on P_1, τ_1 requests R_1 but the resource is not available to P_1, hence it is blocked, put into the local queue of R_1 and suspends. At this time P_1 is not put into the global queue of R_1 since P_1 is already in the global queue. Similarly at time instant 4, τ_4 requests R_1 and is blocked, put into the local queue and suspends. At instant 5, τ_3 accesses R_2 (R_2 becomes available to P_1). At instant 6, R_1 becomes available to P_1 and τ_2 (since it is at the head of the local queue of R_1) preempts τ_3 and accesses R_1. At instant 7, τ'_1 requests R_2. R_2 is

not available, hence P_2 is added to the global queue of R_2 and τ'_1 is added to the local queue and suspends. At instant 9, τ'_2 also requests R_2 and is added to the local queue and suspends. At time instant 11, τ_1 releases R_1, at this instant τ_4 is not eligible to access R_1 ($\text{RHT}_{1,1,4} > Z'_{1,1}(11)$) and should wait until next time R_1 becomes available to P_1. At this instant P_1 is deleted from the head of and is added to the end of the global queue of R_1 and R_1 becomes available to P_2 and is accessed by τ'_3. At instant 14, R_2 becomes available to P_2 and τ'_1 preempts τ'_3 and accesses R_2. Similarly at instant 16, τ'_2 accesses R_2. At instant 17, τ'_3 resumes and continues accessing R_1. At instant 19, τ'_3 releases R_1 and since there is no more tasks in the local queue, R_1 becomes available to P_1 and is accessed by τ_4.

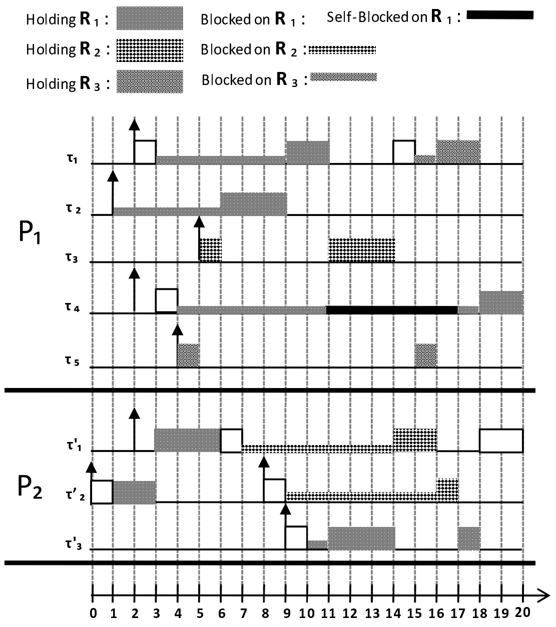

Figure 3: MSPIS

6 Conclusion

In this paper, we have discussed that the emerging of multi-core architectures has arisen the need for methods for migrating existing real-time software systems to these platforms. The methods should be developed to facilitate co-existing of several independent/semi-independent real-time systems on the same multi-core platform in the presence of shared resources. While considerable work has been done in the uniprocessor domain, we are not aware of any work to support independently-developed real-time systems on the a multiprocessor (multi-core) platform with shared resources.

In this paper, we have proposed a synchronization protocol which manages resource sharing among different systems. We have mentioned three possibilities for coexistence of such systems on a multi-core architecture; (i) a system is allocated on one processor, i.e., a processor contains only one system, (ii) several systems can be allocated on the same processor, (iii) a system is distributed on several processors. Our proposed synchronization protocol supports the first alternative. However, by using the uniprocessor techniques for open systems the second alternative can be transformed to the first alternative. Thus by combining our protocol and uniprocessor protocols, e.g., SIRAP [4] which is a protocol for sharing resources among semi-independent systems (subsystems), both the first and second alternative can be supported. Extension to the third alternative remains as a future work.

Furthermore, we have derived schedulability analysis under our synchronization protocol and defined an interface for each processor as a set of requirements. A requirement is a function of worst-case times that the processor may wait for global resources. The processors may use different scheduling policies and priority settings, however this does not affect the schedulability analysis of a processor as processors are abstracted by their interfaces.

Each processor has a budget for each global resource which is the maximum duration of time that the processor can hold a global resource. In this paper, we have set this budget to its minimum value which is the worst-case time any task on the processor can lock the resource. In the future we will work on optimization of distributing budgets among processors.

Another interesting future work is to study the multiprocessor hierarchical scheduling protocols for independent/semi-independent systems with presence of shared resources.

References

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

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

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

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

[5] 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.

[6] C. Bialowas. Achieving Business Goals with Wind Rivers Multicore Software Solution. *Wind River white paper, 2010.*

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

[8] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In

IEEE Intl. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA'07), pages 47–56, 2007.

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

[10] 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.

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

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

[13] 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.

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

[15] 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.

[16] Z. Deng and J.-S. Liu. Scheduling real-time applications in an open environment. In *IEEE Real-Time Systems Symposium (RTSS'97)*, pages 308–319, 1997.

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

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

[19] 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.

[20] 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.

[21] 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, 2007.

[22] 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.

[23] 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.

[24] 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.

[25] T. W. Kuo and C. H. Li. A fixed-priority-driven open environment for real-time applications. In *IEEE Real-Time Systems Symposium (RTSS'99)*, pages 256–267, 1999.

[26] 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.

[27] 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.

[28] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *15th Euromicro Conf. on Real-Time Systems (ECRTS'03)*, pages 151–158, 2003.

[29] 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.

[30] 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.

[31] 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.

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

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

[34] S. Saewong, R. R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of hierarhical fixed-priority scheduling. In *Euromicro Conf. on Real-Time Systems (ECRTS'02)*, pages 152–160, 2002.

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

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

[37] I. Shin and I. Lee. Compositional real-time scheduling framework. In *IEEE Real-Time Systems Symposium (RTSS'04)*, pages 57–67, 2004.

[38] 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.