IPP HURRAY!

www.hurray.isep.ipp.pt

Technical Report

Resource Sharing in Global Fixed-Priority Preemptive Multiprocessor Scheduling

Arvind Easwaran

Bjorn Andersson

HURRAY-TR-090908

Version: 0

Date: 09-14-2009

Resource Sharing in Global Fixed-Priority Preemptive Multiprocessor Scheduling

Arvind Easwaran, Bjorn Andersson

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail:

http://www.hurray.isep.ipp.pt

Abstract

In this paper we consider global fixed-priority preemptive multiprocessor scheduling of constrained-deadline sporadic tasks that share resources in a non-nested manner. We develop a novel resource-sharing protocol and a corresponding schedulability test for this system. We also develop the first schedulability analysis of priority inheritance protocol for the aforementioned system. Finally, we show that these protocols are efficient (based on the developed schedulability tests) for a class of priority-assignments called reasonable priority-assignments.

Resource Sharing in Global Fixed-Priority Preemptive Multiprocessor Scheduling

Arvind Easwaran
CISTER/IPP-HURRAY,
Polytechnic Institute of Porto, Portugal
aen@isep.ipp.pt

Björn Andersson
CISTER/IPP-HURRAY,
Polytechnic Institute of Porto, Portugal
bandersson@dei.isep.ipp.pt

***Abstract*—In this paper we consider global fixed-priority preemptive multiprocessor scheduling of constrained-deadline sporadic tasks that share resources in a non-nested manner. We develop a novel resource-sharing protocol and a corresponding schedulability test for this system. We also develop the first schedulability analysis of priority inheritance protocol for the aforementioned system. Finally, we show that these protocols are efficient (based on the developed schedulability tests) for a class of priority-assignments called *reasonable* priority-assignments.**

I. Introduction

Multicore processors are today standard building blocks in embedded computer systems but their use for applications with real-time requirements is non-trivial. This is because although a comprehensive toolbox of scheduling theories are available for a computer with a single processor, such a comprehensive toolbox is currently not available for multicores. Real-time applications tend to be organized as a set of concurrently executing tasks which need to share resources (for example data structures or I/O devices). Clearly, a resource-sharing protocol is a crucial component in multicore-based embedded real-time systems. Based on empirical findings about real-world applications [6] and based on the properties exhibited by the resource-sharing protocols that are used successfully in uniprocessor systems, we believe that such protocols for multiprocessors should fulfill the following requirements:

R1. The protocol should allow a task to request to hold a resource and then allow the task to release the resource;

R2. At every instant, a resource should be held by at most one task;

R3. The protocol should ensure the absence of deadlocks;

R4. The protocol should take advantage of the parallel processing capability of multicores;

R5. The protocol should have low run-time overhead;

R6. The protocol should work together with global fixed-priority preemptive scheduling. The rationale for this requirement is that such scheduling strategies are supported by many existing operating systems [13];

R7. The protocol should have an associated schedulability test for global fixed-priority preemptive scheduling;

R8. For priority-assignment schemes that are known to offer good performance (deadline monotonic (DM) [10], RM-US(x) [1] and DM-US(x) [12]), the combination of the resource-sharing protocol and the schedulability test should have the property that a large number of task sets can be given pre-run-time guarantees for deadline satisfaction;

R9. Performance (as stated by R8) should be particularly good for the important special case that the duration for which tasks need shared resources is small; it has been shown that this special case applies to most applications in practice [6].

The scientific community has already developed resource-sharing protocols which fulfill these requirements for a uniprocessor system. For example, the priority inheritance protocol (PIP) and priority ceiling protocol (PCP) [18] and the stack resource policy (SRP) [2]. Resource-sharing protocols for multiprocessors have also been proposed for Pfair scheduling [4], partitioned scheduling [17], [16] and global Earliest Deadline First (EDF) scheduling [4]. There is a protocol (called FMLP [4]) which could be used in global fixed-priority scheduling but no schedulability test is available for such use. In fact, the state-of-art in global multiprocessor scheduling currently offers no resource-sharing protocol with schedulability test.

Therefore, in this paper, we propose a new resource-sharing protocol for global fixed-priority preemptive multiprocessor scheduling, called PARALLEL-PCP (P-PCP in short). We also propose two new schedulability tests; one for P-PCP and another for the previously known PIP. These new schedulability tests assume non-nested resource accesses; this assumption has been shown to be true in most practical applications [6]. The prior state-of-art in schedulability analysis of global fixed-priority preemptive scheduling without resource sharing is the response-time analysis by Bertogna and Cirinei [3]. We extend this analysis to incorporate the effects of resource sharing based on either P-PCP or PIP.

P-PCP generalizes PCP to allow lower-priority tasks to execute even when PCP would forbid such an execution.

This work was partially funded by the Portuguese Science and Technology Foundation (Fundação para a Ciência e a Tecnologia - FCT) and the European Commission through grant ArtistDesign ICT-NoE-214373.

For example when a higher-priority task has locked some shared resource, then PCP does not allow any lower-priority task to lock resources, whereas, as we will see, P-PCP can be configured to allow one or more lower-priority tasks to lock resources. In this way, unlike PCP, the new protocol P-PCP allows the parallel processing capability of multicore platforms to be used with greater efficiency.

A common problem in resource sharing in multiprocessor systems is that a job may get blocked multiple times by the same lower priority job. P-PCP and PIP both suffer from this as well but we show that for a certain class of priority-assignments (which we call *reasonable priority-assignment*, and which includes popular priority-assignments such as deadline monotonic (DM) [10], RM-US(x) [1] and DM-US(x) [12]), it holds that the number of times that a job can be blocked by a lower priority job is small. Consequently, the combination of the protocols discussed in this paper and their corresponding schedulability analysis fulfill requirements R1-R9.

The remainder of this paper is organized as follows. Section II presents the system model and gives a background. Section III presents PIP and a new schedulability analysis for it under global fixed-priority preemptive scheduling. It also discusses the concept of *reasonable* priority assignment. Using this PIP analysis as the basis, we then develop the schedulability analysis for P-PCP; this new protocol and its analysis are presented in Section IV. Finally, Section V gives conclusions.

II. System Model and Background

A. System model

Task model. We assume that jobs are generated by sporadic tasks [14], and are scheduled on a multiprocessor platform comprised of m identical processors. A real-time system with shared resources is specified using p shared resources $R_1, \ldots, R_p$, and n sporadic tasks $T = \{\tau_1, \ldots, \tau_n\}$. Each sporadic task τ_i $(1 \leq i \leq n)$ is characterized as (T_i, C_i, D_i), where T_i denotes the minimum inter-arrival time, C_i the worst-case execution time, and D_i the relative deadline. Each job of task τ_i requires C_i units of processing capacity within D_i time units from its release, and this processing capacity must be supplied sequentially, *i.e.*, the job cannot be scheduled on more than one processor at any given time instant. Further, any two successive jobs of this task must be released at least T_i time units apart. In this paper we consider constrained-deadline tasks, *i.e.*, $\forall i : D_i \leq T_i$.

Multiprocessor scheduling. We consider global scheduling, *i.e.*, a job is not assigned to any specific processor; instead jobs which have arrived but whose execution have not finished are stored in a system-wide ready queue shared between processors. In this paper, we focus on global fixed-priority preemptive scheduling. Every task has a *base-priority*. We assume that base priorities are unique and therefore we order tasks (with no loss of generality) such that for every pair of tasks (τ_i,τ_j) it holds that if task τ_i has higher base priority than task τ_j then $i < j$. We let the base priority of a task be a positive integer such that a low number signifies a high priority; *i.e.*, priority level 1 is the highest priority level and priority level n is the lowest priority level. Protocols presented in this paper can temporarily raise the priority of a job. Such an elevated priority level is denoted as the *effective-priority* of the job.

Shared resources. One can distinguish between a resource-sharing protocol and a data-sharing protocol. A resource-sharing protocol ensures that at every moment, at most one job holds a shared resource. A data-sharing protocol ensures that the requests to perform operations on shared-data objects (for example add an element to a linked list or remove an element from a linked list) behave according to the correctness condition linearizability [8]. A resource-sharing protocol satisfies linearizability so it is also a data-sharing protocol, but a data sharing protocol is not necessarily a resource sharing protocol; the protocol may not guarantee mutual exclusion. Therefore, this paper focuses on the more general concept of resource sharing (it can be used to share both resources and data objects).

We assume that jobs can issue requests for exclusive access to the shared resources $R_1, \ldots, R_p$. A request for resource R_k by a job J of task τ_i is said to be *granted* as soon as J *holds* the resource. Once J has executed for the amount of time it requires R_k, the request is said to be *complete* and the resource is said to be *released*. We assume that a task holding a resource may be preempted by the processor scheduler but the task still holds the resource until it explicitly releases the resource. We assume that a resource-sharing protocol cannot force a job to release a resource.

A job of task τ_i could request resource R_k on multiple occasions during its execution, and we denote as $N_{i,k}$ the maximum number of such requests by any single job of τ_i. Associated with these requests is the worst-case duration of time for which a job uses resource R_k. We denote by $C_{i,k}$ the maximum (worst-case) resource usage time among all requests for resource R_k by jobs of τ_i. Similarly, let $CT_{i,k}$ denote the maximum total resource usage time for resource R_k by any single job of τ_i (sum of resource usage times over all requests for resource R_k). Further, we denote by $\mathcal{RS}_i$ $(\subseteq \{R_1, \ldots, R_p\})$ the set of all resources accessed by jobs of τ_i, and denote by $\lceil R_k \rceil$ the largest base-priority among all tasks that access resource R_k.

We make no assumption on the order in which the resources are requested and we make no assumption on where in the execution of a job a request takes place. For example, two jobs of task τ_i may both use resource R_1 and with our model, it is permitted that the request for R_1 is performed in the beginning of the first job but in the second job, the job executes a little without requesting a resource and then requests access to R_1.

In this paper we assume that accesses to shared resources are *non-nested*, *i.e.*, a job does not request for a shared resource while holding another. Note that tasks in our system cannot be deadlocked as a consequence of this assumption. Nested resource accesses can be handled in our protocol however using group locks, just as in the state-of-art [4].

Job blocking. A job J of task τ_i is said to be *directly blocked* at time t on a request for resource R_k, if the three conditions below are true:

1) at time t, job J is one of the m highest effective-priority jobs with remaining execution time;
2) at time t, resource R_k is locked by a job having lower base-priority than J;
3) job J made a request for resource R_k and this request has not been granted until time t.

Note that the above definition of blocking does not include the case when R_k is locked by a job having higher base-priority than J. This is consistent with the notion of blocking (that of being associated with priority-inversion) in the standard literature on uniprocessor systems [18], [2]. A job is said to be *ready* for execution whenever it is not directly blocked and not requesting a resource locked by a higher-base-priority job.

B. Related work

For multiprocessor systems, there has been a growing interest in the area of resource sharing. Rajkumar *et al.* were the first to propose a resource-sharing protocol on multiprocessors [17], [15], [16]. Two variants of PCP were presented by them for systems that use partitioned[1] fixed-priority scheduling. Several protocols related to PCP have since been proposed for systems scheduled under partitioned dynamic-priority (EDF) scheduling. Chen and Tripati [5] proposed two extensions to the basic protocol, but these extensions were only valid for periodic[2] (and not sporadic) task systems. Further, executions that use resources shared across processors (global shared-resource executions) were assumed to be non-preemptable and nesting was not allowed between such global resources and executions that use resources only shared within a processor (local shared-resource executions). In later work, López *et al.* [11] presented an implementation of SRP for partitioned EDF. However, this study required that tasks sharing resources be assigned to the same processor. Gai *et al.* [7] also presented an implementation of SRP for partitioned EDF and compared it to PCP. They have implemented a first-in-first out (FIFO) queue based spin-lock for global shared-resource executions, which has the potential to waste processing time (tasks can busy-wait for other tasks accessing global shared-resources). Further, accesses to different global shared-resources are not allowed to be nested, and these executions are scheduled in a non-preemptive manner.

In resource-sharing under global scheduling algorithms, there have been a few studies in the past [6], [9], [4]. Under global EDF, Devi *et al.* [6] proposed a FIFO-queue based spin-lock implementation for non-nested resource accesses. They also modified the global EDF scheduler to enforce non-preemptive scheduling of executions that use shared resources. Holman and Anderson [9] have proposed techniques for implementing non-nested resource accesses under Pfair global scheduling. They allow FIFO-queue based access to locked resources and present different techniques for handling short and long shared-resource executions. Flexible Multiprocessor Locking Protocol (FMLP), proposed by Block *et al.* [4], can be used under partitioned EDF, global EDF, and Pfair scheduling. They handle short shared-resource executions using FIFO-queue based spin-locks, and long shared-resource executions using priority inheritance similar to PCP. Under partitioned scheduling, their approach requires global shared-resource executions to be non-preemptive. Further, nested resource accesses are required to have group locks (separately for short and long accesses), thus negating the benefits of nesting.

Common to all these previous studies is that none of them offer a resource-sharing protocol for global scheduling with an associated schedulability test.

III. PIP AND ITS ANALYSIS

In this section, we calculate an upper bound on the response time of a task in a system using global fixed-priority scheduling with resources shared under PIP [18]. Before presenting the analysis, we discuss PIP characteristics specific to multiprocessors.

A. PIP for multiprocessors

Under PIP, whenever a job J of task τ_i is directly blocked on a resource R_k, the effective-priority of the job that is holding resource R_k (say J' of task τ_j) is raised to i (priority inheritance). In addition to direct blocking, job J may also experience interference from other lower priority jobs under PIP. For instance, this can happen when the effective-priority of a lower priority job is raised above i because of priority inheritance.

On uniprocessors, PIP ensures that J only experiences direct blocking (from job J') and lower priority interference from carry-in jobs; jobs that are released before J's release time. This is because any lower-base-priority job that is released after J cannot execute until J finishes its processing requirements. In other words, only those lower-base-priority jobs that hold a resource when J is released, can potentially interfere with J's executions.

On multiprocessors, PIP can cause interference even from lower priority jobs that are released after J's release time.

[1] Under partitioned scheduling, the tasks are first assigned to processors and uniprocessor scheduling algorithms and analysis techniques are used on each processor.

[2] A periodic task is similar to a sporadic task, except that T_i now denotes the exact inter-arrival time instead of minimum.

This can be explained using the example shown in Figure 1. It comprises of seven tasks $\tau_1, \ldots, \tau_7$, three shared resources R_1, R_2 and R_3 and $m = 3$ processors. Tasks τ_2 and τ_7 share resource R_1, τ_3 and τ_6 share R_2 and τ_4 and τ_5 share R_3. Task τ_1 does not use any shared resource. Initially, when task τ_4 requests resource R_3 it gets blocked (time t_1 in the figure). Prior to t_1, jobs of task τ_6 and τ_7 had their effective-priorities increased to 3 and 2 respectively. Further, a new job of task τ_1 is also released at time t_1. Therefore in the interval $(t_1, t_2]$ task τ_4 cannot execute, because three tasks with higher effective priority execute on the three available processors. In this interval τ_4 experiences interference from jobs of tasks τ_6 and τ_7. Later on in the interval $(t_3, t_4]$, the job of τ_5 that is directly blocking τ_4 executes. However during this interval, a new job of task τ_7 is released and it locks resource R_1. Then at t_4 jobs of tasks τ_1,τ_2 and τ_3 arrive, and τ_2 blocks raising the effective priority of the new job of τ_7. Then in the interval $(t_4, t_5]$, task τ_4 again experiences interference from a new job of task τ_7.

B. Response time analysis under PIP

In this section, we calculate an upper bound on the response time of task τ_i scheduled using global fixed-priority scheduler with resource sharing under PIP. We let the response time of a job be the time from when the job arrives until its execution has finished. We let the response time of a task be the maximum response time that is possible for jobs released by this task under the assumption that all jobs from all tasks are released according to the specifications given in Section II-A. We let RT_i denote an upper bound on the response time of task τ_i.

Intuitively, RT_i depends on three parameters: 1) the execution time C_i of task τ_i, 2) the amount of time that a job of τ_i needs to wait before being granted resources it requests, and 3) the amount of execution by other jobs that have higher effective-priority than τ_i. Note that if there are less than m jobs at some time instant that execute at higher effective-priority than τ_i, then they do not delay the execution of τ_i; only when all the processors are busy with such higher priority executions can the execution of τ_i be delayed. Therefore, the maximum delay that jobs of higher effective-priority can cause is the sum of such executions divided by the number of processors m. Specifically, this maximum delay occurs when all the higher priority executions are packed from left-to-right on the m processors, followed by the executions of τ_i. RT_i can therefore be calculated as follows:

$$\begin{aligned} RT_i =& C_i + \text{time\_to\_wait\_for\_resources\_held\_by\_other\_tasks} \\ &+ \frac{\text{executions\_with\_effective\_priority\_higher\_than\_i}}{m} \end{aligned} \tag{1}$$

Elaborating on this equation gives us

$$\begin{aligned} RT_i =& C_i + DB_i + Whp_i^{(dsr)} \\ &+ \frac{Whp_i^{(osr)} + Whp_i^{(nsr)} + Wlp_i}{m}, \end{aligned} \tag{2}$$

where the terms are as described below.

- DB_i denotes an upper bound on the amount of direct blocking that τ_i experiences;
- $Whp_i^{(dsr)}$ (direct shared-resource) denotes an upper bound on the amount of execution that tasks with a higher base-priority than τ_i can perform, when holding a resource that is also used by τ_i (resources in $\mathcal{RS}_i$). $DB_i + Whp_i^{(dsr)}$ therefore is an upper bound on the amount of time that any job of τ_i needs to wait before being granted resources it requests.
- $Whp_i^{(osr)}$ (other shared-resource) denotes an upper bound on the amount of execution that tasks with a higher base-priority than τ_i can perform, when holding a resource not in $\mathcal{RS}_i$;
- $Whp_i^{(nsr)}$ (no shared-resource) denotes an upper bound on the amount of execution that tasks with a higher base-priority than τ_i can perform, when they do not hold any shared resources;
- Wlp_i denotes an upper bound on the amount of execution that tasks with a lower base-priority than τ_i can perform, when having an effective-priority greater than τ_i. $Whp_i^{(osr)} + Whp_i^{(nsr)} + Wlp_i$ therefore is an upper bound on the amount of executions with effective-priority higher than τ_i.

The variables above with 'W' denote 'workload'. We find it convenient however to rewrite Equation (2) into

$$\begin{aligned} RT_i =& C_i + DB_i + Ihp_i^{(dsr)} \\ &+ Ihp_i^{(osr)} + Ihp_i^{(nsr)} + Ilp_i \end{aligned} \tag{3}$$

with the obvious connection of the variables to the variables in Equation (2). These variables with 'I' denote 'interference'. In the remainder of this section, we derive equations for each of the aforementioned terms.

Workload upper bound for a task. To upper bound the total workload of a task τ_l in the interval RT_i, we use the dispatch and execution pattern shown in Figure 2. Under this pattern, which was first considered by Bertogna and Cirinei [3], the carry-in job is assumed to execute as-late-as possible and every successive job is assumed to execute as-soon-as possible within the interval. It is easy to see that this pattern maximizes the amount of execution from jobs of τ_l in the interval of interest. We generalize this worst-case execution pattern as shown in Figure 3, giving importance to a certain portion x of the entire execution C_l (for example, executions that use shared resources). In this pattern, we assume that the x units of relevant executions happen as-late-as possible for the carry-in job and as-soon-as possible for every successive job within the interval. It is easy to see

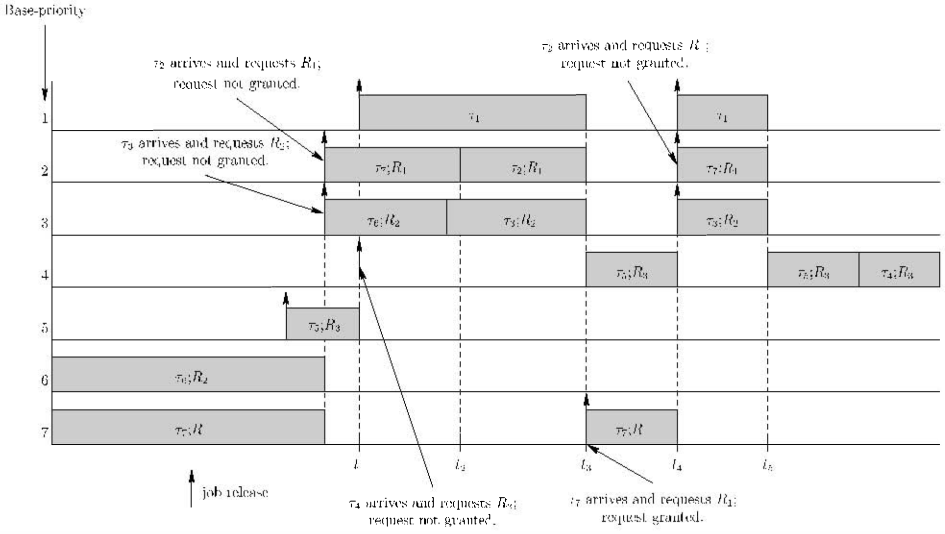
Base-priority
1
2
3
4
5
6
7
τ_2 arrives and requests R_1; request not granted.
τ_3 arrives and requests R_2; request not granted.
τ_2 arrives and requests R; request not granted.
τ_1
$\tau_7;R_1$
$\tau_2;R_1$
$\tau_6;R_2$
$\tau_3;R_2$
$\tau_5;R_3$
$\tau_4;R_3$
$\tau_6;R_2$
$\tau_7;R$
t
t_2
t_3
t_4
t_5
job release
τ_4 arrives and requests R_3; request not granted.
τ_7 arrives and requests R_1; request granted.

Figure 1. PIP for multiprocessors ($m = 3$)

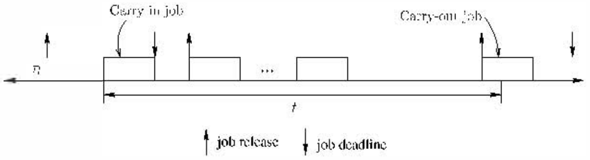

Figure 2. Worst-case execution pattern

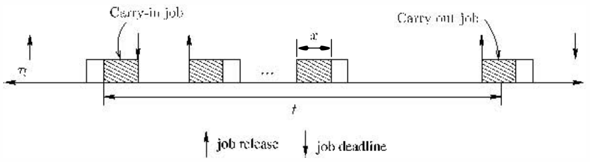

Figure 3. General worst-case execution pattern

that this pattern maximizes the amount of relevant executions of jobs of τ_l in the interval of interest[3].

We define the following notations for each task τ_l, based on the general worst-case execution pattern.

$$N_l(t,x) = \left\lfloor \frac{t - x + D_l}{T_l} \right\rfloor \tag{4}$$

$$\mathcal{W}_l(t,x) = xN_l(t,x) + \min\{x, t - x + D_l - T_lN_l(t,x)\} \tag{5}$$

$N_l(t,x)$ denotes the total number of jobs of task τ_l whose x relevant units completely execute within the interval of length t (including the carry-in job). $\mathcal{W}_l(t,x)$ gives the total workload of jobs of task τ_l, considering that x of the C_l execution units are relevant for this computation. We also define $\mathcal{W}_l(t,x) = 0$ for $t < 0$.

Waiting time before granting of resources. For every request of a shared resource $R_k \in \mathcal{RS}_i$, a job of task τ_i may be directly blocked by some lower base-priority job J, *i.e.*, in the worst-case J locked R_k just before it was requested by the job of τ_i. Note that once the job of τ_i is blocked, J is executed with effective-priority i. Therefore we can set

$$DB_i = \sum_{k:R_k \in \mathcal{RS}_i} N_{i,k} \max_{\substack{(l>i) \\ \wedge (R_k \in \mathcal{RS}_l)}} \{C_{l,k}\} \tag{6}$$

Whenever a job of task τ_i requests a shared resource R_k, it may so happen that all the higher base-priority jobs that use R_k also request the resource at the same time. In this case, τ_i will have to wait for each of these resource requests to complete in sequence, even though other processors are available. Further, when R_k is being locked by these higher priority jobs, more such higher base-priority jobs could be released that also request resource R_k. All these requests must be satisfied before the job of τ_i can get access to R_k. Therefore we set

$$Ihp_i^{(dsr)} = \sum_{l<i} \mathcal{W}_l \left(RT_i, \sum_{k:R_k \in \mathcal{RS}_l \cap \mathcal{RS}_i} CT_{l,k} \right) \tag{7}$$

Higher effective-priority executions. The remaining interference of higher base-priority jobs, *i.e.*, executions with resources not shared with τ_i ($Ihp_i^{(osr)}$) and executions with no shared resources ($Ihp_i^{(nsr)}$), can be expressed as:

$$Ihp_i^{(osr)} = \frac{\sum_{l<i} \mathcal{W}_l \left(RT_i, \sum_{k:R_k \in \mathcal{RS}_l \setminus \mathcal{RS}_i} CT_{l,k} \right)}{m} \tag{8}$$

$$Ihp_i^{(nsr)} = \frac{\sum_{l<i} \mathcal{W}_l \left(RT_i, C_l - \sum_{k:R_k \in \mathcal{RS}_l} CT_{l,k} \right)}{m} \tag{9}$$

[3] This pattern is not pessimistic because the software code of task τ_l could be such that in one path (carry-in job) the x relevant executions are performed at the end, whereas in another path (other jobs) these executions are performed at the beginning.

When a job J of task τ_i is either blocked or executing on one of the m processors, it may so happen that a job with a base-priority lower than J executes on another processor and locks a resource. Later on, this lower priority job may have its effective-priority higher than that of J, thereby causing interference to J's execution. This can happen even when the resource locked by the lower priority job is not used by J. In the worst-case, it is then possible that all those shared-resource executions of lower priority jobs whose effective-priority can be higher than i, may interfere with J. This interference can therefore be upper bounded as follows:

$$Ilp_i = \frac{\sum_{l>i} W_l \left(RT_i, \sum_{x:R_x \in \mathcal{RS}_l \wedge \lceil R_x \rceil < i} CT_{l,x} \right)}{m} \tag{10}$$

Improvement for high base-priority tasks. Equation (3) can be further improved for tasks that have the m highest base-priorities. This is because such tasks do not suffer from interference due to execution of lower priority tasks (Ilp_i), no shared-resource execution of higher priority tasks ($Ihp_i^{(nsr)}$) and other shared-resource execution of higher priority tasks ($Ihp_i^{(osr)}$). These can be explained as follows. Consider any task τ_i, $1 \leq i \leq m$. For a job with base-priority lower than i to have effective-priority higher than i, some job with base-priority higher than i must block on a resource request. Therefore, at any time instant, there can be at most $i-1$ jobs whose effective-priorities are increased from lower than i to higher than i. Suppose there are k such jobs at some time instant. Then it must also hold that there are k jobs with base-priority higher than i that are blocked. In other words, the total number of ready jobs that have effective-priority higher than i (jobs competing with task τ_i) can never be greater than $i-1$ at any time instant. Since there are m ($> i-1$) available processors, none of these $i-1$ jobs can interfere with the jobs of τ_i, except when they are using shared resources that are also requested by jobs of τ_i. The following theorem is a direct consequence of the above discussions.

Theorem 1: Consider a constrained deadline sporadic task system $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ that share resources $R_1, \ldots, R_p$ under PIP. Let this system be scheduled on m processors using a global fixed-priority preemptive scheduler such that task τ_i has higher base-priority than task τ_j for all $i < j$. Then RT_i is given as:

$$RT_i = \begin{cases} C_i + DB_i + Ihp_i^{(dsr)} & i \leq m \\ C_i + DB_i + Ihp_i^{(dsr)} & \\ +Ihp_i^{(osr)} + Ihp_i^{(nsr)} + Ilp_i & \text{Otherwise} \end{cases} \tag{11}$$

where DB_i, $Ihp_i^{(dsr)}$, $Ihp_i^{(osr)}$, $Ihp_i^{(nsr)}$ and Ilp_i are given by Equations (6), (7), (8), (9) and (10) respectively.

C. *Reasonable priority assignment*

Consider the calculations in Equation (11) which describe the response time upper bound of a job of task τ_i. There is a term which describes lower priority interference (Ilp_i) and it states that a lower priority task τ_l may interfere with the job of τ_i multiple times. It may appear that a lower priority task delays τ_i as much as if its executions with any shared resource was done at a priority higher than τ_i. And it may appear that this would have a severe negative effect on the ability to meet deadlines. It turns out (as will be seen in this section) that this is not the case for a class of priority-assignments called *reasonable priority assignment.*

Definition 1: A priority-assignment to tasks is *reasonable* if $D_i \leq D_j$ for every pair (τ_i,τ_j) such that τ_i and τ_j are two of the n-m lowest base-priority tasks and τ_i has higher base-priority than τ_j.

The definition of *reasonable priority-assignment* is interesting because it encompasses the priority-assignment schemes DM [10], RM-US(x) [1] and DM-US(x) [12], which are all known to offer good performance among global fixed-priority schedulers.

Lemma 1: For a reasonable priority-assignment, we can calculate Equation (10) using:

$$Ilp_i = \frac{\sum_{l>i} \sum_{x:(R_x \in \mathcal{RS}_l) \wedge (\lceil R_x \rceil < i)} 2 \cdot CT_{l,x}}{m} \tag{12}$$

Proof: When we calculate RT_i, we are only interested in performing calculations for $RT_i \leq D_i$ because otherwise we cannot guarantee that deadlines are met. Therefore, in the expressions used for calculating Ilp_i we can assume $RT_i \leq D_i$. We also know that $D_i \leq D_j$ because the priority-assignment is reasonable, where j is the index of a task with lower base-priority than τ_i. Further, we also know that $D_j \leq T_j$ because of the assumption of constrained-deadline tasks. Combining all these observations gives us that in the calculation of Ilp_i, we can assume $RT_i \leq T_j$. Then, using the knowledge that $RT_i \leq T_j$ in the expression for $W_l(t, x)$ gives us the statement of the lemma. ■

Thus we have seen that the use of a reasonable priority-assignment is efficient because a job will only experience at most two interferences from a lower base-priority job for each shared resource.

IV. P-PCP AND ITS ANALYSIS

In this section we present the P-PCP resource-sharing protocol and calculate an upper bound on the response time of a task in a system that uses global fixed-priority scheduling and P-PCP.

A. P-PCP

α_i ($1 \leq i \leq n$) denotes a tuning parameter in P-PCP that offers a trade-off between the allowable lower-priority interference for jobs of τ_i and the amount of executions that can be scheduled in parallel. A lower value for α_i implies that fewer jobs with base-priority lower than i are simultaneously allowed to execute at effective-priority higher than i. Although this reduces the interference for jobs of τ_i, it also reduces the number of jobs that can be executed in parallel. Parallelism can be increased by using a higher value for α_i, which also increases the interference for jobs of τ_i.

Before presenting P-PCP, we introduce some notations below which are defined for each time instant.

- PTY_i: Effective-priority of currently active job of task τ_i. $PTY_i = i$ when τ_i does not hold a resource and it can be modified by the protocol otherwise, temporarily raising the priority of the job.
- $PPTY_i$: The pseudo effective-priority of currently active job of task τ_i. It is the largest effective-priority that the job can have when it is holding the resource that it has currently locked. If it has not locked any resource, then the pseudo effective-priority is the same as its base-priority. If the job has locked resource R_k, then $PPTY_i = \lceil R_k \rceil$.
- $POPUP_i$: The number of jobs with base-priority lower than i, but pseudo effective-priority higher than i.
- HPR_i: The number of jobs with base-priority higher than i that have currently locked some shared resource.
- $MINC_i$: A task index j such that among all jobs with base-priority lower than i and pseudo effective-priority higher than i[4], the job corresponding to this index (say holding resource R_k) has the smallest worst-case resource usage time ($C_{j,k}$).

The main idea of P-PCP can be explained as follows. **A job is allowed to lock a shared resource if the following invariant remains true: the total number of jobs with base-priority less than i and pseudo effective-priority greater than i is at most α_i ($POPUP_i \leq \alpha_i$) for all i.** Otherwise the job is suspended, even if the resource being requested is available. If the aforementioned invariant remains true, then it is guaranteed that jobs of task τ_i will not experience lower-priority interference from more than α_i jobs simultaneously. This reduced interference comes at a price however; that of suspension of jobs even when the requested resource is not locked. In order for the aforementioned invariant to hold in our protocol, **we require that the α_i's satisfy the following properties: each α_i is a positive integer and $\alpha_1 \geq \alpha_2 \geq \ldots \geq \alpha_n$.**

The global fixed-priority scheduling algorithm with P-PCP is given by Algorithm 1. This algorithm executes at every time that any of the following events occurs: (i) a job arrives, (ii) a job finishes execution, (iii) a job requests a resource and (iv) a job releases a resource. The algorithm prioritizes jobs based on their effective priorities at the current time, *i.e.*, the currently active job of task τ_i has higher priority than the currently active job of task τ_j iff $PTY_i < PTY_j$. If a job is not requesting any resource at the current time, then it is immediately scheduled (Lines 4 and 5). On the other hand, if it requests a resource that is currently locked, then PIP is employed (Line 9). If the resource being requested is not locked, the condition in Line 11 is used to make a decision. A job J' with lower base-priority than J has higher pseudo effective-priority, if it has locked a resource which can also be requested by a job that has higher base-priority than J. And therefore it is possible that J' contributes to $POPUP_j$ for some j, s.t. $1 \leq j < i$. Similarly, jobs with base-priority higher than i that have locked some shared resources, may also contribute to $POPUP_j$ for some j, s.t. $1 \leq j < i$. If there are already α_i such jobs ($HPR_i + POPUP_i \geq \alpha_i$), then granting this resource to job J may lead to a violation of the invariant $POPUP_j \leq \alpha_j$. This follows from the fact that J can also potentially contribute to $POPUP_j$. Therefore the request is granted in Line 12 only when there are at most $\alpha_i - 1$ ($< \alpha_j$) such jobs. Note that the condition in Line 11 is not necessary (but only sufficient) to guarantee the invariant. We however use it because this enables us to derive the schedulability test in Section IV-B. If J is denied access to the resource and suspended, then the effective-priority of the job corresponding to task index $MINC_i$ is increased to i (Line 15). This ensures that the number of jobs in $POPUP_i$ is reduced as quickly as possible[5], thereby reducing the suspension time of τ_i.

Algorithm 1 Global fixed-priority scheduling with P-PCP

```
Input: $\forall i : 1 \leq i \leq n$, $\alpha_i$, the tuning parameters.
 1: for each job $J$ in priority order (based on $PTY$) do
 2:     Let $i$ denote the index of the task to which $J$ belongs.
 3:     if there are unassigned processors then
 4:         if $J$ is not requesting any resource then
 5:             Assign a processor to $J$.
 6:         else
 7:             Let $R_k$ denote the resource that $J$ is requesting.
 8:             if $R_k$ is locked then
 9:                 $PTY_j \leftarrow i$, if $R_k$ is locked by $\tau_j$ and $j > i$.
10:             else
11:                 if $HPR_i + POPUP_i < \alpha_i$ then
12:                     Assign a processor to $J$.
13:                 else
14:                     if $POPUP_i > 0$ then
15:                         $PTY_j \leftarrow i$, where $j = MINC_i$.
16:                     end if
17:                 end if
18:             end if
19:         end if
20:     end if
21: end for
```

We now illustrate P-PCP using an example. It comprises of tasks $\tau_1, \ldots, \tau_8$, shared resources $R_1, \ldots, R_3$ and $m = 3$ processors. Tasks τ_3 and τ_8 request resource R_1, τ_4 and τ_7 request R_2, and τ_5, τ_6 and τ_2 request resource R_3. The schedule of this example under PIP is shown in Figure 4 and its schedule under P-PCP is shown in Figure 5. Under

[4] Each such job has locked a shared resource.

[5] At each time instant, $MINC_i$ points to a job which has the smallest resource usage time among all jobs in $POPUP_i$.

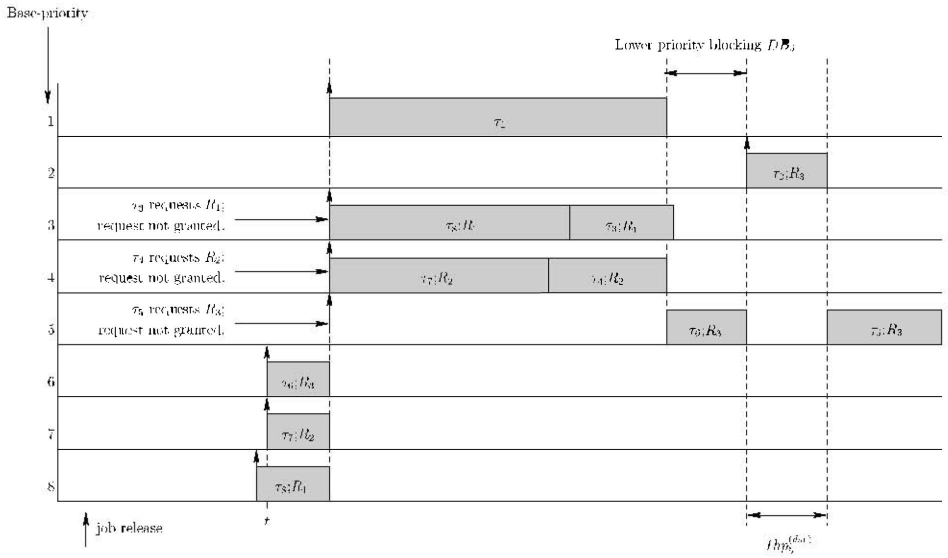

Figure 4. Example under PIP ($m = 3$)

P-PCP we have assumed that $\alpha_i = 3$ for all i s.t. $i \leq 5$ and $\alpha_i = 2$ otherwise. Under PIP, resource request of task τ_7 at time t is granted, which then leads to a large lower priority interference for task τ_5. In contrast, under P-PCP, the request is not granted and task τ_7 is suspended. As a result, the lower priority interference of task τ_5 reduces, thereby decreasing its response time. This example illustrates how α_i helps to reduce the lower priority interference for task τ_5, at the expense of introducing suspension delay for task τ_7. The following lemma proves the invariant preserved by P-PCP.

Lemma 2: At every time instant and for each task τ_i, $POPUP_i \leq \alpha_i$.

Proof: We prove this lemma by contradiction. Suppose there exists some time t and task τ_i for which the above condition does not hold. Then there exists some time instant t' ($\leq t$) such that: 1) the above condition was true an instant before t' (it was certainly true at time 0), 2) a request for resource R_k by a task τ_l with $l > i$ was granted at t' and as a result of this action $PPTY_l$ became smaller than i (if there are many such tasks then consider the one with the lowest base-priority among them), and 3) the above condition was false at an instant after t'. This follows from the fact that pseudo effective-priorities (and therefore $POPUP_i$) increase only when a resource request is granted. Just before task τ_l was granted resource R_k, there were at least α_i jobs that had base-priority lower than i but pseudo effective-priority greater than i. Suppose x of these α_i jobs have base-priority smaller than l and the remaining have base-priority between l and i. Then $HPR_l \geq \alpha_i - x$ and $POPUP_l \geq x$ by definition. Therefore $HPR_l + POPUP_l \geq \alpha_i \geq \alpha_l$. The last inequality follows from the fact that τ_i has higher base-priority than τ_l. Thus we have arrived at a contradiction because the job of task τ_l should have been suspended according to Line 11 of Algorithm 1. ■

B. *Response time analysis under* P-PCP

We now derive the response time upper bound for task τ_i scheduled using global fixed-priority scheduling with resources shared under P-PCP. Higher priority interference from no shared-resource execution ($Ihp_i^{(nsr)}$) is identical to that under PIP, because P-PCP behaves identical to PIP when dealing with execution that does not use any shared resource. Similarly, direct blocking ($\boldsymbol{DB}_i$) and higher priority interference from direct shared-resource execution ($Ihp_i^{(dsr)}$) are identical to that under PIP. This follows from the fact that even under P-PCP a job of task τ_i requesting resource R_k has to wait for at most one lower priority job (the one with largest resource usage time) and all higher priority jobs to release the resource.

For each request of a resource R_k by a job of task τ_i, the job may suspend when condition in Line 11 of Algorithm 1 is violated. If the job is suspended, then there are two contributing factors to consider: 1) suspension from lower base-priority jobs; $POPUP_i$ (considered here), and 2) suspension from higher base-priority jobs; HPR_i (considered later). In the worst-case, the job has to wait for every lower priority job from $POPUP_i$ to finish in sequence. Note that once the job of τ_i is suspended, no new lower priority job (say of task τ_l) can lock resources such that it increases $POPUP_i$. This is because $HPR_l + POPUP_l$ in this case would be at least as much as α_i ($\geq \alpha_l$). Since $POPUP_i \leq \alpha_i$ is always true, there are at most α_i such lower priority jobs. Therefore, the maximum suspension time from lower base-priority jobs is bounded by

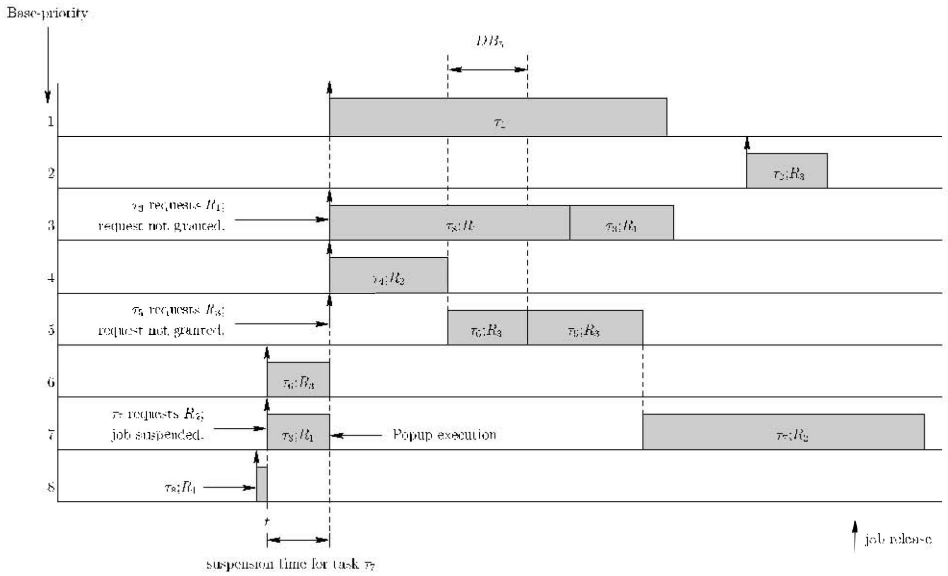

Figure 5. Example under P-PCP ($m = 3$)

$$sus_{i,k} = \text{Sum of } \alpha_i \text{ largest values in } \{C_{l,j} | (l > i) \wedge (R_j \neq R_k)\} \tag{13}$$

Then the total suspension time over all resource requests is bounded by

$$sus_i = \sum_{k:R_k \in \mathcal{RS}_i} N_{i,k} \cdot sus_{i,k} \tag{14}$$

To bound the suspension time from higher priority jobs, we consider the interference $Ihp_i^{(osr)}$. Note that only shared resource execution of higher priority jobs can contribute to suspension time of task τ_i (these are the jobs in HPR_i). Two observations can be made for this interference. 1) At each time instant that the job of τ_i is suspended, all the jobs in HPR_i are scheduled. This is because they all have priority higher than that of τ_i and they all hold shared resources. 2) If in any time interval the number of such higher priority jobs is smaller than α_i ($HPR_i < \alpha_i$), then the job of τ_i cannot be suspended in that interval unless $POPUP_i > 0$. In the latter case, sus_i defined above accounts for the suspension time. Therefore, suspension from higher priority jobs can be bounded as

$$Ihp_i^{(osr)} = \frac{\sum_{l:1 \le l < i} W_l \left(RT_i, \sum_{j:R_j \in \mathcal{RS}_l \setminus \mathcal{RS}_i} CT_{l,j} \right)}{\min\{m, \alpha_i\}} \tag{15}$$

To bound the lower priority interference Ilp_i, we use the same equation as for PIP which was shown in Equation (10). For reasonable priority-assignment schemes however, it is possible to do better when tasks have deadline equal to period. Note that, under reasonable priority assignment, the number of jobs from a task with priority lower than τ_i that can inherit a priority higher than τ_i is at most two (just like we showed in Section III-C). Further, since we use P-PCP, there can be at most α_i lower priority jobs that can simultaneously have priority higher than τ_i. Let us imagine the worst-case situation for PIP as characterized by Equation (10). Let us choose one of the tasks τ_l with priority lower than τ_i. We can move the arrival times of jobs of task τ_l to the right and observe how $W_l(\ldots)$ changes. Continue doing this as long as $W_l(\ldots)$ does not change. Once $W_l(\ldots)$ starts to change however, stop and observe this amount of shifting that has been done; call this value shift_l. Note that after this value, for ever time unit more of shifting, the amount of work done by τ_l is reduced by one. Compute the value shift_l for each task τ_l with base-priority lower than τ_i. Let A denote a tuple of tasks; these tasks all have lower base-priority than τ_i and are sorted in ascending order of shift_l. For the first α_i tasks in A, use $W_l(\ldots)$ from PIP. For the remaining tasks use $W_l(RT_i', \sum_{x:(R_x \in \mathcal{RS}_l) \wedge (\lceil R_x \rceil < i)} CT_{l,x})$ where $RT_i' = RT_i - \min_{l>i} \sum_{x:(R_x \in \mathcal{RS}_l) \wedge (\lceil R_x \rceil < i)} CT_{l,x}$. Thus lower priority interference can be bounded by

$$\begin{aligned} Ilp_i \quad = \quad & \frac{\sum_{(l>i) \wedge l \in A} W_l(RT_i, \sum_{x:(R_x \in \mathcal{RS}_l) \wedge (\lceil R_x \rceil < i)} CT_{l,x})}{m} \\ & + \frac{\sum_{(l>i) \wedge l \notin A} W_l(RT_i', \sum_{x:(R_x \in \mathcal{RS}_l) \wedge (\lceil R_x \rceil < i)} CT_{l,x})}{m} \end{aligned} \tag{16}$$

Whenever the total number of lower priority tasks is strictly greater than α_i ($n - i > \alpha_i$), Ilp_i has a tighter bound under P-PCP when compared to PIP, assuming task deadlines are equal to their respective periods and reasonable priority assignments are used. The following theorem is a direct consequence of above discussions.

Theorem 2: Consider a constrained deadline sporadic task system $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ that share resources $R_1, \ldots, R_p$ under P-PCP. Let this system be scheduled on m processors using a global fixed-priority preemptive scheduler such that task τ_i has higher base-priority than task τ_j for all $i < j$. Then RT_i is given as:

$$RT_i = C_i + DB_i + sus_i + Ihp_i^{(dsr)} + Ihp_i^{(osr)} + Ihp_i^{(nsr)} + Ilp_i \tag{17}$$

where DB_i, sus_i, $Ihp_i^{(dsr)}$, $Ihp_i^{(osr)}$, $Ihp_i^{(nsr)}$ are given by Equations (6), (14), (7), (15) and (9) respectively. And Ilp_i is given by Equation (16) if priority assignment is reasonable and tasks have deadline equal to period, and by Equation (10) otherwise.

Configuring P-PCP. When $\alpha_i = 1$ for all i, $1 \le i \le n$, P-PCP is identical to PCP. Similarly, when $\alpha_i = n$ for all i, $1 \le i \le n$, P-PCP is identical to PIP. It follows from the fact that in this special case, P-PCP does not induce any lower priority suspension ($sus_i = 0$) and the bounds on Ilp_i and $Ihp_i^{(osr)}$ under P-PCP are identical to those under PIP.

In general, it is always beneficial to set $\alpha_i = n$ for the m highest base-priority tasks ($i \le m$). This follows from the fact that $Ilp_i = Ihp_i^{(nsr)} = Ihp_i^{(osr)} = sus_i = 0$ for these tasks when $\alpha_i = n$. For other tasks we consider setting $\alpha_i = m$. In this case, the bound on $Ihp_i^{(osr)}$ under P-PCP is identical to that under PIP. The bound on Ilp_i under P-PCP, on the other hand, can be tighter in comparison to PIP. However, since P-PCP also has the additional suspension term sus_i, the protocols cannot be compared based on the given analysis.

V. Conclusions

We have presented a new resource-sharing protocol, P-PCP, and developed schedulability analysis for global fixed-priority preemptive multiprocessor scheduling under P-PCP as well as under PIP. These fulfill many of the requirements that application developers have; in particular the new protocol allows the parallel processing capability of multicores to be better used.

We left two important questions open:

Q1. How to tighten the upper bound on the lower-priority interference and the suspension term in P-PCP?

Q2. Is it possible to design an efficient resource-sharing protocol and corresponding schedulability analysis for nested resource requests?

References

[1] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. In *Proc. 22nd IEEE Real-Time Systems Symposium*, pages 193–202. Society Press, 2001.

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

[3] M. Bertogna and M. Cirinci. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Proc. 22nd IEEE Real-Time Systems Symposium*, pages 149–160. Society Press, 2007.

[4] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proc. of Real-time and Embedded Computing Systems and Applications Conference*, pages 47–56, 2007.

[5] C.-M. Chen and S. K. Tripathi. Multiprocessor priority ceiling based protocols. Technical report, 1994.

[6] U. C. Devi, H. Leontyev, and J. H. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *Proc. of Euromicro Conference on Real-Time Systems*, pages 75–84, 2006.

[7] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proc. of IEEE Real-Time Systems Symposium*, page 73, 2001.

[8] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[9] P. Holman and J. H. Anderson. Locking in pfair-scheduled multiprocessor systems. In *Proc. of IEEE Real-Time Systems Symposium*, page 149, 2002.

[10] J.Y.T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

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

[12] L. Lundberg and H. Lennerstad. Guaranteeing response times for aperiodic tasks in global multiprocessor scheduling. *Real-Time Syst.*, 35(2):135–151, 2007.

[13] L. Matassa and Y. Patil. Best practices: Adoption of symmetric multiprocessing using VxWorks and Intel multicore processors. Wind River whitepaper.

[14] A.K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment.* PhD thesis, PhD Thesis, Department of Computer Science, Massachusetts Institute of Technology (MIT), 1983.

[15] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *10th International Conference on Distributed Computing Systems*, pages 116–123, 1990.

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

[17] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proc. of IEEE Real-Time Systems Symposium*, pages 259–269, 1988.

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