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 multicore. 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 strate-

gies 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-runtime 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.
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 by a set of resources \( R_1, \ldots, R_p \), and for sporadic tasks \( T = \{ \tau_1, \ldots, \tau_n \} \).

Each sporadic task \( \tau_i \) (\( 1 \leq i \leq n \)) 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 \( \tau_i \) 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 assigned to a 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 \( (\tau_i, \tau_j) \) it holds that if task \( \tau_i \) has higher base priority than task \( \tau_j \) 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 \( \tau_i \) 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 \( \tau_i \) could request resource \( R_k \) on multiple occasions during its execution, and we denote as \( N_{ik} \) the maximum number of such requests by any single job of \( \tau_i \). Associated with these requests is the worst-case duration of time for which a job uses resource \( R_k \). We denote by \( C_{ik} \) the maximum (worst-case) resource usage time among all requests for resource \( R_k \) by jobs of \( \tau_i \). Similarly, let \( CT_{ik} \) denote the maximum total resource usage time for resource \( R_k \) by any single job of \( \tau_i \) (sum of resource usage times over all requests for resource \( R_k \)). Further, we denote by \( RS_k \) the set of all resources accessed by jobs of \( \tau_i \), and denote by \( |RS_k| \) 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 \( \tau_i \) 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 $R_k$.

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 $\tau_i$ 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 fixed-priority scheduling. Several protocols related to PCP have since been proposed for systems scheduled under partitioned dynamic-priority (EDF) scheduling. Chen and Tripudi [5] proposed two extensions to the basic protocol, but these extensions were only valid for periodic\(^1\) (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], [14]. 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 $\tau_i$ is directly blocked on a resource $R_k$, the effective-priority of the job that is holding resource $R_k$ (say $J'$ of task $\tau_j$) 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.
This can be explained using the example shown in Figure 1. It comprises of seven tasks \( \tau_1, \dots, \tau_7 \), three shared resources \( R_1, R_2 \) and \( R_3 \) and \( m = 3 \) processors. Tasks \( \tau_2 \) and \( \tau_7 \) share resource \( R_1 \), \( \tau_5 \) and \( \tau_6 \) share \( R_2 \) and \( \tau_4 \) and \( \tau_5 \) share \( R_3 \). Task \( \tau_1 \) does not use any shared resource. Initially, when task \( \tau_1 \) requests resource \( R_3 \) it gets blocked (time \( t_1 \) in the figure). Prior to \( t_1 \), jobs of task \( \tau_6 \) and \( \tau_7 \) had their effective-priorities increased to 3 and 2 respectively. Further, a new job of task \( \tau_1 \) is also released at time \( t_1 \). Therefore in the interval \((t_1, t_2]\) task \( \tau_1 \) cannot execute, because three tasks with higher effective priority execute on the three available processors. In this interval \( \tau_4 \) experiences interference from jobs of tasks \( \tau_6 \) and \( \tau_7 \). Later on in the interval \((t_3, t_4]\), the job of \( \tau_5 \) that is directly blocking \( \tau_4 \) executes. However, during this interval, a new job of task \( \tau_7 \) is released and it locks resource \( R_1 \). Then at \( t_4 \) jobs of tasks \( \tau_1, \tau_2 \) and \( \tau_3 \) arrive, and \( \tau_2 \) blocks raising the effective priority of the new job of \( \tau_7 \). Then in the interval \((t_4, t_5]\), task \( \tau_4 \) again experiences interference from a new job of task \( \tau_7 \).

\[ R_{T_i} = C_i + DB_i + W_{R_i}^{(d)} + \frac{W_{R_i}^{(o)}}{m} + W_{R_i}^{(s)} + W_{R_i}^{(n)} \]  

(2)

where the terms are as described below.

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

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

\[ R_{T_i} = C_i + DB_i + W_{R_i}^{(d)} + W_{R_i}^{(o)} + W_{R_i}^{(s)} + W_{R_i}^{(n)} \]  

(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 \( \tau_i \) 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 Cinieri [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 \( \tau_i \) in the interval of interest. We generalize this worst-case execution pattern as shown in Figure 3, giving importance to a certain portion \( \chi \) of the entire execution \( C_i \) (for example, executions that use shared resources). In this pattern, we assume that the \( \chi \) 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
that this pattern maximizes the amount of relevant executions of jobs of \( \tau_1 \) in the interval of interest\(^3\).

We define the following notations for each task \( \tau_i \), based on the general worst-case execution pattern.

\[
\begin{align*}
N_i(t, x) &= \left\lfloor \frac{t - x + D_i}{T_i} \right\rfloor \quad (4) \\
W_i(t, x) &= xN_i(t, x) + \min \left\{ x, t - x + D_i - T_iN_i(t, x) \right\} \quad (5)
\end{align*}
\]

\( N_i(t, x) \) denotes the total number of jobs of task \( \tau_i \) whose \( x \) relevant units completely execute within the interval of length \( t \) (including the carry-in job). \( W_i(t, x) \) gives the total workload of jobs of task \( \tau_i \) considering that \( x \) of the \( C_i \) execution units are relevant for this computation. We also define \( W_i(t, x) = 0 \) for \( t < 0 \).

\(^3\)This pattern is not pessimistic because the software code of task \( \tau_i \) 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.

---

**Figure 1.** PIP for multiprocessors \((m = 3)\)

**Figure 2.** Worst-case execution pattern

**Figure 3.** General worst-case execution pattern

**Waiting time before granting of resources.** For every request of a shared resource \( R_k \in RS_i \), a job of task \( \tau_i \) 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 \( \tau_i \). Note that once the job of \( \tau_i \) is blocked, \( J \) is executed with effective-priority \( i \). Therefore we can set

\[
DB_i = \sum_{k:R_k\in RS_i} N_{i,k} \max_{(C_{i,k} \in RS_i)} \left\{ C_{i,k} \right\} \quad (6)
\]

Whenever a job of task \( \tau_i \) requests a shared resource \( R_{ks} \), 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, \( \tau_i \) 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 looked 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 \( \tau_i \) can get access to \( R_k \). Therefore we set

\[
IH_{i}^{(disr)} = \sum_{l<i} W_l \left( RT_{i}, \sum_{k:R_k\in (RS_i \setminus RS_l)} CT_{i,k} \right) \quad (7)
\]

**Higher effective-priority executions.** The remaining interference of higher base-priority jobs, i.e., executions with resources not shared with \( \tau_i \) \((IH_{i}^{(cor)})\) and executions with no shared resources \((IH_{i}^{(nors)})\), can be expressed as:

\[
IH_{i}^{(cor)} = \sum_{l<i} W_l \left( RT_{i}, \sum_{k:R_k\in RS_i \setminus RS_l} CT_{i,k} \right) \quad (8)
\]

\[
IH_{i}^{(nors)} = \sum_{l<i} W_l \left( RT_{i}, C_l - \sum_{k:R_k\in RS_i} CT_{i,k} \right) \quad (9)
\]
When a job $J$ of task $\tau_i$ 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:

$$I_{p_i} = \sum_{t \geq i} W_t \left( \frac{RT_t \cdot \sum_{x \in \mathcal{R}_t \cap \mathcal{R}_i} \text{CT}_{t,x}}{m} \right)$$  \hspace{1cm} (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 ($I_{p_i}$), no shared-resource execution of higher priority tasks ($I_{p_i}^{\text{osr}}$) and other shared-resource execution of higher priority tasks ($I_{p_i}^{\text{osr}}$). These can be explained as follows. Consider any task $\tau_i$, $1 \leq i \leq m$. 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 $\tau_i$) 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 $\tau_i$, except when they are using shared resources that are also requested by jobs of $\tau_i$. The following theorem is a direct consequence of the above discussion.

**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 $\tau_i$ has higher base-priority than task $\tau_j$ for all $i < j$. Then $RT_i$ is given as:

$$RT_i = \begin{cases} C_t + DB_t + I_{p_i}^{\text{osr}} & i \leq m \\ C_t + DB_t + I_{p_i}^{\text{osr}} + I_{p_i}^{\text{osr}} & \text{Otherwise} \end{cases}$$  \hspace{1cm} (11)$$

where $DB_t$, $I_{p_i}^{\text{osr}}$, $I_{p_i}^{\text{osr}}$, $I_{p_i}^{\text{osr}}$ and $I_{p_i}$ 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 $\tau_i$. There is a term which describes lower priority interference ($I_{p_i}$) and it states that a lower priority task $\tau_j$ may interfere with the job of $\tau_i$ multiple times. It may appear that a lower priority task delays $\tau_i$ as much as its executions with any shared resource was done at a priority higher than $\tau_i$. 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 $(\tau_i, \tau_j)$ such that $\tau_i$ and $\tau_j$ are two of the $n \cdot m$ lowest base-priority tasks and $\tau_i$ has higher base-priority than $\tau_j$.

The definition of reasonable priority-assignment is interesting because it encompasses the priority-assignment schemes DM [10], RM-US(\infty) [11] and DM-US(\infty) [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:

$$I_{p_i} = \sum_{t \geq i} \sum_{x \in \{R_t \cap \mathcal{R}_i\}} \left( \frac{\text{CT}_{t,x}}{m} \right)^2$$  \hspace{1cm} (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 $I_{p_i}$ 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 $\tau_i$. 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 $I_{p_i}$, we can assume $RT_i \leq T_j$. Then, using the knowledge that $RT_i \leq T_j$ in the expression for $W_i(t,x)$ 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**

$\alpha_i$ ($1 \leq i \leq n$) denotes a tuning parameter in P-PCP that offers a trade-off between the allowable lower-priority interference for jobs of $\tau_i$ and the amount of executions that can be scheduled in parallel. A lower value for $\alpha_i$ 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 $\tau_i$, it also reduces the number of jobs that can be executed in parallel. Parallelism can be increased by using a higher value for $\alpha_i$, which also increases the interference for jobs of $\tau_i$.

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 $\tau_i$. $PTY_i = i$ when $\tau_i$ 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 $\tau_i$. 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 $PPTY_i = [R_k]$. 
- $POPUP_i$: 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 $i$, 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 $\alpha_i$ ($POPUP_i \leq \alpha_i$) 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 $\tau_i$ will not experience lower-priority interference from more than $\alpha_i$ 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 $\alpha_i$'s satisfy the following properties: each $\alpha_i$ 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 occur: (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 $\tau_i$ has higher priority than the currently active job of task $\tau_j$ iff $PTY_i < PTTY_j$. 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 $POPUP_i$ for some $j$, s.t. $1 \leq j < i$. If there are already $\alpha_i$ such jobs ($HPR_i + POPUP_i \geq \alpha_i$), then granting this resource to job $J$ may lead to a violation of the invariant $POPUP_i \leq \alpha_i$. 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 $\alpha_i - 1$ ($< \alpha_i$) 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, thereby reducing the suspension time of $\tau_i$.

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 $PTY_i$) 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:                 $PTY_{ij} = i$ 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:                        $PTY_{ij} = i$, 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 $\tau_1, \ldots, \tau_5$, shared resources $R_1, \ldots, R_3$ and $m = 3$ processors. Tasks $\tau_3$ and $\tau_5$ request resource $R_1$, $\tau_4$ and $\tau_7$ request resource $R_2$, and $\tau_5$, $\tau_6$ and $\tau_7$ 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

5At each time instant, $MINC_i$ points to a job which has the smallest resource usage time among all jobs in $POPUP_i$. 
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 $\tau_i$ at time $t$ is granted, which then leads to a large lower priority interference for task $\tau_5$. In contrast, under P-PCP, the request is not granted and task $\tau_i$ is suspended. As a result, the lower priority interference of task $\tau_5$ reduces, thereby decreasing its response time. This example illustrates how $\alpha_i$ helps to reduce the lower priority interference for task $\tau_5$ at the expense of introducing suspension delay for task $\tau_i$. The following lemma proves the invariant preserved by P-PCP.

**Lemma 2:** At every time instant and for each task $\tau_i$, $\text{POPUP}_i \leq \alpha_i$.

**Proof:** We prove this lemma by contradiction. Suppose there exists some time $t$ and task $\tau_i$ for which the above condition does not hold. Then there exists some time instant $t'$ such that: 1) the above condition was true at time $t'$, 2) a request for resource $R_k$ by a task $\tau_i$ with $l > i$ was granted at $t'$, and as a result of this action $PPTY_i$ 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 the above condition only when a resource request is granted. Just before task $\tau_i$ was granted resource $R_k$, there were at least $\alpha_i$ jobs that had base-priority lower than $i$ but pseudo effective-priority greater than $i$. Suppose $x$ of these $\alpha_i$ jobs have base-priority smaller than $l$ and the remaining have base-priority between $l$ and $i$. Then $HPR_i \geq \alpha_i - x$ and $\text{POPUP}_i \geq x$ by definition. Therefore $HPR_i + \text{POPUP}_i \geq \alpha_i \geq \alpha_i$. The last inequality follows from the fact that $\tau_i$ has higher base-priority than $\tau_i$. Thus we have arrived at a contradiction because the job of task $\tau_i$ 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 $\tau_i$ scheduled using global fixed-priority scheduling with resources shared under P-PCP. Higher priority interference from no shared-resource execution ($HPR_i^{\text{nosh}}$) is identical to that under PIB because P-PCP behaves identical to PIP when dealing with execution that does not use any shared resource. Similarly, direct blocking ($DB_i$) and higher priority interference from direct shared-resource execution ($HPR_i^{\text{sh}}$) are identical to that under PIB. This follows from the fact that even under P-PCP a job of task $\tau_i$ 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 $\tau_i$, 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, $\text{POPUP}_i$ (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 $\text{POPUP}_i$ to finish before. Note that once the job of $\tau_i$ is suspended, no new lower priority job (say of task $\tau_i$) can lock resources such that it increases $\text{POPUP}_i$. This is because $HPR_i + \text{POPUP}_i$ in this case would be as large as $\alpha_i \geq \alpha_i$. Since $\text{POPUP}_i \leq \alpha_i$ is always true, there are at most $\alpha_i$ such lower priority jobs. Therefore, the maximum suspension time from lower base-priority jobs is bounded by
\[ sus_{i,k} = \text{Sum of } \alpha_i \text{ largest values in } \{ C_{i,j} | (j > i) \land (R_j \neq R_i) \} \]  

(13)

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

\[ sus_i = \sum_{k : R_k \in R_i} N_{i,k} \cdot sus_{i,k} \]  

(14)

To bound the suspension time from higher priority jobs, we consider the interference \( Ip_i^{(csw)} \). Note that only shared resource execution of higher priority jobs can contribute to suspension time of task \( \tau_i \) (these are the jobs in \( HPR_i \)). Two observations can be made for this interference. 1) At each time instant that the job of \( \tau_i \) is suspended, all the jobs in \( HPR_i \) are scheduled. This is because they all have priority higher than that of \( \tau_i \) and they all hold shared resources. 2) If in any time interval the number of such higher priority jobs is smaller than \( \alpha_i \) (\( HPR_i < \alpha_i \)), then the job of \( \tau_i \) 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

\[ Ip_i^{(csw)} = \sum_{k : 1 \leq k \leq i} \sum_{R_k \in R_i \setminus R_S} W_i (RT_i, R_j, R_{y} \in R_i \setminus R_S, CT_{kj}) \frac{\alpha_i}{\min\{m, \alpha_i\}} \]  

(15)

To bound the lower priority interference \( Ip_i \), 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 \( \tau_i \) that can inherit a priority higher than \( \tau_i \) is at most two (just like we showed in Section III-C). Further, since we use P-PCP, there can be at most \( \alpha_i \) lower priority jobs that can simultaneously have priority higher than \( \tau_i \). Let us imagine the worst-case situation for PIP as characterized by Equation (10). Let us choose one of the tasks \( \tau_i \) with priority lower than \( \tau_i \). We can move the arrival times of jobs of task \( \tau_i \) to the right and observe how \( W_i(\ldots) \) changes. Continue doing this as long as \( W_i(\ldots) \) does not change. Once \( W_i(\ldots) \) starts to change however, stop and observe this amount of shifting that has been done; call this value \( shf_i \). Note that after this value, for ever time unit more of shifting, the amount of work done by \( \tau_i \) is reduced by one. Compute the value \( shf_i \) for each task \( \tau_i \) with base-priority lower than \( \tau_i \). Let \( A \) denote a tuple of tasks; these tasks all have lower base-priority than \( \tau_i \) and are sorted in ascending order of \( \alpha_i \). For the first \( \alpha_i \) tasks in \( A \), use \( W_i(\ldots) \) from PIP. For the remaining tasks use \( W_i(\sum_{i : R_x \in R_i \setminus [R_{y} \leq i]} CT_{iy}) \) where \( RT_i = RT_i - \min_{i > 1} \frac{\sum_{R_x \in R_i \setminus [R_{y} \leq i]} CT_{iy}}{m} \). Thus lower priority interference can be bounded by

\[ Ip_i = \sum_{(i > 1) \in A} \frac{W_i(\sum_{R_x \in R_i \setminus [R_{y} \leq i]} CT_{iy})}{m} + \sum_{(i > 1) \in A} \frac{W_i(\sum_{R_x \in R_i \setminus [R_{y} \leq i]} CT_{iy})}{m} \]  

(16)

Whenever the total number of lower priority tasks is strictly greater than \( \alpha_i \), \( (i - 1) > \alpha_i \), \( Ip_i \) 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 \( T = \{\tau_1, \ldots, \tau_n\} \) that share resources \( R_1, \ldots, R_n \) under P-PCP. Let this system be scheduled on \( m \) processors using a global fixed-priority preemptive scheduler such that task \( \tau_i \) has higher base-priority than task \( \tau_j \) for all \( i < j \). Then \( RT_i \) is given as:

\[
RT_i = C_i + DB_i + sus_i + IHP_i^{(dwar)} + IHP_i^{(osar)} + IHP_i^{(osw)} + IHP_i
\]

where \( DB_i, sus_i, IHP_i^{(dwar)}, IHP_i^{(osar)}, IHP_i^{(osw)} \) are given by Equations (6), (14), (7), (15) and (9) respectively, and \( IHP_i \) 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 \( i, 1 \leq i \leq n \), P-PCP is identical to PIP. Similarly, when \( \alpha_i = n \) for all \( i, 1 \leq i \leq n \), 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 \( IHP_i^{(osw)} \) 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 (\( i \leq m \)). This follows from the fact that \( IHP_i = IHP_i^{(osw)} = sus_i = 0 \) for these tasks when \( \alpha_i = n \). For other tasks we consider setting \( \alpha_i = m \). In this case, the bound on \( IHP_i^{(osw)} \) under P-PCP is identical to that under PIP. The bound on \( IHP_i \) 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 multicore systems 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


