International Workshop on Parallel and Distributed Real-Time Systems (IPDPS), Long Beach, California, Mar 2007.

Static-Priority Scheduling and Resource Hold Times*

Marko Bertogna Nathan Fisher Sanjoy Baruah

Abstract

The duration of time for which each application locks each shared resource is critically important in composing multiple independently-developed applications upon a shared "open" platform. In a companion paper [13], we formally defined and studied the concept of *resource hold time* (RHT) — the largest length of time that may elapse between the instant that an application system locks a resource and the instant that it subsequently releases the resource. [13] focused exclusively upon systems scheduled using EDF; in this paper, we extend the discussion and results from [13] to systems scheduled using static-priority scheduling algorithms. We present an algorithm for computing resource hold times for every resource in an application that is scheduled using static-priority scheduling, with resource access arbitrated using the Stack Resource Policy (SRP), or the Priority Ceiling Protocol (PCP). We also present an algorithm for decreasing these RHT's without changing the semantics of the application or compromising application feasibility.

Keywords: Resource-sharing systems; Sporadic tasks; Static-Priority Systems; Stack Resource Policy; Priority Ceiling Protocol; Resource holding times.

1 Introduction

In this paper, we study real-time systems that can be modelled as collections of *sporadic tasks* [23, 5], and are implemented upon a platform comprised of a single preemptive processor, and additional non-preemptable serially reusable resources. We assume that the shared resources are accessed within (possibly nested) critical sections which are guarded by semaphores, that the system is scheduled using a static-priority scheduling algorithm, such as the *deadline-monotonic* or *rate-monotonic* algorithm (e.g. see [1, 2, 17, 16, 15]), and that access to the shared resource is arbitrated by the Stack Resource Policy (SRP) [3] or the Priority Ceiling Protocol (PCP) [28].

Given the specifications of such a system, our objective is to *determine, for each non-preemptable serially reusable resource, the length of the longest interval of time for which the resource may be locked*. That is, we wish to determine, for each resource, the maximum amount of time that may elapse between the instant that the resource is locked, and the instant that it is next released.

Motivation and Significance. There has recently been much interest in the design and implementation of *open* environments [10] for real-time applications. Such open environments allow for multiple independently developed and validated real-time applications to co-execute upon a single shared platform.

While a large body of work has studied scheduling issues in such open/ hierarchical real-time systems, to our knowledge all of this work has focused primarily upon the scheduling of the (fully preemptive) processor. The few papers we could find [8, 9, 11] that do address the sharing of additional shared non-preemptable resources place severe restrictions on the individual applications – in effect requiring that the resource-requesting jobs comprising these applications be made available in a first-come first-serve manner to the higher-level scheduler (as would happen, e.g., if the applications were scheduled using table-driven scheduling).

The research reported here arose out of our ongoing efforts at building an open environment that offers support for sharing non-preemptable resources in addition to a preemptive processor. Clearly, a high-level scheduler that arbitrates access to such non-preemptable shared resources among different applications must have knowledge of how long each individual application may hold each resource. Hence we find it necessary to characterize the *application-wide* (as opposed to task- or job-specific) usage of these resources. Specifically, we wish to be able to determine separately, for each individual application and each shared resource, the maximum amount of time for which the application may hold the shared resource. This notion is encapsulated here in the concept of *resource holding times* (RHT's).

Contributions in this paper. Resource holding times (RHT's) quantify the largest amount of time for which an individual application may keep a resource locked. In a

*This research has been supported in part by the National Science Foundation (Grant Nos. CCR-0309825, CNS-0408996 and CCF-0541056).

companion paper [13], we formally defined and studied RHT's for EDF-scheduled applications using SRP. In this paper, we continue our study of RHT's by deriving an algorithm for computing such resource holding times from application system specifications for SRP under static-priority scheduling, and informally describe how this algorithm may be modified when PCP rather than SRP is used . We also present, and prove correct, an algorithm for modifying a given application to obtain a semantically equivalent application that will have decreased RHT's under SRP.

Organization. The remainder of this paper is organized as follows. In Section 2, we briefly describe our perspective on open environments in order to provide a context within which the contributions of this paper should be viewed. In Section 3, we present the formal model for resource-sharing sporadic task systems that is used in the remainder of this paper and summarize prior results on feasibility analysis of such resource-sharing sporadic task systems. Recall that our focus here is on resource holding times (RHT's); in Section 4, we describe how these RHT's may be computed from a given system's specifications. In Section 5, we present, and prove properties of, an algorithm for modifying a given resource-sharing sporadic task system in such a manner that its semantics do not change but its RHT's tend to decrease. Finally, in Section 6, we briefly describe why we are unable to reduce RHT's using this scheme for PCP and discuss an approach to calculating the RHT's for PCP.

2 Open environments

Most prior research on open environments has tended to focus on sharing only a preemptive processor among the applications. Hence, the proposed interfaces have been concerned with specifying the processor-specific aspects of the application, specifically, a model for its computational requirements, and a description of the scheduling algorithm used. Deng and Liu [10], Feng and Mok [22, 12], Saewong et al. [26], Shin and Lee [29], and Lipari and Bini [18] all model individual applications as collection of implicit-deadline periodic ("Liu and Layland") tasks [21]; some of these papers assume rate-monotonic local scheduling and the others assume EDF local scheduling. The Bandwidth Sharing Server [19, 20] has a more general workload model in that it specifies for each application the speed or computational capacity of a processor upon which the application is guaranteed to meet all deadlines, if executing in isolation, using EDF as the local scheduling algorithm.

Our perspective of an open system is somewhat more general, since we allow that non-preemptable serially reusable resources may be shared among the applications in addition to the preemptive processor. Hence we will model each individual application as a collection of sporadic tasks [23], each of which generates a potentially infinite sequence of jobs. Each such job may access the shared non-preemptive serially reusable resources within (possibly nested) critical sections guarded by semaphores. An open environment will validate that such an application meets all its timing constraints when implemented in isolation upon a dedicated "virtual" processor (VP) of a particular computing capacity, when scheduled using the preemptive static-priority scheduling algorithm, such as rate-monotonic (RM) [21] or deadline-monotonic (DM) [2], and with access to the shared resources being arbitrated using the Stack Resource Policy [3] (SRP) or the Priority Ceiling Protocol [28] (PCP). (Such validation, which is performed off-line prior to actual implementation, can be done using known techniques from real-time scheduling theory, such as those in [25, 15, 16].) If the system does indeed meet all its timing constraints, we will compute the resource hold times (RHT's) – the maximum length of time for which a resource is kept locked – for each shared resource. *The computation of these resource hold times is the subject of the current paper.* Provided with knowledge of the computing capacity of the VP, the resource holding times for all shared resources, and some additional information, an open environment can be designed to ensure that, if this application is admitted, then it will continue to meet all its timing constraints. A detailed and formal description of the design of such an open environment will be the subject of a future paper.

Let $\tau_1, \tau_2, \ldots, \tau_n$ denote the tasks, and $R_1, R_2, \ldots, R_m$ denote the additional shared resources. Tasks are assumed to be indexed according to non-increasing priority order (i.e. if $0 < i < j \leq n$, then τ_i has higher priority than τ_j).

1. Each resource R_j is statically assigned a *ceiling* $\Pi(R_j)$, which is set equal to the index of the lowest-indexed task that may access it:
$$\Pi(R_j) = \min\{i \mid \tau_i \text{ accesses } R_j\}$$
2. A *system ceiling* is computed each instant during run-time. This is set equal to the minimum ceiling of any resource that is currently being held by some job.
3. At any instant in time, a job generated by τ_i may begin execution only if it is the highest-priority job with remaining execution, and i is strictly less than the system ceiling. (It is shown [3] that a job that begins execution will not subsequently be blocked.)

Figure 1. Static-Priority Scheduling + SRP

3 System model and prior results

For the purposes of this paper, we assume that the application system, denoted by τ, can be modelled as a collection of n *sporadic* tasks [23, 5] $\tau_1, \tau_2, \ldots, \tau_n$. Each sporadic task τ_i ($1 \leq i \leq n$) is characterized by a worst-case execution time (WCET) C_i; a relative deadline parameter D_i; a

period/ minimum inter-arrival separation parameter T_i; and its resource requirements (discussed below). Each such task generates an infinite sequence of jobs, each with execution requirement at most C_i and deadline D_i time-units after its arrival, with the first job arriving at any time and subsequent successive arrivals separated by at least T_i time units. For the purpose of this paper, we assume that each task has $T_i \leq D_i$. Furthermore, we assume that *WCET parameters are normalized with respect to the speed of the dedicated processor*, i.e., each job of τ_i needs to execute for at most C_i time units upon the available dedicated processor.

We also assume that the task's are indexed in the decreasing priority order assigned by the given static-priority scheduling algorithm (i.e. if $0 < i < j \leq n$, then τ_i has higher priority than τ_j). For example, a static-priority algorithm such as DM assigns a priority to each task proportional to the inverse of task's relative deadline; in other words, for τ_i and τ_k, if $D_i > D_k$ then the priority τ_k is greater than τ_i.

The application is assumed to execute upon a platform comprised of a single dedicated preemptive processor, and m other non-preemptable serially reusable resources $R_1, R_2, \ldots, R_m$. The resource requirements of the sporadic tasks may be specified in many ways (see, e.g., [3, 19, 24]); for our purposes, we will let

(i) S_{ij} denote the length (in terms of WCET) of the largest critical section in τ_i that holds resource R_j; and

(ii) C_{ik} denote the length (in terms of WCET) of the largest critical section in τ_i that holds some resource that is also needed by τ_k's jobs ($i \neq k$).

We will indicate with S_j the largest critical section holding resource R_j among all task, i.e. $S_j = \max_{\tau_i \in \tau}\{S_{ij}\}$.

For any sporadic task τ_i and any non-negative number t, the **request bound function** $\text{RBF}(\tau_i, t)$ denotes the maximum cumulative execution requests that could be generated by jobs of τ_i that have both their arrival-times within a contiguous time-interval of length t. It has been shown (e.g. see [15]) for sporadic task τ_i that the request bound function is:

$$\text{RBF}(\tau_i, t) \stackrel{\text{def}}{=} \left\lceil \frac{t}{T_i} \right\rceil C_i \tag{1}$$

The cumulative execution request of all tasks with priority greater than τ_i and one job of τ_i over any interval of length t is given by:

$$W_i(t) \stackrel{\text{def}}{=} C_i + \sum_{j=1}^{i-1} \text{RBF}(\tau_j, t) \tag{2}$$

Static-Priority Scheduling + SRP. In the remainder of this paper, reasonable familiarity with the Stack Resource Policy (SRP) [3] is assumed. When it is used in conjunction with a static-priority scheduling algorithm, the rules used by the SRP to determine execution rights are summarized in Figure 1. Our focus in this paper is on SRP; therefore, due to space constraints, we will not present the entire rules for the Priority Ceiling Protocol (PCP) [28]. Section 6 will informally discuss how the techniques of this paper relate to PCP.

Feasibility analysis

We now review the feasibility analysis of systems that are scheduled using static-priority scheduling and SRP. A *priority inversion* is said to occur during run-time if the highest-priority job that is active – awaiting execution – at that time cannot execute because some resource needed for its execution is held by some other job. These lower-priority jobs are said to be the *blocking* jobs, and they *block* the higher-priority job. The higher-priority job is said to be *blocked* during the time that it is pending but does not execute, while lower-priority jobs execute.

It has been shown [3] that no job can be blocked after it has started executing and any job is blocked for at most one (outermost) critical section of a lower-priority task. Therefore, the maximum length of time that τ_i may be blocked by a lower-priority job is:

$$B_i \stackrel{\text{def}}{=} \max\{C_{ki} | (\tau_k \in \tau) \wedge (k > i)\} \tag{3}$$

Combining the above blocking term with the feasibility analysis of Lehoczky et al. [15], we can obtain the following result.

Theorem 1 *A task system τ is schedulable with a particular static priority assignement and with resource access arbitrated by* SRP *if and only if $\forall \tau_i \in \tau$, there exists $t \in (0, T_i]$ such that*

$$W_i(t) + B_i \leq t \tag{4}$$

The above inequality does not need to be evaluated at every $t \in (0, T_i]$. Bini et al. [6] show that to determine whether τ_i is schedulable according to the static-priority assignment and SRP, it is sufficient to evaluate Inequality (4) at the following set of points:

$$\mathcal{TS}(\tau_i) \stackrel{\text{def}}{=} \mathcal{P}_{i-1}(D_i) \tag{5}$$

where $\mathcal{P}_i(t)$ is defined by the following recurrent expression:

$$\begin{cases} \mathcal{P}_0(t) = \{t\} \\ \mathcal{P}_i(t) = \mathcal{P}_{i-1}\left(\left\lfloor \frac{t}{T_i} \right\rfloor T_i\right) \cup \mathcal{P}_{i-1}(t) \end{cases}$$

The above set $\mathcal{TS}(\tau_i)$ is referred to as the **testing set** for task τ_i. The size of $\mathcal{TS}(\tau_i)$ is *pseudo-polynomial* in

the parameters of the task system τ [6]. In the remainder of the paper we will assume that testing set $\mathcal{TS}(\tau_i)$ from Equation (5) is used, and that its elements $r_1, r_2, r_3, \ldots$ are indexed according to increasing value (i.e. $r_j < r_{j+1}$ for all j). It can be shown that the above schedulability analysis is identical for PCP [28]).

4 Computing the resource hold times

In this section, we describe how RHT's may be computed for each resource in a given feasible resource-sharing sporadic task system under static-priority scheduling and SRP. That is, we assume that the input task system has been deemed feasible (e.g., by the feasibility-testing algorithm described in the preceding section), and describe how we may compute RHT's for this system.

For any resource R_j and any task τ_i, let $\text{RHT}(R_j, \tau_i)$ denote the maximum length of time for which τ_i may keep resource R_j locked. Let $\text{RHT}(R_j)$ denote the system-wide resource holding time of R_j: $\text{RHT}(R_j) = \max_{i=1}^{n}\{\text{RHT}(R_j, \tau_i)\}$. Our objective is to compute $\text{RHT}(R_j)$ for each R_j.

Computing $\text{RHT}(R_j, \tau_i)$**.** We first describe how we would compute $\text{RHT}(R_j, \tau_i)$ for a given resource R_j and a given τ_i which uses R_j.

1. The first step is to identify the longest (in terms of WCET) critical section of τ_i accessing R_j. Recall that S_{ij} denotes the length of this longest critical section.

2. It follows from the definition of the static-priority scheduling and SRP protocol that no task with index $\geq \Pi(R_j)$ may execute while R_j is locked. Hence the only jobs that may execute while τ_i holds the lock on R_j are those generated by tasks $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$.

3. We now consider the number of times that a task with priority greater than $\Pi(R_j)$ may preempt τ_i while R_j is locked. Suppose τ_i holds R_j for at most t. While τ_i is holding the resource, it could be preempted by a task $\tau_\ell \in \{\tau_1, \ldots, \tau_{\Pi(R_j)}\}$ for at most:

$$\text{RBF}(\tau_\ell, t) \tag{6}$$

   Also, observe that if τ_i acquires R_j at time t_{ij}, then at time t_{ij} there are no active jobs of tasks $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ (otherwise, τ_i could not execute at time t_{ij}). So, the cumulative execution requests of jobs $\{\tau_1, \ldots, \tau_{\Pi(R_j)-1}\}$ that can preempt τ_i while it is holding resource R_j for t units of time, along with maximum amount τ_i can execute on resource R_j is given by:

$$W_i^{(j)}(t) \stackrel{\text{def}}{=} S_{ij} + \sum_{\ell=1}^{\Pi(R_j)-1} \text{RBF}(\tau_\ell, t). \tag{7}$$

4. Let t_i^* be the smallest fixed point of function $W_i^{(j)}(t)$ (i.e. $W_i^{(j)}(t_i^*) = t_i^*$). Using techniques from [14, 15], we can obtain t^* in time complexity that is pseudo-polynomial in the parameters of $\{\tau_1, \ldots, \tau_{\Pi(R_j)-1}\} \cup \{\tau_i\}$. t_i^* is the maximum amount of time τ_i can hold resource R_j. Thus,

$$\text{RHT}(R_j, \tau_i) \stackrel{\text{def}}{=} t_i^*. \tag{8}$$

It turns out that in order to find $\text{RHT}(R_j)$ we do not need to evaluate $\text{RHT}(R_j, \tau_i)$ for every τ_i that accesses resource R_j. Let $\tau_{\max}^{(j)} \stackrel{\text{def}}{=} \arg\max_{\tau_i \in \tau}\{S_{ij}\}$; that is, $\tau_{\max}^{(j)}$ is the task with the largest critical section (in terms of WCET) that holds resource R_j. The following theorem shows that we only need to calculate $\text{RHT}(R_j, \tau_{\max}^{(j)})$ to determine $\text{RHT}(R_j)$.

Theorem 2 $\text{RHT}(R_j)$ *equals* $\text{RHT}(R_j, \tau_{\max}^{(j)})$.

Proof: Rewriting Equation (7) for a tasks τ_k that access resource R_j, we have

$$W_k^{(j)}(t) = S_{kj} + \sum_{\ell=1}^{\Pi(R_j)-1} \text{RBF}(\tau_\ell, t). \tag{9}$$

Since Equations (7) and (9) differ only by the S_{ij} and S_{kj} terms (i.e. the sum of the $\text{RBF}(\tau_\ell, t)$ terms in the equations is independent of tasks τ_i and τ_k), the smallest fixed point of Equation (7) exceeds or equals the smallest fixed point of Equation (9) *if and only if* $S_{ij} \geq S_{kj}$. Therefore, $S_{ij} \geq S_{kj}$ implies $\text{RHT}(R_j, \tau_i) \geq \text{RHT}(R_j, \tau_k)$. Since $\tau_{\max}^{(j)}$ has the largest critical section of length S_j holding R_j among all tasks in terms of WCET, it also has the largest total holding time of R_j. Therefore, $\text{RHT}(R_j, \tau_{\max}^{(j)}) \geq \text{RHT}(R_j, \tau_i), \forall \tau_i$ and the theorem follows. ■

5 Minimizing the resource hold times

In the previous section we computed the resource holding times for systems scheduled with static priority that use SRP to arbitrate the access to resources; in this section, we address the issue of *decreasing* these RHT's, adapting the result we obtained in [13] for EDF to the static-priority systems considered in this paper. Since our objective is to apply the results obtained here for implementing open real-time environments that allow for resource-sharing, reducing the

RHT's will be beneficial to decrease the "interference" between different applications executing upon a common platform. (In the extreme, if all the RHT's were equal to zero then the open environment could ignore resource-sharing entirely.)

The algorithm we present in this section maintains the schedulability properties of SRP (an example of a maintained property, is the optimality of DM+SRP under certain assumptions [7]), while reducing resource holding times if possible. That is, we attempt to modify a given resource-sharing sporadic task system such that its semantics do not change, but the resource holding times in the resulting system are smaller (or in any event, no larger) than in the original. Furthermore, the modified system is scheduled by the exact same scheduling protocol as the original system; i.e., *we are proposing no changes to the application semantics, nor to the scheduling algorithm deployed*.

In [27], a similar technique to the one presented in the next subsection is used to decrease the number of preemptions in a static priority system with no critical sections. We improve that technique using a more general task model, reducing the overall complexity of the algorithm and taking into consideration the access to exclusive resources.

5.1 Reducing RHT for a single resource

In this section, we derive an algorithm for modifying a given resource-sharing sporadic task system such that the resource holding time $\text{RHT}(R_j)$ is reduced, for a single resource R_j. In Section 5.2, we extend this algorithm to reduce RHT's for all the shared resources in the system. We will first informally motivate and explain the intuition behind our algorithm; a more formal treatment follows.

Suppose that task τ_i uses resource R_j. We saw (in Section 4 above) that τ_i's execution on R_j may only be interrupted by jobs of tasks with index strictly less than $\Pi(R_j)$. Hence the smaller this preemption ceiling $\Pi(R_j)$, the smaller the value of $\text{RHT}(R_j, \tau_i)$; in the extreme, $\Pi(R_j) = 1$ and $\text{RHT}(R_j, \tau_i) = S_{ij}$ (i.e., the critical section executes without preemption). Hence, our RHT minimization strategy aims to make the ceiling $\Pi(R_j)$ of each resource R_j as small as possible without rendering the system infeasible. Let us consider a particular resource R_j with $\Pi(R_j) > 1$ to illustrate our strategy (if $\Pi(R_j) = 1$, then R_j's preemption ceiling cannot be reduced any further). We follow the same approach as in [13]:

> We add a "dummy" critical section — one of zero WCET — that accesses R_j to the task $\tau_{\Pi(R_j)-1}$, and check the resulting system for feasibility. If the resulting system is infeasible, then we remove the critical section: we are unable to reduce $\text{RHT}(R_j, \tau_i)$.
>
> Observe that adding such a critical section effectively decreases the preemption ceiling of R_j by one. Hence, the $\text{RHT}(R_j, \tau_i)$ in the resulting system is no larger than in the original system.
>
> Observe also that adding such a critical section with zero WCET is a purely syntactic change; hence, this change has no semantic effect on the task system.

By repeatedly applying the above strategy until it cannot be applied any further, we will have reduced each resource's preemption ceiling to the smallest possible value, thereby reducing the RHT's as much as possible using this strategy.

Having provided an informal description above, we now proceed in a more formal fashion by providing the necessary technical details. Let us begin with some assumptions: Our algorithm assumes that the schedulability of the initial system has been verified. We will also assume that the initial schedulability test for each task τ_i evaluates Inequality 4 for each $r_k \in \mathcal{TS}(\tau_i)$ in increasing order. If Inequality 4 is not satisfied for r_k, the r_k is removed from $\mathcal{TS}(\tau_i)$; otherwise, if Inequality 4 is satisfied the schedulability test for τ_i terminates and the "reduced-size" $\mathcal{TS}(\tau_i)$ is returned. Therefore, r_1 (the first element) of each modified testing set $\mathcal{TS}(\tau_i)$ is guaranteed to satisfy Inequality 4.

The pseudo-code for reducing R_j's preemption ceiling to the minimum possible value is given in Figure 2, as Procedure $\text{MINCEILING}(R_j)$. We will now argue that Procedure $\text{MINCEILING}(R_j)$ is correct.

```
MINCEILING(R_j)
    ▷ Reduce R_j's preemption ceiling as much as possible.
    ▷ Let TS(·) denote the testing set for a task. (Note:
      test points that do not satisfy Inequality 4 in the
      initial schedulability test and previous executions
      of MINCEILING have been removed).
    ▷ Let r_1, r_2, r_3, ... denote its elements indexed for
      increasing value (i.e. r_ℓ < r_{ℓ+1}, ∀ℓ)
1   while (Π(R_j) > 1) do
2           for r_k ∈ TS(τ_{Π(R_j)-1}) do
                    ▷ Validate Condition (4) for (t = r_k)
3                   if (W_{Π(R_j)-1} + S_j ≤ r_k) then
4                           Π(R_j) ← Π(R_j) − 1;
5                           break (for loop);
                    ▷ Remove r_k from TS(τ_{Π(R_j-1)})
6                   else
                TS(τ_{Π(R_j)-1}) ← (TS(τ_{Π(R_j)-1}) − r_k);
            end for
    end while
7   return; ▷ Ceiling safely decreased.
```

Figure 2. Reducing the preemption ceilings for R_j.

We show that each iteration of the *while* loop (Line 1) maintains system schedulability: suppose that we are attempting to decrease the ceiling $\Pi(R_j)$ of a resource R_j from its current value $i+1$ to i. Observe this change would only affect the blocking term B_i for task τ_i: higher-priority tasks than τ_i cannot be blocked by a task accessing resource R_j, and tasks with priority lower than τ_i have already calculated the maximum blocking due to a task that accesses R_j. In fact, if we decrease $\Pi(R_j)$ to i, B_i would change as follows:

$$B_i \leftarrow \max(B_i, S_j) \tag{10}$$

To prove that Procedure MINCEILING(R_j) maintains schedulability, we must prove the following theorem:

Theorem 3 *During a single iteration of the while loop (Line 1) of Procedure* MINCEILING(R_j), $\Pi(R_j)$ *is reduced by one* if and only if τ_i *is schedulable with* B_i *updated according to Equation 10.*

Before we prove this theorem, we prove the following useful lemma concerning updates of the testing set $\mathcal{TS}(\tau_{\Pi(R_j)-1})$ by Procedure MINCEILING(R_j):

Lemma 1 *Prior to each iteration of the while loop (Line 1) of Procedure* MINCEILING(R_j)*, the minimum element of* $\mathcal{TS}(\tau_k)$ *satisfies Inequality 4 for all* $\tau_\ell \in \tau$.

Proof Sketch: The proof is by induction. Assume that the lemma holds prior iteration of the *while* loop. Now consider a loop where $\Pi(R_j)$ is currently $i+1$. Obviously, only τ_i can be affected by the update of Line 6. So, we only need to prove that Inequality 4 holds for the first test point of $\mathcal{TS}(\tau_i)$ after completion of the loop.

The *for* loop considers test points of $\mathcal{TS}(\tau_i)$ in increasing order. Let B_i^{old} be the blocking term for τ_i prior when $\Pi(R_j) = i+1$; B_i^{new} is the blocking term if $\Pi(R_j) = i$ (calculated by Equation 10). Consider the first test point r_1 in the current $\mathcal{TS}(\tau_i)$. By induction hypothesis, Inequality 4 is satisfied at r_1 with B_i^{old}. If $S_j \leq B_i^{old}$, then Inequality 4 will be trivially satisfied at r_k, and the lemma will be true.

So, we will assume that $S_j > B_i^{old}$ (i.e. S_j equals B_i^{new}). Again consider the first test point r_1. If the condition of Line 3 for r_1 is violated, then Inequality 4 is also false, and Line 6 is executed, removing the first test point. The removal of successive test points is repeated until the condition of Line 3 is satisfied. When the condition of Line 3 is finally satisfied, $\Pi(R_j)$ is updated to i, and Inequality 4 is satisfied with the new blocking term. Otherwise, iff there are no more elements, $\mathcal{TS}(\tau_i)$ is empty. In either case, the lemma continues to hold. ■

We are now prepared to prove Theorem 3:

Proof of Theorem 3 Again, observe that only the schedulability of $\tau_{\Pi(R_j)-1}$ is affected by the iteration of the *while* loop of MINCEILING(R_j). Let $\Pi(R_j)$ be equal to $i+1$ prior to the considered iteration.

We will prove the "only if" part of the theorem first. Assume that τ_i is not schedulable after the update of $\Pi(R_j)$ to i (and update of B_i according to Equation 10). By Theorem 1, there does not exist an element of $\mathcal{TS}(\tau_i)$ that satisfies Inequality 4 with the updated B_i. $\Pi(R_j)$ will not be reduced.

Next, we will prove the "if" portion of the theorem. Assume that τ_i is schedulable according to Theorem 1 if $\Pi(R_j)$ is updated to i. Again, let B_i^{old} and B_i^{new} denote the same values as Lemma 1. So, there exists a $t \in (0, D_i]$ such that Inequality 4 is satisfied with B_i^{new}. If $S_j \leq B_i^{old}$, then $B_i^{old} = B_i^{new}$. By Lemma 1, the smallest value of the current $\mathcal{TS}(\tau_i)$ satisfies Inequality 4; thus, the condition of Line 3 is satisfied, and $\Pi(R_j)$ is reduced to i. Now, assume that $S_j > B_i^{old}$ (i.e. B_i^{new} equals S_j). Since in this section we minimize only one resource ceiling, we know that prior to the first *for* iteration, $\mathcal{TS}(\tau_i)$ has only been reduced by the initial schedulability test. Then, using Lemma 1, we know that r_1 satisfies Inequality 4 with blocking term B_i^{old}, and all points removed from the *initial* testing set failed the same Inequality. Being $B_i^{new} > B_i^{old}$, we know that the removed points would also fail the test using B_i^{new} as blocking term. Therefore, we could safely check the condition of Line 3 starting from r_1, and MINCEILING will eventually update $\Pi(R_j)$. ■

5.2 Reducing RHT's for all resources

In this section, we describe how the algorithms of Section 5.1 may be used to decrease the resource hold times for all the resources in a resource-sharing sporadic task system. The next result gives an algorithm to minimize each resource ceiling in an "optimal" way. We will say that a minimizing strategy is *optimal* if each $\Pi(R_i)$ obtained with this strategy will be equal to the lowest possible ceiling that could safely be assigned to R_i in the original task set (i.e. when no other ceiling has been modified).

Theorem 4 *Calling* MINCEILING(R_i) *for every resource in order according to their* S_j *value (i.e. for every* R_j, R_k*, if* $S_j < S_k$ *then* R_j*'s ceiling is minimized before* R_k*'s) is an optimal minimizing strategy.*

Proof: We begin by ordering the resources according to their S_j value (i.e. for every R_j and R_k in $\{R_1, R_2, \ldots, R_m\}$, if $j < k$ then $S_j < S_k$). We have to prove that calling MINCEILING(R_i) starting from R_1 and systematically proceeding in the given order, every time there is an element in the *initial* testing set that satisfies Condition 4 when it is evaluated to check if a resource ceiling could be decreased by one, then there exists an element

REDUCEALL(τ)

▷ Order resources $R_1, \ldots, R_m$ for non-decreasing
▷ value of their S_j (i.e. if $j < k$ then $S_j < S_k, \forall(R_j, R_k)$)
1 **for** $j \leftarrow 1$ **to** m **do** MINCEILING(R_j)

Figure 3. Reducing Preemption ceilings for all resources.

in the corresponding *reduced* set satisfying the same condition.

To see this, note that when MINCEILING(R_i) is called to decrease $\Pi(R_i)$, the condition at Line 3 is evaluated for every point of the corresponding reduced testing set, which is equal to the initial set excluding the elements that failed a similar condition during a previous call to MINCEILING(R_k). Since the removed points failed Condition 4 with S_k, then they would also fail Condition 4 with a higher value S_j. ■

The pseudo-code of procedure REDUCEALL(τ) in Figure 3 makes repeated calls to MINCEILING(R_j) in the order given by Theorem 4.

5.3 Computational Complexity

As can be seen from the pseudo-code in Figure 2, to decrease the ceiling of a resource R_j by 1 to i, Condition (4) must potentially be re-evaluated for every element in the testing set $\mathcal{TS}(\tau_i)$. There are potentially pseudo-polynomially many such elements; hence, the computational complexity of reducing the preemption ceiling of a single resource by 1 is pseudo-polynomial in the representation of the task system. For a resource that had an initial ceiling equal to n, that can be reduced to 1, Condition (4) has to be tested for $(n-1)$ different tasks, which remains in pseudo-polynomial time. Observe that we also avoid re-checking points that do not satisfy Condition (4). So, it is easy to see that the worst-case number of test points evaluated to minimize the ceiling of all resources is the same of the initial schedulability test.

6 Calculating and Minimizing RHT's for PCP

In SRP, a task holding a resource R_j executes at the priority of $\Pi(R_j)$ for the duration of the critical section for R_j. PCP differs by increasing the priority of a task τ_k holding R_j only when a higher-priority task τ_i (with priority lower than the system ceiling) is blocked. In this case, τ_k "inherits" τ_i's priority. (The reader is referred to [28] for a detailed description of PCP). Thus, the resource-holding time of τ_i for R_j is maximized when τ_i never inherits any task's priority and may be preempted by all higher-priority tasks, i.e. when all tasks τ_k with $\Pi(R_j) \leq k < i$ don't try to enter any critical section. Without making any assumption on the location of critical sections inside the worst-case code of any task (see [4]), an upper-bound on the RHT(R_j, τ_i) for this scenario is the smallest solution t satisfying:

$$S_{ij} + \sum_{\ell=1}^{i-1} \text{RBF}(\tau_\ell, t) \leq t. \tag{11}$$

RHT$(R_j) \stackrel{\text{def}}{=} \max_{\tau_i \in \tau}\{\text{RHT}(R_j, \tau_i)\}$. Please note that Theorem 2 does not hold for PCP.

We would also like to emphasize that the technique discussed in Section 5 does not apply to PCP. Since in the worst-case scenario a task τ_i locking a resource R_j could be preempted also by tasks with priority lower than the ceiling of R_j, artificially lowering the ceiling will have no effect on RHT(R_j). Therefore, PCP may be inappropriate to use for systems that require small RHT's.

7 Conclusion

Open environments, which allow for multiple independently-developed and validated applications to co-execute concurrently upon a common platform, are currently a hot topic in real-time systems research. Thus far, however, much work has assumed that shared execution platforms are comprised of only a single preemptive processor.

We believe that the logical next step in open environment design and implementation is to extend such environments to allow for more general shared platforms, that may be comprised of non-preemptable serially-reusable resources in addition to a preemptive processor. We are currently working on designing such a "second-generation" open environment. We have discovered that these more general second-generation open environments require a characterization of the amount of time that individual applications may keep specific resources locked (thereby denying the other applications access to the locked resource). In [13], we presented a systematic and methodical study of this specific behavioral feature for dynamic-priority scheduling. In this paper, we continue our study of resource locking durations by analyzing the behavior of static-priority systems. Our contributions include the following

- With respect to application systems that can be modelled using the resource-sharing sporadic task model, we have abstracted out what seems to be the most critical aspect of such resource locking. We have formalized this abstraction into the concept of resource hold

times (RHT's).

- We have presented an algorithm for computing RHT's for resource-sharing sporadic task systems scheduled using static-priorities and SRP. We have also briefly described how to obtain RHT's for PCP.

- We have presented, and proved the optimality of, an algorithm for decreasing RHT's of resource-sharing sporadic task systems scheduled using the static-priority scheduling and SRP.

References

[1] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings. Fixed priority preemptive scheduling: An historical perspective. *Real-Time Systems*, 8:173–198, 1995.

[2] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, pages 127–132, Atlanta, May 1991.

[3] T. P. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems: The International Journal of Time-Critical Computing*, 3, 1991.

[4] S. Baruah and A. Burns. Sustainable scheduling analysis. In *Proceedings of the IEEE Real-time Systems Symposium*, Rio de Janeiro, December 2006. IEEE Computer Society Press.

[5] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 182–190, Orlando, Florida, 1990. IEEE Computer Society Press.

[6] E. Bini and G. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11):1462–1473, November 2004.

[7] K. Bletsas and N. Audsley. Optimal priority assignment in the presence of blocking. *Inf. Process. Lett.*, 99(3):83–86, 2006.

[8] M. Caccamo and L. Sha. Aperiodic servers with resource constraints. In *Proceedings of the IEEE Real-Time Systems Symposium*, London, UK, December 2001. IEEE Computer Society Press.

[9] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proceedings of the IEEE Real-time Systems Symposium*, Rio de Janeiro, 2006. IEEE Computer Society.

[10] Z. Deng and J. Liu. Scheduling real-time applications in an Open environment. In *Proceedings of the Eighteenth Real-Time Systems Symposium*, pages 308–319, San Francisco, CA, December 1997. IEEE Computer Society Press.

[11] X. Feng. *Design of Real-Time Virtual Resource Architecture for Large-Scale Embedded Systems*. PhD thesis, Department of Computer Science, The University of Texas at Austin, 2004.

[12] X. A. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 26–35. IEEE Computer Society, 2002.

[13] N. Fisher, M. Bertogna, and S. Baruah. Resource-locking durations in edf-scheduled systems. Manuscript under review, 2006.

[14] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, Oct. 1986.

[15] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the Real-Time Systems Symposium - 1989*, pages 166–171, Santa Monica, California, USA, Dec. 1989. IEEE Computer Society Press.

[16] J. P. Lehoczky. Fixed priority scheduling of periodic tasks with arbitrary deadlines. In *IEEE Real-Time Systems Symposium*, pages 201–209, Dec. 1990.

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

[18] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *Proceedings of the EuroMicro Conference on Real-time Systems*, pages 151–160, Porto, Portugal, 2003. IEEE Computer Society.

[19] G. Lipari and G. Buttazzo. Schedulability analysis of periodic and aperiodic tasks with resource constraints. *Journal Of Systems Architecture*, 46(4):327–338, 2000.

[20] G. Lipari, J. Carpenter, and S. Baruah. A framework for achieving inter-application isolation in multiprogrammed, hard real-time environments. In *Proceedings of the Real-Time Systems Symposium*, Orlando, FL, November 2000. IEEE Computer Society Press.

[21] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[22] A. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *7th IEEE Real-Time Technology and Applications Symposium (RTAS '01)*, pages 75–84. IEEE, May 2001.

[23] A. K. Mok. *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983. Available as Technical Report No. MIT/LCS/TR-297.

[24] R. Pellizzoni and G. Lipari. Feasibility analysis of real-time periodic tasks with offsets. *Real-Time Systems: The International Journal of Time-Critical Computing*, 30(1–2):105–128, May 2005.

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

[26] S. Saewong, R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of hierarchical fixed-priority scheduling. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 173–181, Vienna, Austria, June 2002. IEEE Computer Society Press.

[27] M. Saksena and Y. Wang. Scalable real-time system design using preemption thresholds. In *Proceedings of the IEEE Real-Time Systems Symposium*, Los Alamitos, CA, Nov. 2000. IEEE Computer Society.

[28] 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.

[29] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 2–13. IEEE Computer Society, 2003.