Resource-locking durations in EDF-scheduled systems*

Nathan Fisher Marko Bertogna 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. 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 — is formally defined and studied in this paper. An algorithm is presented for computing resource hold times for every resource in an application that is scheduled using Earliest Deadline First scheduling, with resource access arbitrated using the Stack Resource Policy. An algorithm is presented for decreasing these* RHT*'s without changing the semantics of the application or compromising application feasibility.*

Keywords: *Resource-sharing systems; Sporadic tasks; Critical sections; Earliest Deadline First; Stack Resource Policy; Resource holding times.*

1 Introduction

In this paper, we study real-time systems that can be modelled as collections of *sporadic tasks* [19, 3], 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 the (preemptive) Earliest Deadline First scheduling algorithm (EDF) [9, 17], and that access to the shared resource is arbitrated by the Stack Resource Policy (SRP) [1].

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.

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

Motivation and Significance. There has recently been much interest in the design and implementation of *open* environments [8] 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. If an application is validated to meet its timing constraints when executing in isolation, then an open environment that accepts (or *admits*, through a process of admission control) this application guarantees that it will continue to meet its timing constraints upon the shared platform. The open environment has a run-time scheduler which arbitrates access to the platform among the various applications; each application has its own local scheduler for deciding which of its competing jobs executes each time the application is selected for execution by the "higher level" scheduler. (Such open environments are hence also often called "hierarchical" real-time environments.)

While a large body of work (see, e.g., [21, 5, 18, 11, 4, 24, 10, 6, 7] – this list is by no means exhaustive) 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 [5, 7] that do address the sharing of additional shared non-preemptable resources place severe restrictions on the individual applications – in effect requiring that the 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).

As part of ongoing efforts at building an open environment that offers support for sharing non-preemptable resources in addition to a preemptive processor, we have found it necessary to extend current scheduling-theoretic analysis techniques. 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 captured in this paper in the concept of *resource holding times* (RHT's).

Contributions in this paper. First, we formally study the concept of resource holding times (RHT's), that quantify the largest amount of time for which an individual application may keep a resource locked. Next, we present an algorithm for computing such resource holding times from application system specifications. Finally, we derive an algorithm for modifying a given application to obtain a semantically equivalent application that will have decreased RHT's.

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. We illustrate both the computation, and the minimization, of RHT's via an example in Section 6.

2 Open environments

In a typical open environment, an *interface* is defined between each application and the open environment. The goal of this interface specification is to abstract out and encapsulate the salient features of the application's resource requirements. The open environment uses this information during *admission control*, to determine whether the application can be supported concurrently with other already admitted applications. If an application is admitted, the interface also represents its "contract" with the open environment, which may use this information to "police" the application's run-time behavior. As long as the application behaves as specified by its interface, it is guaranteed to meet its timing constraints; if it violates its interface, it may be penalized while other applications are isolated from the effects of this misbehavior.

As stated above, 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 [8], Feng and Mok [18, 11], Saewong et al. [22], Shin and Lee [24], and Lipari and Bini [14] all model individual applications as collection of implicit-deadline periodic ("Liu and Layland") tasks [17]; some of these papers assume rate-monotonic local scheduling and the others assume EDF local scheduling. The Bandwidth Sharing Server [15, 16] 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.

Since our larger goal is to implement open environments upon platforms comprised of other non-preemptable serially reusable shared resources in addition to the preemptive processor, the interface must necessarily contain more information. Such information includes the resource holding times (RHT) of the application for each shared resource: the maximum amount of time that may elapse between the instant that some job of the application executes a lock on the resource and the instant that it next releases this resource (under the assumption that the application is executing on a dedicated processor). This information is used by the open environment during admission control and run-time scheduling; from the open environment's perspective, it is irrelevant which specific job within the application is holding the resource or whether this job is currently executing or not when this application is chosen for execution by the open environment.

Context of this paper. Our perspective of an open system is as follows. We will model each individual application as a collection of sporadic tasks [19], 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 Earliest Deadline First scheduling algorithm [17, 9] (Algorithm EDF), and with access to the shared resources being arbitrated using the Stack Resource Policy [1] (SRP) resource-sharing protocol. (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 [3, 1, 2].) If the system does indeed meet all its timing constraints, we will compute the resource hold times (RHT's) – the maximum length

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-decreasing relative deadlines: $D_i \leq D_{i+1}$ for all i.

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 earliest-deadline active job, and i is strictly less than the system ceiling. (It is shown [1] that a job that begins execution will not subsequently be blocked.)

Figure 1. EDF + SRP

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.

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 [19, 3] $\tau_1, \tau_2, \ldots, \tau_n$. 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.

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., [1, 15, 20]); for our purposes, we will let

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

(i) 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 ($i \neq k$).

τ	Sporadic task system
$\tau_i, 1 \leq i \leq n$	The i'th sporadic task
C_i, D_i, T_i	τ_i's WCET, rel. deadline, and period
$R_j, 1 \leq j \leq m$	The j'th shared resource
$\Pi(R_j)$	R_j's preemption ceiling
S_{ij}	Max. size of CS in τ_i that locks R_j
C_{ik}	Max. size of CS in τ_i locking some resource also needed by τ_k
$B(\cdot)$	Blocking function
$\text{DBF}(\tau_i, t)$	demand bound function of τ_i
$\text{DBF}(\tau, t)$	$\sum_{\tau_i \in \tau} \text{DBF}(\tau_i, t)$
$\mathcal{TS}(\tau)$	The testing set for τ
$d_1, d_2, \ldots$	Elements of $\mathcal{TS}(\tau)$
$\text{RHT}(R_j, \tau_i)$	Max time R_j may be locked by τ_i
$\text{RHT}(R_j)$	$\max_{\tau_i \in \tau} \text{RHT}(R_j, \tau_i)$

Assumptions:

1. Tasks indexed by relative deadline: $D_i \leq D_{i+1}$ $(\forall i)$
2. Elements of $\mathcal{TS}(\tau)$ indexed by value: $d_k < d_{k+1}$ $(\forall k)$

Figure 2. Notation and assumptions.

Some assumptions: in the remainder of this paper, we assume that *the sporadic tasks are indexed in non-decreasing order of their relative deadline parameters*: $D_i \leq D_{i+1}$ $(\forall i)$. 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.

For any sporadic task τ_i and any non-negative number t, the **demand bound function** $\text{DBF}(\tau_i, t)$ denotes the maximum cumulative execution requirement that could be generated by jobs of τ_i that have both their arrival-times and deadlines within a contiguous time-interval of length t. Algorithms are described in [3] for efficiently computing $\text{DBF}(\tau_i, t)$ in constant time, for any $t \geq 0$.

Some more notation: for task system τ and any non-negative t, we let $\text{DBF}(\tau, t)$ denote the following sum

$$\text{DBF}(\tau, t) = \sum_{\tau_i \in \tau} \text{DBF}(\tau_i, t) \ .$$

EDF+SRP. In the remainder of this paper, reasonable familiarity with the Stack Resource Policy (SRP) [1] is assumed. When it is used in conjunction with EDF, the rules used by the SRP to determine execution rights are summarized in Figure 1.

For convenience, some of the notation introduced in this section (and the next) that will be used in future sections is collected in Figure 2.

Feasibility analysis

We now review some definitions and results from [2], concerning the feasibility analysis of systems that are scheduled using EDF+SRP.

A *priority inversion* is said to occur during run-time if the earliest-deadline 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 (later-deadline) jobs are said to be the *blocking* jobs, and they *block* the earliest-deadline job. The earliest-deadline job is said to be *blocked* during the time that it is pending but does not execute, while later-deadline jobs execute.

Definition 1 (from [2]) *For any $L \geq 0$, the* **blocking function** *$B(L)$ denotes the largest amount of time for which a job of some task with relative-deadline $\leq L$ may be blocked by a job of some task with relative deadline $> L$.* ■

Recall that C_{ik} denotes the length of the largest critical section in τ_i that holds some resource that is also needed by τ_k. Given these C_{ik}'s, we can easily compute the blocking function $B(L)$ as follows:

$$B(L) \stackrel{\text{def}}{=} \max\{C_{ik} \mid D_i > L \text{ and } D_k \leq L\}$$

It has been shown [2] that ensuring

$$\text{DBF}(\tau, L) + B(L) \leq L \quad (1)$$

for all values of $L \geq 0$ is sufficient for ensuring feasibility – intuitively (see [2] for a formal proof) the first term on the left hand side (LHS) denotes the maximum cumulative demand of all the tasks over an interval of length L, and the second term denotes the maximum possible additional overhead due to blocking.

Observe from the definition of the blocking function above that $B(L) = 0$ for all $L \geq D_n$. Hence for all $L \geq D_n$, checking Condition 1 reduces to checking that $\text{DBF}(\tau, L)$ is at most L. The blocking term plays no role in determining the truth or falsity of this.

Observe that since both terms in the LHS of Condition 1 may increase only for values of L satisfying $L \equiv (k \cdot T_i + D_i)$ for some i, $1 \leq i \leq n$, and some integer $k \geq 0$, the system is feasible if this condition holds at all such values of L. Let $d_1, d_2, d_3, \ldots$ denote all such L, indexed according to increasing value (i.e., with $d_k < d_{k+1}$ for all k).

An *upper bound* has been determined [3] such that, if Condition 1 is not violated for some d_k smaller than this upper bound, then Condition 1 will not be violated for any d_k at all. This bound is equal to the smaller of (i) the least common multiple (lcm) of $T_1, T_2, \ldots, T_n$, and (ii) the following expression

$$\max\left(D_{\max}, \frac{1}{1-U} \cdot \sum_{i=1}^{n} U_i \cdot \max\left(0, T_i - D_i\right)\right)$$

where $D_{\max} \stackrel{\text{def}}{=} \max_{i=1}^{n}\{D_i\}$; U_i denotes the utilization (the ratio C_i/T_i) of τ_i; and U denotes the system utilization: $U \stackrel{\text{def}}{=} U_1 + U_2 + \cdots + U_n$. This bound may in general be exponential in the parameters of τ; however, it is pseudo-polynomial if the system utilization is a priori bounded from above by a constant less than one.

A definition: we will use the term **testing set** of τ to refer to the set of d_k's that are no larger than this upper bound. We will denote this testing set by the notation $\mathcal{TS}(\tau)$.

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. 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 EDF+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 $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ that have their deadline $\leq$ the deadline of the job of τ_i that holds the lock.

3. We now consider the maximum number of times any task $\tau_\ell \in \{\tau_1, \ldots, \tau_{\Pi(R_j)-1}\}$ could preempt τ_i while it (i.e., τ_i) is holding resource R_j. Let J_{ik} be a job of task τ_i that arrives at time a_{ik}. If J_{ik} acquires R_j at time t_{ij} ($\geq a_{ik}$), then at time t_{ij} there are no active jobs with deadlines prior to or at $a_{ik} + D_i$ of tasks $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ (otherwise, τ_i could not execute at time t_{ij}). Thus, only jobs of $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ with arrival times and deadlines within the interval $[t_{ij}, a_{ik} + D_i]$ may delay the execution of τ_i while it holds resource R_j. For a sporadic task τ_ℓ, it has been shown [3] that the maximum number of jobs of τ_ℓ that have both arrival times and deadline in any interval of

length t is $\left(\left\lfloor \frac{t-D_\ell}{T_\ell} \right\rfloor + 1\right)$. Since the earliest time at which job J_{ik} could acquire resource R_j is a_{ik}, an upper bound on the number of times task τ_ℓ can preempt τ_j while τ_j is holding resource R_j is:

$$\left(\left\lfloor \frac{D_i - D_\ell}{T_\ell} \right\rfloor + 1\right). \tag{2}$$

The above upper bound is dependent on the relative deadline of task τ_i. We can obtain a different upper bound on the maximum number of preempting jobs of τ_ℓ that is instead dependent upon the time that has elapsed since τ_i acquired resource R_j at time t_{ij}. Let t be the amount of elapsed time since t_{ij}. The maximum number of jobs of τ_ℓ that could have arrived and preempted job J_{ik} in the interval $[t_{ij}, t_{ij} + t)$ is:

$$\left\lceil \frac{t}{T_\ell} \right\rceil. \tag{3}$$

Combining Equations 2 and 3, we may obtain an overall upper bound on the number of preemptions by task τ_ℓ:

$$\min\left(\left\lceil \frac{t}{T_\ell} \right\rceil, \left(\left\lfloor \frac{D_i - D_\ell}{T_\ell} \right\rfloor + 1\right)\right). \tag{4}$$

4. We now quantify the maximum execution requests of jobs of tasks $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ that have deadlines prior to the deadline of job J_{ik}. Assume that t time units have elapsed since job J_{ik} has acquired R_j at time t_{ij}. Then, by Equation 4, the maximum execution requests by jobs of τ_ℓ (where $\tau_\ell \in \{\tau_1, \ldots, \tau_{\Pi(R_j)-1}\}$) that can preempt J_{ik} is given by the following function RBF which stands for "request-bound function:"

$$\text{RBF}(\tau_\ell, \tau_i, t) \stackrel{\text{def}}{=} \min\left(\left\lceil \frac{t}{T_\ell} \right\rceil, \left(\left\lfloor \frac{D_i - D_\ell}{T_\ell} \right\rfloor + 1\right)\right) \cdot C_\ell. \tag{5}$$

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:

$$W_i(t) \stackrel{\text{def}}{=} S_{ij} + \sum_{\ell=1}^{\Pi(R_j)-1} \text{RBF}(\tau_\ell, \tau_i, t). \tag{6}$$

5. Let t_i^* be the smallest fixed point of function $W_i(t)$ (i.e. $W_i(t_i^*) = t_i^*$). Using techniques from [12, 13], we can obtain t^* in time complexity that is pseudo-polynomial in the parameters of $\{\tau_1, \ldots, \tau_{\Pi(R_j)-1}\} \cup \{\tau_i\}$. By the next theorem, t_i^* is the maximum amount of time τ_i can hold resource R_j. Thus,

$$\text{RHT}(R_j, \tau_i) \stackrel{\text{def}}{=} t_i^*. \tag{7}$$

Theorem 1 *The maximum resource-holding time* $\text{RHT}(R_j, \tau_i)$ *is equal to the smallest fixed point of* $W_i(t)$.

Proof Sketch: As mentioned above in the third point, there are no active jobs at time t_{ij} for tasks $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$. By arguments similar to the Critical Instant Theorem of [17], it can be shown that the maximum time for job J_{ik} to complete its critical section for R_j occurs when tasks that may preempt J_{ik} (i.e. tasks $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$) simultaneously release jobs an arbitrarily small time ϵ after J_{ik} has acquired R_j, and subsequent jobs are released as soon as legally possible. For such a release sequence, $\text{RBF}(\tau_\ell, \tau_i, t)$ is an upper bound for task $\tau_\ell \in \{\tau_1, \ldots, \tau_{\Pi(R_j)-1}\}$ on the maximum execution requirements of jobs of τ_ℓ that may preempt J_{ik} released in the interval $[t_{ij}, t_{ij} + t)$. Thus $W_i(t)$ is an upper bound on the execution requirement of S_{ij} and all jobs of $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ that arrive in the interval $[t_{ij}, t_{ij} + t)$ and could preempt J_{ik}.

J_{ik} will relinquish the resource R_j at the first time the processor has completed the execution of the critical section of length S_{ij} and the requests of all higher priority jobs. Let t_i^* be the smallest fixed point of $W_i(t)$. $W_i(t_i^*) = t_i^*$ implies that the processor must have completed execution of the critical section of length S_{ij} (and all preempting jobs of $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$) by time $t_{ij} + t_i^*$. Therefore, t_i^* is equal to $\text{RHT}(\tau_i, t)$. ■

5 Minimizing the resource hold times

Section 4 above was concerned with computing the resource hold times; in this section, we address the issue of *decreasing* these RHT's. Since our objective is to apply the results obtained here for implementing open real-time environments that allow for resource-sharing, such decreases are beneficial since they reduce 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.)

In general, there is an easily-identified tradeoff between system feasibility and resource-holding time minimization.

- At one extreme, one could execute each critical section non-preemptively (as proposed by Mok [19]) to obtain the minimum possible resource holding time $\max_i\{S_{ij}\}$ for each R_j. However, such non-preemptive execution of critical sections means that

any job may be blocked by any other job, forcing deadline misses that may have been avoided by other resource-sharing policies – indeed, avoiding such unnecessary blocking was the primary motivation for the development of resource-sharing policies such as SRP.

- At the other extreme lie SRP and similar policies. SRP is known to be *optimal* in the sense that if a task system cannot be scheduled by EDF+SRP to meet all deadlines, then it would miss deadlines under any other work-conserving policy as well [2, Lemma 1]. However, such policies were not designed to minimize the duration of time for which a resource is locked; as we will see, they may end up keeping resources locked for a greater duration than is needed to ensure feasibility.

The algorithm we present in this section maintains the optimality of EDF+SRP, 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 – EDF+SRP – as the original system; i.e., *we are proposing no changes to the application semantics, nor to the scheduling algorithm deployed*.

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 resource 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[1]. 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).

[1]We note that reducing preemption thresholds has been used by Saksena et al. [25, 23], for a different purpose, namely, as a means of decreasing preemption overhead in static-priority scheduling. Our work differs from theirs in several ways: we are concerned with EDF scheduling; we consider shared resources as well; and our task model is more general than theirs.

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 $\text{RHT}(R_j, \tau_i)$.

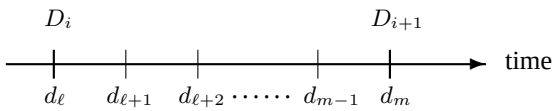
Observe that adding such a critical section effectively decreases the preemption ceiling of R_j by one. Hence, it may be hoped that $\text{RHT}(R_j, \tau_i)$ in the resulting system is smaller than in the original system; in any event, it is no larger than in the original system.

Observe also that adding such a critical section with zero WCET is a purely syntactic change. A "reasonable" implementation of the task system (which we assume here) would not have the task execute its lock for such a null-sized critical section; hence, this change has no semantic effect on the task system. Consequently, the modified task system is semantically identical to the original.

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. The pseudo-code for reducing R_j's preemption ceiling by one is given in Figure 3, as Procedure REDUCECEILING(R_j). Procedure REDUCECEILING(R_j) returns **false** if reducing $\Pi(R_j)$ by one would render the system infeasible; if reducing $\Pi(R_j)$ by one retains feasibility, then Procedure REDUCECEILING(R_j) modifies the task system accordingly (by adding a zero-WCET critical section using R_j to τ_i) and returns **true**.

We now explain why Procedure REDUCECEILING is correct. Let us suppose that $\Pi(R_j)$ is currently $(i+1)$, and we wish to explore whether we can reduce it to i without rendering the system infeasible. Clearly, both D_i and D_{i+1} are members of the testing set $\mathcal{TS}(\tau)$; recall that $d_1, d_2, \ldots$ denote the elements of $\mathcal{TS}(\tau)$ indexed in increasing order, and let $D_i \equiv d_\ell$ and $D_{i+1} \equiv d_m$:



Observe (from Definition 1 which defines the blocking function) that reducing $\Pi(R_j)$ to i from its current value of $(i+1)$ will have no effect on $B(L)$ for $L < D_i$, since no critical section of $\tau_i, \tau_{i+1}, \ldots \tau_n$ that could not already block some task in $\tau_1, \ldots, \tau_{i-1}$ is now able to do so. Similarly, this change will have no effect on $B(L)$ for $L \geq D_{i+1}$, since the preemption ceiling of R_j was already

REDUCECEILING(R_j)

▷ Reduce R_j's preemption ceiling by one, if possible
▷ Let $\Pi(R_j) \equiv (i+1)$, and let $d_\ell, d_{\ell+1}, \ldots, d_m$ denote the elements of the testing set $\mathcal{TS}(\tau)$ in the range
▷ $[D_i, D_{i+1}]$,indexed in increasing order. (I.e., $D_i = d_\ell$; $D_{i+1} = d_m$; and $d_k < d_{k+1}$ $(\forall k)$)
1 **for** $k = \ell$ **to** $(m-1)$ **do**
▷ Validate Condition 1 for $L \leftarrow d_k$
2 **if** $\left(\text{DBF}(\tau, d_k) + \max_{\ell=i+1}^{n}\{S_{\ell j}\}\right) > d_k$ **then**
3 **return false** ▷ Cannot lower $\Pi(R_j)$
end if
end for
▷ Decrease R_j's preemption ceiling to i
4 Add a zero-WCET critical section in τ_i, accessing R_j
5 **return true** ▷ Can lower $\Pi(R_j)$

MINCEILING(R_j)

▷ Reduce R_j's preemption ceiling as much as possible
1 **if** $(\Pi(R_j) > 1)$ **then**
2 **if** REDUCECEILING(R_j)) **then goto** 1
3 **return**

Figure 3. Reducing the preemption ceilings for R_j.

$(i+1)$ and hence critical sections holding R_j were already able to block tasks in $\tau_{i+1}, \ldots, \tau_n$.

In fact $B(L)$ may change only for L within the range $[D_i, D_{i+1})$, i.e., for each of $d_\ell, d_{\ell+1}, \ldots, d_{m-1}$. For each such d_k, $B(d_k)$ must modified as shown below[2]:

$$B(d_k) \leftarrow \max\left(B(d_k),\ \max_{\ell=i+1}^{n}\{S_{\ell j}\}\right). \qquad (8)$$

This is obtained based upon the following reasoning. Since we have added a "dummy" (zero-WCET) critical section using R_j to task τ_i, the value of $C_{\ell i}$ may change for all $\ell > i$ (recall that $C_{\ell i}$ denotes the length (in terms of WCET) of the largest critical section in τ_ℓ that holds some resource also needed by τ_i). Specifically, it is possible that the largest critical section using R_j in tasks $\tau_{i+1}, \tau_{i+2}, \ldots, \tau_n$ (i.e., $\max_{\ell=i+1}^{n}\{S_{\ell j}\}$) is larger than the current value of $B(d_k)$. In that case, the value of $B(d_k)$ must be updated, as shown above in Equation 8.

Hence to determine whether R_j's preemption ceiling can indeed be reduced from $(i+1)$ to i, we must re-validate the truth of Condition 1 with $B(L)$ computed according to Equation 8 above, when L takes on each of the values $d_\ell, d_{\ell+1}, \ldots, d_{m-2}, d_{m-1}$.

[2]Observe that the value of $B(d_k)$ is the same for all such d_k both prior to, and after, such modification. I.e., this updating need to be done only once overall, and not once for each d_k.

In fact since it is assumed that the task system is feasible when R_j's preemption ceiling $\Pi(R_j) = (i+1)$, we can simplify the evaluation of Condition 1, based on the following reasoning. If the new value of $B(d_k)$, as computed in Equation 8, were equal to the original value (i.e., the outer "max" in Equation 8 returns the first term), then the value of Condition 1 cannot have changed. Hence the re-evaluation only needs to be done if the the outer "max" equals the second term: (the $\max_{\ell=i+1}^{n}\{S_{\ell j}\}$). Alternatively, we could simply check Condition 1 with $B(d_k)$ set equal to $\max_{\ell=i+1}^{n}\{S_{\ell j}\}$, since the evaluation is guaranteed to return true if this term is not the max. This is the strategy employed in Procedure REDUCECEILING(R_j). The for-loop iterates through the values of d_k, while the conditional in Line 2 is the re-evaluation of Condition 1 if $\Pi(R_j)$ were to take on the value i. If Condition 1 does not hold for any of these values for L, then this reduction in $\Pi(R_j)$ will cause the system to become infeasible; hence Procedure REDUCECEILING(R_j) returns (line 3) without making the change.

On the other hand, if Condition 1 evaluates to true for all these d_k's, then $\Pi(R_j)$ may safely be decreased by one. This is done in line 4 of Procedure REDUCECEILING(R_j), by the artefact of adding a zero-WCET critical section accessing resource R_j to task τ_i.

By calling REDUCECEILING(R_j) repeatedly until it returns **false** or $\Pi(R_j) \equiv 1$, we can ensure that resource

R_j's ceiling is reduced to the smallest possible value; this is represented in pseudo-code form as Procedure MINCEILING(R_j) in Figure 3.

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.

Of course, the obvious way of doing this is to make individual calls to Procedure MINCEILING separately for each resource in the system. However, it is not clear what effect the *order* in which these calls are made has on the final preemption ceilings obtained for all the resources. For example, given a system with two resources R_1 and R_2, is the "better" strategy the one that calls MINCEILING(R_1) first and then MINCEILING(R_2), or the one that calls MINCEILING(R_2) first followed by MINCEILING(R_1)? (Or perhaps an even better strategy is to interleave calls to REDUCECEILING(R_1) and REDUCECEILING(R_2) in some specific sequence?) If the order in which calls are made to Procedure MINCEILING and Procedure REDUCECEILING make a difference to the amount by which the individual ceilings decrease, then we would need to study application semantics to decide what a "best" combination of RHT's would be, from the application's perspective, and then solve the question of deciding how to go about achieving an outcome close to this best combination.

Fortunately, it turns out that the order in which calls are made to Procedure REDUCECEILING (and hence, to Procedure MINCEILING) has no effect on the final preemption ceilings that are obtained; this is formalized in the following lemma.

Lemma 1 *For any pair of resources R_j and R_p ($p \neq j$), the truth value returned by* REDUCECEILING(R_j) *is not influenced by the number of times that* REDUCECEILING(R_p) *has already been called.*

Proof Sketch: Suppose that REDUCECEILING(R_p) has indeed been called one or more times prior to calling REDUCECEILING(R_j). Let the current value of $\Pi(R_j)$ be $(i+1)$.

Recall that we assume the system to initially be feasible, and observe that no call to REDUCECEILING changes this: a preemption ceiling is changed only if doing so does not render the system infeasible. Hence regardless of how many calls were made to to REDUCECEILING(R_p) prior to calling REDUCECEILING(R_j), we are guaranteed that the sporadic task system is feasible prior to the this call to REDUCECEILING(R_j).

Observe from the pseudo-code (Figure 3) the success or failure of the call to REDUCECEILING(R_j) is determined by the value of $\max_{\ell=i+1}^{n}\{S_{\ell j}\}$: the call fails if this exceeds $(d_k - \text{DBF}(\tau, d_k)$ for any d_k, and succeeds otherwise. Since the value of $\max_{\ell=i+1}^{n}\{S_{\ell j}\}$ is independent of prior calls to REDUCECEILING, it follows that the success or failure of this call is not dependent on whether REDUCECEILING(R_p) has been called previously or not. ■

```
REDUCEALL(τ)
    ▷ There are m resources, labelled R_1, ..., R_m
1   for j ← 1 to m MINCEILING(R_j)
```

Figure 4. Reducing Preemption ceilings for all resources.

As a consequence of Lemma 1, it follows that the order in which we choose to make calls to Procedure REDUCECEILING (and hence, to Procedure MINCEILING) has no influence upon the final values returned by calls to Procedure MINCEILING. We may therefore consider the resources in any order, and minimize the preemption ceiling of each before moving on to the next. This is formalized in the pseudo-code in Figure 4, which considers the resources in the order in which they are indexed.

5.3 Computational Complexity

As can be seen from the pseudo-code in Figure 3, Condition 1 must potentially be re-evaluated for every element in the testing set $\mathcal{TS}(\tau)$ between $D_{\Pi(R_j)}$ and $D_{\Pi(R_j)-1}$. For a resource R_j that had a ceiling initially equal to n which Procedure MINCEILING(R_j) reduces to 1 (by making $(n-1)$ calls to REDUCECEILING(R_j)), Condition 1 would need to be evaluated at each element of $\mathcal{TS}(\tau)$ that is smaller than D_n. There are potentially pseudo-polynomially many such elements; hence, the computational complexity of reducing the preemption ceiling of a single resource is pseudo-polynomial in the representation of the task system. The computational complexity of Procedure REDUCEALL(τ) (which reduces the preemption ceiling of all m resources in τ) is m times this, which remains in pseudo-polynomial time.

Since feasibility-analysis on the original system must be done anyway, there is an alternative implementation of REDUCEALL(τ) that, by requiring some additional "bookkeeping" operations during the pseudo-polynomial feasibility analysis, results in a computational complexity polynomial in the number of tasks. To explain this approach, let us define SLACK$_i(\tau)$ to equal the minimum value of $(d_k - \text{DBF}(\tau, d_k))$, for all $d_k \in \mathcal{TS}(\tau)$ that lie in the range $[D_i, D_{i+1})$. These SLACK$_i(\tau)$ values are easily computed during feasibility-analysis of τ, without increasing the computational complexity by more than a constant factor. It can

be shown that resource R_j's preemption ceiling may be reduced by one from $(i+1)$ to i if and only if

$$\max_{\ell=i+1}^{n} \{S_{\ell j}\} \leq \text{SLACK}_i(\tau) \ .$$

This is a polynomial-time test that leads to a polynomial-time implementation of REDUCECEILING(R_j), which can be used as the basis for polynomial-time implementations of both MINCEILING(R_j) and REDUCEALL(τ).

6 An illustrative example

In this section, we illustrate by means of a simple example task system how significant reductions in resource holding times may be obtained by using the algorithm derived in Section 5 above. The task system τ is comprised of the following four tasks:

τ_i	C_i	D_i	T_i
τ_1	1	3	3
τ_2	2	4	6
τ_3	1	6	6
τ_4	2	10	12

There is one additional shared resource R_1, which is only used by tasks τ_3 and τ_4, each for one time unit (i.e., $S_{31} = S_{41} = 1$). Hence, $\Pi(R_1) = 3$.

Validating feasibility. For this system, it may be verified the testing set of values is as follows:

$$\mathcal{TS}(\tau) = \{3, 4, 6, 9, 10, 12\}.$$

Also, $\text{DBF}(\tau, t) = \sum_{i=1}^{4} \text{DBF}(\tau_i, t)$ computed at each of these values of t yields the following

t	3	4	6	9	10	12
DBF(τ, t)	1	3	5	6	10	12

Computing the blocking function for this task system according to Definition 1 yields the following:

$$B(L) = \begin{cases} 1, & \text{for } L \in [6, 10) \\ 0, & \text{for all other } L \end{cases}$$

From the $B(L)$ values and the DBF(τ, t) values computed above, it may be validated that Condition 1 holds at all t in the testing set, and the system is therefore feasible as specified.

Computing RHT(R_1)**.** The algorithm of Section 4 may be used to compute RHT(R_1). The "worst case" scenario identified by that algorithm, that maximizes this resource holding time may be informally described as follows. Task τ_4's job locks the resource R_1 at time-instant 0 and jobs of the remaining three tasks arrive immediately after this lock is acquired with subsequent jobs of each task arriving at the maximum frequency. It may be verified that τ_1 executes over $[0, 1)$, and τ_2 over $[1, 3)$. At this time, the second job of τ_1 arrives, and executes over $[3, 4)$. Task τ_4 finally executes its critical section over $[4, 5)$, and releases the lock at time-instant 5 for a resource holding time RHT(R_1) of 5.

Decreasing RHT(R_1)**.** Using the technique of Section 5, we can however *reduce* the preemption ceiling of resource R_1 by one: $\Pi(R_1) \leftarrow 2$. In this case, the worst-case scenario remains the same as the one above but τ_2's job may not begin execution at time-instant 1, resulting in τ_4 executing its critical section over $[1, 2)$ and releasing the lock at time-instant 2 for a resource holding time RHT(R_1) of 2.

7 Conclusion

Over the past decade, much progress has been made in designing open environments that allow for multiple independently-developed and validated applications to co-execute concurrently upon a common platform. Thus far, much of this 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.

The first generation of open environments required a precise characterization of the processing requirements of each individual application. We believe that these more general second-generation open environments require, in addition, a characterization of the manner in which the individual applications make use of the other shared resources. One behavioral feature of application systems that seems particularly useful is the amount of time that individual applications may keep specific resources locked (thereby denying the other applications access to the locked resource).

In this paper, we have presented a systematic and methodical study of this specific behavioral feature. 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 the EDF+SRP framework. Although we have selected this particular scheduling framework since it relates most closely to the design of our planned open environment, we believe that our technique is general enough that it may be adapted, with minimal changes, to other scheduling frameworks (such as, for example, deadline-monotonic (DM) schedulingwith resource arbitration done according to the priority ceiling protocol (PCP)).

- We have presented, and proved the optimality of, an algorithm for decreasing RHT's of resource-sharing sporadic task systems scheduled using the EDF+SRP framework. Again, we believe that our technique is easily adapted to apply to systems scheduled using other scheduling frameworks (such as DM+PCP).

In a companion paper, currently under preparation, we will describe in detail how the RHT's computed here play a critical role in a general-purpose second-generation open environment that provides full support for shared non-preemptable serially-reusable resources.

References

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

[2] S. Baruah. Resource sharing in EDF-scheduled systems: A closer look. In *Proceedings of the IEEE Real-time Systems Symposium*, Rio de Janeiro, December 2006. IEEE Computer Society Press.

[3] 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.

[4] G. Bernat and A. Burns. Multiple servers and capacity sharing for implementing flexible scheduling. *Real- Time Systems*, 22:49–75, 2002.

[5] 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.

[6] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proceedings of the IEEE Real-time Systems Symposium*, pages 389–398, Miami, Florida, 2005. IEEE Computer Society.

[7] 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.

[8] 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.

[9] M. Dertouzos. Control robotics : the procedural control of physical processors. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: a resource-centric approach to real-time and multimedia systems. pages 476–490, 2001.

[22] 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.

[23] 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.

[24] 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.

[25] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proceedings of the International Conference on Real-time Computing Systems and Applications*. IEEE Computer Society, 1999.