Real-Time Syst
DOI 10.1007/s11241-008-9057-2

Resource holding times: computation and optimization

Marko Bertogna · Nathan Fisher · Sanjoy Baruah

Abstract In scheduling hard-real-time systems, the primary objective is to meet all deadlines. We study the scheduling of such systems with the secondary objective of minimizing the duration of time for which the system locks each shared resource. We abstract out this objective into the *resource hold time* (RHT)—the largest length of time that may elapse between the instant that a system locks a resource and the instant that it subsequently releases the resource, and study properties of the RHT. We present an algorithm for computing resource hold times for every resource in a task system that is scheduled using Earliest Deadline First scheduling, with resource access arbitrated using the Stack Resource Policy. We also present and prove the correctness of algorithms 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

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

M. Bertogna
ReTiS Lab CEIIC—Scuola Sant'Anna, via Moruzzi 1, 56124 Pisa, Italy
e-mail: marko@ssup.it

N. Fisher
Department of Computer Science, Wayne State University, 431 State Hall, 5143 Cass Avenue, Detroit, MI 48202, USA
e-mail: fishern@cs.wayne.edu

S. Baruah (✉)
Department of Computer Science, The University of North Carolina, Campus Box 3175, Sitterson Hall, Chapel Hill, NC 27599-3175, USA
e-mail: baruah@cs.unc.edu

1 Introduction

Real-time systems were initially modeled as collections of relatively simple independent recurring tasks executing on a simple shared preemptive platform. Given the specifications of such a system, the objective was to ensure that all timing constraints (usually, deadlines) were satisfied. Over time, this simple perspective has been generalized in several directions—for instance, more complex models for recurring tasks have been introduced, and different tasks are able to interact with each other through the sharing of (logical as well as physical) resources that allow exclusive access to only one task at any point in time.

Even with all these generalizations, the primary objective in real-time systems design and analysis has remained the same—ensuring that no deadlines are missed. A number of scheduling algorithms (such as earliest deadline first (EDF), rate-monotonic (RM) and deadline monotonic (DM), etc.) and resource sharing protocols (including the Priority Ceiling Protocol (PCP) and the Stack Resource Policy (SRP)) have been proposed and studied, for scheduling such general real-time systems upon preemptive uniprocessor platforms. It has been shown that specific scheduling frameworks—combinations of scheduling algorithms and resource-sharing protocols—are *optimal* under different constraints, meaning that if a set of tasks with shared resources can be scheduled under a specified scheduling model (periodic, sporadic, etc.), than it can also be scheduled with the optimal framework under the same scheduling model. As an example, it has been proven (Baruah 2006) that EDF + SRP is an optimal scheduling strategy in the sporadic task model. Using EDF + SRP, it is therefore possible to schedule every feasible sporadic task set requiring access to shared resources.[1]

However, while meeting deadlines remains a primary objective in many real-time systems, more recent developments in real-time systems research have made certain secondary objectives very important as well. For instance, portable embedded real-time devices require that scheduling frameworks meet all deadlines while minimizing energy consumption. For high-volume consumer electronics, where small savings in per-unit cost translates into significant over-all savings, the goal may be to identify the least powerful (expensive) platform on which all deadlines may be met. Both these secondary objectives have been studied elsewhere; in this paper, we consider another such secondary objective: to *minimize the duration of time for which the task system holds on to shared resources*.

Motivation and significance This secondary objective is motivated by two very significant recent developments in real-time and embedded systems design: *open environments*, and *multicore* platforms.

[1]Each of the aforementioned resource-sharing protocols and the techniques presented in this paper assumes that each resource is a *serially-reusable non-preemptive shared resource*. A task accessing a serially-reusable non-preemptive shared resource relinquishes the resource only upon completion of execution on the resource. Note that this does *not* prohibit processor preemption of a task holding the shared resource; however, a task will not release the resource upon processor preemption. The term "serially-reusable non-preemptive resource" has been adopted from Baker (1991).

Open environments There has recently been much interest in the design and implementation of open environments (Deng and Liu 1997) for real-time applications. In this setting, a real-time application consists of real-time task system, local (application-specific) run-time scheduler, and a set of local shared resources. An open environment allows for multiple independently developed and validated real-time applications to co-execute upon a single shared platform. The open environment has a *global* run-time scheduler which arbitrates access to the platform among the various applications; each application will use its *local* scheduler for deciding which of its competing jobs executes each time the application is selected for execution by the global scheduler. If an application is validated to meet its timing constraints when executing in isolation, then an open environment that admits this application (through a process of admission control) guarantees that it will continue to meet its timing constraints upon the shared platform. Such open environments are hence also often called "hierarchical" real-time environments. An example of a hierarchical system is given by the two-level architecture proposed by Deng and Liu (1997), where a global EDF scheduler is used, with a Total Bandwidth Server (Spuri and Buttazzo 1996) for each application. Kuo and Li (1999) extended the model to fixed priority global schedulers, using a Sporadic Server (Sprunt et al. 1989) for each application. Both papers derive sufficient schedulability conditions. Saewong et al. (2002) present a response time analysis for Deferrable Servers and Sporadic Servers used in a hierarchical environment. Feng and Mok (2002) presented a general methodology for hierarchical partitioning of a computational resource at arbitrary levels of the hierarchy, deriving simple sufficient schedulability test for any scheduler at any level. Shin and Lee (2003) derived exact schedulability conditions for EDF and fixed priority scheduling at local level. Lipari and Bini (2003) addressed the problem of optimizing the server parameters given an application scheduled by a local fixed priority scheduler. Davis and Burns (2005) developed response time analysis for the schedulability of task systems scheduled under fixed priority servers (Periodic, Sporadic or Deferrable). However, strictly speaking, the hierarchical approach used by Davis and Burns (2005) cannot be completely classified as an open environment, since the temporal constraints of an application are not validated in isolation.

Sharing of "global" resources (i.e., those resources that are used by multiple applications co-executing in the open environment) presents a major challenge to the design and implementation of open environments. Clearly, a high-level scheduler that arbitrates access to shared resources that are accessed by multiple applications must have knowledge of how long each individual application may hold each global resource. Furthermore, in order to minimize interference between different admitted applications, it is desirable that each individual application minimizes the duration of time for which it holds each shared resource. Open environment frameworks that explicitly minimize and take into account each application's resource usage have been presented in Fisher et al. (2007a, 2007b) and Behnam et al. (2007); each of these aforementioned frameworks make use of techniques presented in this paper. Similar to the Hierarchical Stack Resource Protocol (HSRP) proposed by Davis and Burns in Davis and Burns (2006), the open environments of Fisher et al. (2007a, 2007b), and Behnam et al. (2007) use a high-level SRP used to arbitrate the access to resources shared among different applications. However, Fisher et al. (2007a, 2007b), and Behnam et al. (2007) differ from the solution adopted by Burns and Davis, where each critical section is executed with local preemptions disabled, by allowing local preemptions, when needed, even if a global resource is locked. This modification is particularly useful for applications with a task with short period and other tasks accessing a long global critical section: if the critical section is executed non-preemptively, the task with low period would miss its deadline. The drawback of this approach is that the time for which an application keeps a global resource locked increases. To avoid the problem of budget exhaustion inside critical sections, the open environment grants application locks for a global resource only when the capacity of the corresponding server is at least equal to the maximum amount of time the global resource could be held by that application. To optimally utilize the available processing capacity avoiding over-dimensioned servers, it is therefore very important to have low resource holding times. We will describe in this paper how to compute these values, and will propose a shared resource policy that can be used to reduce as much as possible the amount of time for which an application keeps a resource locked, possibly executing it non-preemptively.

Multicore platforms Recent trends in chip design and computer architecture indicate that most processor chips in the future will contain multiple computing "cores" within them. Indeed, both Intel's and AMD's current top-of-the-line processors already contain four computing cores. Based upon fundamental technological barriers and heat-dissipation issues, many technology forecasters predict that Moore's law will be satisfied in the future primarily by continually increasing the number of computing cores per CPU, rather than by developing more powerful single cores. In one approach that is being explored to exploit the tremendous computing capacity that will become available in such multicore processors, the cores are partitioned into different *clusters* and different task systems are executed simultaneously upon each cluster. As with open environments, the sharing of global resources between clusters present a major design challenge, since a system may be blocked from executing upon its cluster while waiting for some other system to release a locked resource. To reduce the duration of such blocking, it is again desirable that the resource holding times be minimized.

Contributions in this paper In this paper, we study real-time systems that can be modelled as collections of *sporadic tasks* (Mok 1983; Baruah et al. 1990), and are implemented upon a platform comprised of a single preemptive processor, and additional resources. We assume that the shared resources are accessed within (possibly nested) critical sections which are guarded by semaphores. For such systems

1. We formally study the concept of resource holding times (RHT's), that quantify the largest amount of time for which a task system may keep a resource locked.
2. We present an algorithm for computing such resource holding times from task system specifications.
3. We present an algorithm for minimizing such resource holding times, when the task system is scheduled using the (preemptive) Earliest Deadline First scheduling algorithm (EDF) (Dertouzos 1974; Liu and Layland 1973), and access to shared resources is arbitrated by the Stack Resource Policy (SRP) (Baker 1991).

4. We derive a resource access policy that generalizes SRP, and prove that RHT's are further reduced if this resource access policy is used in preference to SRP.

Organization The remainder of this paper is organized as follows. In Sect. 2, we present the formal model for resource-sharing sporadic task systems that is used in the remainder of this paper, summarize prior results on feasibility analysis of such resource-sharing sporadic task systems (Sect. 2.2), and present an example to illustrate how the algorithms we will derive in this paper may be used to reduce resource hold times (Sect. 2.1). In Sect. 3, we derive an algorithm for computing such resource holding times. In Sect. 4, 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. In Sect. 5, we derive a resource sharing protocol that generalizes the Stack Resource Policy (SRP), and that permits further reductions in RHT's. We conclude in Sect. 6 with a summary of the results presented in this paper.

2 System model and illustrative example

A real-time task system, denoted by τ, is comprised of n *sporadic* tasks (Mok 1983; Baruah et al. 1990), denoted $\tau_1, \tau_2, \ldots, \tau_n$. Each sporadic task τ_i $(1 \le i \le n)$ is characterized by a worst-case execution time parameter (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 a 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 above parameters are assumed to be in the domain of positive real numbers.

The system is assumed to execute upon a platform comprised of a single preemptive processor, and m other shared resources $R_1, R_2, \ldots, R_m$. The resource requirements of the sporadic tasks may be specified in many ways (e.g., see papers Baker 1991; Lipari and Buttazzo 2000; Pellizzoni and Lipari 2005); 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;
(ii) $S_j^{\max}$ denote the length of the largest critical section holding resource R_j among all tasks in τ:

$$S_j^{\max} = \max_{i=1}^{n}\{S_{ij}\};$$

(iii) C_{ih} 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_h $(i \neq h)$:

$$C_{ih} = \max\{S_{ij} \mid R_j \text{ is accessed by } \tau_h\}.$$

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*: $D_i \le D_{i+1}$

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: $D_i \le D_{i+1}$ 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 time a resource is locked/unlocked. 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 Baker 1991 that a job that begins execution will not subsequently be blocked.)

Fig. 1 EDF + SRP

$(\forall i)$. Second, 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. Third, we will use the convention that right half-open intervals $[t, t')$ represent continuous execution of a task from time t to time t'. Finally, we assume that tasks do not self-suspend execution; i.e., tasks do not transition to a "suspended" state prior to the completion of their job.

EDF + SRP In the remainder of this paper, reasonable familiarity with the Stack Resource Policy (SRP) (Baker 1991) is assumed. When it is used in conjunction with EDF, the rules used by the SRP to determine execution rights are summarized in Fig. 1—see Baker's paper (Baker 1991) for proofs of correctness. We will limit our attention to *work-conserving* scheduling algorithms: an algorithm is work-conserving if it doesn't idle the available processing unit when there is a ready task waiting to be scheduled. In Baruah (2006) Baruah proves that EDF + SRP is an optimal work-conserving policy for sporadic task-system, meaning that if a sporadic task set with shared resources can be scheduled with a given work-conserving scheduling policy, then it can also be scheduled with EDF + SRP.

2.1 Sharing global resources: an example

Most prior open environment designs that permit global resource sharing (e.g. Davis and Burns 2006; Behnam et al. 2006) mandate that such global resources be accessed non-preemptively by each individual application. The intuition behind this approach is sound: by holding global resources for the least possible amount of time, each application minimizes the blocking interference to which it subjects the other applications. However, the downside of such non-preemptive execution is felt *within* each application's task system—by requiring certain critical sections to execute non-preemptively, it is more likely that a system will fail to meet its own (internal/ local) deadlines. In fact, the strategy of non-preemptively executing all critical sections has been previously explored (e.g., see Mok's Ph.D. thesis, Mok 1983). But it was soon recognized that such a strategy creates additional—avoidable—priority inversions: this recognition led to the development of the sophisticated resource-access arbitration protocols such as PCP (Sha et al. 1990) and SRP (Baker 1991).

More specifically, for each individual application that uses global resources, we can distinguish between three different cases:

1. If an application is feasible on its designated fraction of the computing platform when it executes its global resources non-preemptively, then it should indeed execute its global resources non-preemptively, thereby minimizing the interference to which it subjects other applications.
2. If an application is infeasible on its designated fraction of the computing platform when scheduled using EDF + SRP (i.e., when global resources are preempted as they would be under SRP), it follows from the optimality of EDF + SRP (Baruah 2006) that no (work-conserving) scheduling strategy can result in this application being feasible upon its designated fraction of the computing platform.
3. The interesting case is when neither of the two above holds: the application is infeasible on its designated fraction of the computing platform when accessing a global resource non-preemptively, but feasible when access to the global resource is preemptive.

The third case above is the one we address in this paper. More specifically, *we seek to devise a scheduling framework that schedules any feasible task system to meet all deadlines and minimizes the resource holding times*. Note that an application's "designated fraction of the computing platform" may be some value α between zero and one; however, the results contained in this paper will assume that each task's execution has been normalized by α. Therefore, any application running on some fractional portion on a unit-speed processor can logically be viewed as an application executing on a dedicated processor of speed $1/\alpha$. Throughout the remainder of this paper, we will restrict our attention to minimizing the resource-holding times of task systems executing on their own dedicated processing platform.

We illustrate by an example below. The task system in this example is indeed successfully scheduled by EDF + SRP to meet all deadlines—this may be validated using the feasibility test in Baruah (2006). However, EDF + SRP makes no attempt to minimize resource holding times (since doing so was not a design goal of SRP). We demonstrate two approaches for reducing the resource holding times; when used together, these two approaches can cause significant reduction in resource holding times.

Example 1 Consider the application comprised of the following four sporadic tasks given in Fig. 2, executing upon a processor of unit computing capacity. There is one shared resource R_1, which is accessed by both τ_3 and τ_4 within critical sections for the entire duration of their executions.

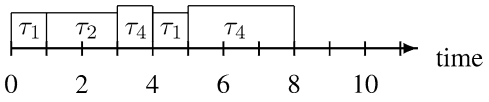
Non-preemptive execution of the critical section could result in a deadline miss: if τ_1 generates a job at the instant that τ_4's job enters its critical section, then τ_1's job would be blocked until its deadline.

Fig. 2 Task parameters for sporadic task system used in Example 1. Also, the WCET of each task's critical section for R_1 is given

	C_i	D_i	T_i	S_{i1}
τ_1	1	4	4	0
τ_2	2	8	8	0
τ_3	2	10	10	2
τ_4	4	16	16	4

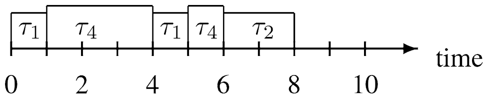
Now let us consider the situation if the local scheduling algorithm used is EDF + SRP (the interested reader may refer to Baruah 2006 for details of the feasibility test). Under SRP, the ceiling of R_1 is 3 ($\Pi(R_1) \leftarrow 3$); i.e., τ_4's execution of the shared resource may be preempted by τ_1 and by τ_2 but not by τ_3. It may be verified, using the test in Baruah (2006), that the system is feasible.

(a) What is the resource holding time? In Sect. 3 below, we present an algorithm for computing such resource holding times; for now, we present an informal argument as to how this may be done. The test of Baruah (2006) reveals that the worst-case blocking scenario occurs when τ_1, τ_2, and τ_3 all begin releasing jobs as frequently as possible immediately after τ_4 has entered its critical section. Assume that τ_4 locks the CS at time-instant zero and that τ_1 and τ_2 generate jobs immediately after the resource lock by τ_4; that is, τ_4 executes a lock of the resource in the interval $[0, \epsilon)$, where $\epsilon > 0$, and both τ_1 and τ_2 generate jobs at time ϵ. As ϵ approaches zero, the resulting EDF + SRP schedule, until the release of the resource R_1, looks as follows:



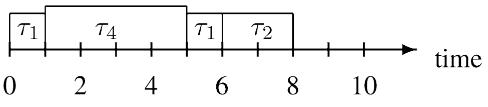
with the critical section executing over $[3 + \epsilon, 4 + \epsilon)$ and $[5 + \epsilon, 8)$ and released at time-instant 8; hence, the resource hold time is equal to 8. (Note, we assume that the resource locking time locking time of ϵ is included in the worst-case execution time, $S_{4,1}$.)

(b) However, observe that it is in fact not necessary to execute τ_2 over $[1 + \epsilon, 3 + \epsilon)$; instead, τ_2's job could have been blocked by the critical section and would still have completed by its deadline. The RHT-minimization strategy presented in Sect. 4 is based on exploiting this fact, by recognizing that the ceiling of R_1 could actually be set to 2 (rather than 3) without compromising feasibility. With $\Pi(R_1) \leftarrow 2$, the schedule over the same time interval as above looks like this:



with the critical section executing over $[1 + \epsilon, 4 + \epsilon)$ and $[5 + \epsilon, 6)$ and released at time-instant 6; hence, the resource hold time is equal to 6.

(c) Depending upon how much one is willing to modify the local resource-access algorithm from "standard" SRP, further reduction in resource holding times may be possible. Such an approach is explored in Sect. 5. With respect to our example above, we notice that while τ_1's jobs cannot be blocked by the entire CS (which has a WCET of 4 time units), each job of τ_1 can however tolerate 3 units of blocking. This fact can be incorporated into the local algorithm which would then recognize that the job of τ_1 arriving at time-instant 4 in the "worst-case" scenario described above needn't preempt the critical section of τ_4, yielding the following schedule over the same time interval as above:



with the critical section executing over $[1+\epsilon, 5)$; hence, the resource hold time is now reduced to 5.

2.2 Feasibility analysis under EDF + SRP scheduling

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

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. It has been shown (Baruah et al. 1990) for a sporadic task τ_i that

$$\text{DBF}(\tau_i, t) = \max\left(0, \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) C_i\right).$$

For task system τ and any non-negative t, we let $\text{DBF}(\tau, t)$ denote the following sum $\text{DBF}(\tau, t) = \sum_{\tau_i \in \tau} \text{DBF}(\tau_i, t)$.

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 Baruah 2006) 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_{jh} denotes the length of the largest critical section in τ_j that holds some resource that is also needed by τ_h. Given these C_{jh}'s, we can easily compute the blocking function $B(L)$ as follows:

$$B(L) = \max\{C_{jh} \mid D_j > L \text{ and } D_h \leq L\}. \tag{1}$$

It has been shown (Baruah 2006) that ensuring

$$B(L) \leq L - \text{DBF}(\tau, L) \tag{2}$$

for all values of $L \geq 0$ is necessary and sufficient for ensuring feasibility—intuitively (see Baruah's paper, Baruah 2006, for a formal proof), for each L the right hand side (RHS) denotes the "slack" left over after the maximum cumulative demand of all the tasks over an interval of length L, while the left hand side (LHS) denotes the maximum possible overhead due to blocking.

From the definition of the blocking function above (1), we note that $B(L) = 0$ for all $L \geq D_n$—since there are no jobs with relative deadline $> D_n$, the index j in (1) will never be instantiated for $L \geq D_n$. Hence for all $L \geq D_n$, checking condition (2) 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 $B(L)$ and $\text{DBF}(\tau, L)$ 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 condition (2) 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 (Baruah et al. 1990) such that, if condition (2) is not violated for some d_k smaller than this upper bound, then condition (2) 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

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

where U_i denotes the utilization (the ratio C_i/T_i) of τ_i; and U denotes the system utilization: $U \stackrel{\text{def}}{=} U_1 + U_2 + \cdots + U_n$. (Recall that we assume tasks are indexed according to non-decreasing order of their relative deadline parameters; hence, D_n equals the largest value of the relative deadline parameter of any task in the task system.) 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 $\mathcal{TS}(\tau)$.

We now derive an implementation of an algorithm for validating, for a given task system τ, whether (2) holds for all $L \in \mathcal{TS}(\tau)$. For this, we need the following lemma.

Lemma 1 *For all L such that $D_i \leq L < D_{i+1}$, $B(L) = B(D_i)$.*

Proof Consider any L such that $D_i \leq L < D_{i+1}$. Since no relative-deadline parameter lies between D_i and L, any pair of relative deadlines (D_j, D_h) such that $((D_j > L)$ and $(D_h \leq L))$ also satisfies $((D_j > D_i)$ and $(D_h \leq D_i))$. Hence by (1), $B(L)$ is identically equal to $B(D_i)$. □

```
ENHANCED PROC. DEMAND TEST($\tau$)
1   $i \leftarrow 0$
2   for each $d_k$ in $\mathcal{TS}(\tau)$, considered in increasing order, do
          ▷ Check feasibility, and compute the $\beta_i$ parameters
3         while ($d_k = D_{i+1}$) do ▷ if multiple tasks have the same relative
                                        deadline parameter
4               $i \leftarrow i + 1$
5               $\beta_i \leftarrow \infty$
6         end while
7         if DBF($\tau, d_k$) $> d_k$ then return "infeasible" end if ▷ ignoring
                                        blocking (which is considered below)
8         $\beta_i \leftarrow \min(\beta_i, d_k - \text{DBF}(\tau, d_k))$
9   end for
    ▷ Now check the effects of blocking
10  for each $i \leftarrow 1$ to $(n - 1)$ do ▷ Does (4) hold?
11        if ($B(D_i) > \beta_i$) then return "infeasible" end if
12  end for
13  return "feasible"
```

Fig. 3 Enhanced Processor Demand Test, which (i) determines whether the input sporadic task system is feasible under EDF + SRP scheduling, and (ii) if feasible, computes the blocking allowances (the β_i's)

Definition 2 For each i, $1 \le i < n$, define the *blocking tolerance* β_i as follows:

$$\beta_i \overset{\text{def}}{=} \min_{D_i \le L < D_{i+1}\}} (L - \text{DBF}(\tau, L)), \tag{3}$$

Observe that in (3) above, β_i takes its value at some value of $L \in \mathcal{TS}(\tau)$ that lies in the interval $[D_i, D_{i+1})$. Hence if τ is infeasible because (2) is violated at some $d_k < D_n$, it must be the case that

$$\exists i : 1 \le i < n : B(D_i) > \beta_i. \tag{4}$$

An EDF + SRP feasibility test is presented in pseudo-code form as Procedure ENHANCED PROC. DEMAND TEST in Fig. 3. The first for-loop (lines 2–9) (i) ignores blocking effects and checks (line 7) whether cumulative processor demand over any interval exceeds the interval length; and (ii) computes the β_i parameters. The second for-loop (lines 10–12) checks condition (4) for each i.

3 Computing the resource hold times

In this section,[2] we describe how resource hold times may be computed for each resource in a given feasible resource-sharing sporadic task system. That is, we assume

[2]Please note the results contained in this subsection contain improvements on earlier results reported in Fisher et al. (2007c). We are grateful to the journal reviewer who suggested a way to "tighten" the computed value of RHT(R_j, τ_i).

that the input task system has been deemed feasible (e.g., by the feasibility-testing algorithm described in Sect. 2.2), 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 $\tau_1, \dots, \tau_{\Pi(R_j)-1}$ 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 $\tau_\ell \in \{\tau_1, \dots, \tau_{\Pi(R_j)-1}\}$ could preempt τ_i while 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 $\tau_1, \dots, \tau_{\Pi(R_j)-1}$ (otherwise, τ_i could not execute at time t_{ij}). Thus, only jobs of $\tau_1, \dots, \tau_{\Pi(R_j)-1}$ 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, the maximum number of jobs of τ_ℓ that have both arrival times and deadline in the half-open interval of length t is $\max(0, \lceil \frac{t-D_\ell}{T_\ell} \rceil)$. Since the earliest time at which job J_{ik} could acquire resource R_j is a_{ik} and since $D_i \geq D_\ell$, an upper bound on the number of times task τ_ℓ can preempt τ_j while τ_j is holding resource R_j is:
$$\left\lceil \frac{D_i - D_\ell}{T_\ell} \right\rceil. \tag{5}$$
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:
$$\left\lceil \frac{t}{T_\ell} \right\rceil. \tag{6}$$
Combining (5) and (6), we may obtain an overall upper bound on the number of preemptions by task τ_ℓ:
$$\left\lceil \frac{\min(t, D_i - D_\ell)}{T_\ell} \right\rceil. \tag{7}$$

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 (7), 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:"

$$\text{RBF}(\tau_\ell, \tau_i, t) \stackrel{\text{def}}{=} \left\lceil \frac{\min(t, D_i - D_\ell)}{T_\ell} \right\rceil \cdot C_\ell. \tag{8}$$

The cumulative execution requests of jobs of tasks $\{\tau_1, \ldots, \tau_{\Pi(R_j)-1}\}$ 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:

$$W_i(t) \stackrel{\text{def}}{=} S_{ij} + \sum_{\ell=1}^{\Pi(R_j)-1} \text{RBF}(\tau_\ell, \tau_i, t). \tag{9}$$

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 Joseph and Pandya (1986); Lehoczky et al. (1989), we can obtain t_i^* in time complexity that is pseudo-polynomial in the parameters of $\{\tau_1, \ldots, \tau_{\Pi(R_j)-1}\} \cup \{\tau_i\}$. Note that a fixed-point must exist since $W_i(t)$ is left-continuous, non-decreasing, and for all time $t' > t_{\text{last}} \stackrel{\text{def}}{=} \max_{\ell \in \{1,\ldots,\Pi(R_j)-1\}}(D_i - D_\ell)$, $W_i(t')$ equals $W_i(t_{\text{last}})$; that is, eventually $W_i(t)$ ceases to increase and the left-continuous, non-decreasing property ensures that if $W_i(t) < t$ then there must exist a $t' < t$ where $W_i(t') = t'$. Furthermore, the range of $W_i(t)$ is finite; thus, the iterative techniques from Joseph and Pandya (1986); Lehoczky et al. (1989) are guaranteed to terminate. By the next theorem, t_i^* is the maximum amount of time τ_i can hold resource R_j. Thus,

$$\text{RHT}(R_j, \tau_i) \stackrel{\text{def}}{=} t_i^*. \tag{10}$$

Theorem 1 *The maximum resource-holding time*, $\text{RHT}(R_j, \tau_i)$, *of resource* R_j *by task* τ_i *under* EDF + SRP *is equal to the smallest fixed point of* $W_i(t)$ (*i.e.*, t_i^*).

Proof To show that t_i^* is equal to the $\text{RHT}(R_j, \tau_i)$, we must prove two statements:

1. if τ_i locks resource R_j at time t_{ij}, τ_i will complete execution of its critical section for R_j by time $t_{ij} + t_i^*$;
2. there exists some legal sequence of job arrivals for τ that cause τ_i to complete the execution of its critical section at exactly time $t_{ij} + t_i^*$.

In order to prove Statement 1, we will show that τ_i will complete its critical section execution of resource R_j by time $t_{ij} + t_i^*$ under the "worst-case" arrival sequence. Recall from the definition of EDF+SRP that no task with index $\geq \Pi(R_j)$ can preempt job J_{ik} while it locks R_j. Therefore only tasks $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ can execute from time t_{ij} when J_{ik} acquires R_j until τ_i releases the lock. There is a direct analogy for computing $\text{RHT}(R_j, \tau_i)$ to calculating the worst-case response time of a job in a fixed-priority system. $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ are all the tasks that generate jobs of (potentially) higher priority than task τ_i while it is executing a critical section for R_j. The *critical instant theorem* (Liu and Layland 1973) states that in a fixed-priority system the worst-case response time of a job of a given task τ_i occurs when it is released simultaneously with jobs of all tasks with higher-priority than τ_i and each higher-priority task releases subsequent jobs as soon as legally possible. Let $t_{ij} + \epsilon$ (where $\epsilon > 0$) be the earliest time after t_{ij} that a job of any task of $\{\tau_1, \ldots, \tau_{\Pi(R_j)-1}\}$ could arrive. By identical argument to the critical instant theorem, the maximum resource-hold time for job J_{ik} and resource R_j occurs when tasks that may preempt J_{ik} (i.e. tasks $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$) simultaneously release jobs ϵ time after J_{ik} has acquired R_j, and subsequent jobs are released as soon as legally possible; hereafter, we will refer to this arrival sequence as the "critical-instant arrival sequence." Observe if the release time of one of these higher-priority jobs J_ℓ occurs later than $t_{ij} + \epsilon$, the amount of pre-emption of J_{ik} by τ_ℓ while J_{ik} is holding R_j could never increase. Similarly, moving a job earlier is not possible: by the third step in computing $\mathrm{RHT}(R_j, \tau_i)$ above, there are no active jobs for tasks $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ at time t_{ij}. Moving the activation of the first job of a task of $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ forward in time would either contradict this property (if the job is still active at time t_{ij}), or would result in a lower interference (if the job completed its execution at time t_{ij}). Moving subsequent jobs of tasks $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ earlier in time would instead produce an illegal sequence. Moving subsequent jobs later would force later jobs of $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ to have deadline greater than J_{ik}'s deadline and decrease the amount that such a task executes while τ_i is in its critical section. Thus, the worst-case response time for the execution of τ_i's critical section on R_j occurs when all tasks $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ release jobs simultaneously at ϵ at $t_{ij} + \epsilon$, with ϵ approaching zero, and all subsequent jobs of $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ arrive as quickly as possible. By the logic of Steps 3 and 4 of computing $\mathrm{RHT}(R_j, \tau_i)$, the worst-case execution requirements of J_{ik} and jobs of any task $\tau_j \in \{\tau_1, \ldots, \tau_{\Pi(R_j)-1}\}$ that may preempt J_{ik} while accessing R_j over an interval of length t are exactly expressed by $W_i(t)$.

By definition, t_i^* is the smallest fixed point of $W_i(t)$, i.e., $W_i(t_i^*) = t_i^*$. For the devised release sequence, $\mathrm{RBF}(\tau_\ell, \tau_i, t_i^*)$ 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 $(t_{ij}, t_{ij} + t_i^*)$. Thus $W_i(t_i^*)$ 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 $(t_{ij}, t_{ij} + t_i^*)$ 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. Since $W_i(t_i^*) = t_i^*$ and $W_i(t_i^*)$ represents the worst-case execution requirement of "higher-priority" jobs of $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ that could arrive in $[t_{ij} + \epsilon, t_{ij} + t_i^*)$ plus the worst-case execution time for a critical section of τ_i on resource R_j, 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 $t_{ij} + t_i^*$.

To see that Statement 2 also holds, observe that for the worst-case arrival sequence $W_i(t)$ is an exact quantification (as ϵ tends to zero) on the execution requirements of jobs of $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ and J_{ik} over the interval $[t_{ij} + \epsilon, t_{ij} + t_i^*)$. Therefore, t_i^* is equal to $\mathrm{RHT}(\tau_i, t)$. □

4 Minimizing RHT's within the SRP

We now address the issue of *decreasing* these RHT's, without making any changes to the scheduling framework (EDF + SRP) used. More specifically, 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.*

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 1983) 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 (Baruah 2006, 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.

Recall that we desire to continue to use the EDF + SRP scheduling framework, while reducing resource holding times if possible. We achieve this by modifying the parameters of 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. The details of such modification are presented below.

4.1 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 Sect. 4.2, 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. By definition of the SRP, 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

```
REDUCECEILING($R_j$)
      ▷ Suppose that $\Pi(R_j) \equiv (i+1)$; can we reduce it to $i$?
1     if ($S_j^{\max} > \beta_i$) then return "false" end if ▷ Cannot reduce $\Pi(R_j)$
      ▷ Decrease $R_j$'s preemption ceiling to $i$
2     Add a zero-WCET critical section in $\tau_i$, accessing $R_j$
3     return "true" ▷ Have reduced $\Pi(R_j)$ by 1

MINCEILING($R_j$)
      ▷ Reduce $R_j$'s preemption ceiling as much as possible
1     while ($\Pi(R_j) > 1$) do
2           if (REDUCECEILING($R_j$) == "false") break done
3     return
```

Fig. 4 Reducing the preemption ceilings for R_j

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 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 RHT(R_j, τ_i).
> Observe that adding such a critical section effectively decreases the preemption ceiling of R_j by one. Hence, it may be hoped that RHT(R_j, τ_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 Fig. 4, 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.

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 jobs with deadline $< D_i$ in the critical-instant arrival sequence 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 $(i+1)$ and hence critical sections holding R_j were already able to block (directly or indirectly) jobs with deadline $\geq D_{i+1}$ in the critical-instant arrival sequence.

In fact, $B(L)$ may change only for L within the range $[D_i, D_{i+1})$. And for all such L, Lemma 1 tells us that $B(L) = B(D_i)$. It therefore remains to determine the value that $B(D_i)$ would assume if we made the change and reduced R_j's ceiling to i. In fact $B(D_i)$ must be modified as shown below:

$$B(D_i) \leftarrow \max\Bigl(B(D_i),\ \max_{\ell=i+1}^{n}\{S_{\ell j}\}\Bigr). \tag{11}$$

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 $B(D_i)$. In that case, the value of $B(D_i)$ must be updated, as shown above in (11).

Hence to determine whether R_j's preemption ceiling can indeed be reduced from $(i+1)$ to i, we must re-evaluate condition (4) for i to determine whether it remains false when $B(D_i)$ is updated according to (11) above. Anyway, since it is assumed that the task system is feasible when R_j's preemption ceiling $\Pi(R_j) = (i+1)$, we can simplify this re-evaluation of condition (4) for i, based on the following reasoning. If the new value of $B(D_i)$, as computed in (11), were equal to the original value (i.e., the outer "max" in (11) returns the first term), then the value of condition (4) cannot have changed. Hence the re-evaluation only needs to be done if the outer "max" equals the second term: $\max_{\ell=i+1}^{n}\{S_{\ell j}\}$. Alternatively, we could simply check condition (4) with $B(D_i)$ 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.

Moreover, since we deemed the system initially feasible, we need to consider only i indexes below the original resource ceiling (the one without any artificial critical section). Giving all dummy critical sections a null WCET, it follows that $\max_{\ell=i+1}^{n}\{S_{\ell j}\}$ equals $S_j^{\max}$. Therefore, we can use this last term every time we need to verify if the ceiling of resource R_j could be decreased by one. This is the strategy adopted in Procedure REDUCECEILING(R_j). The conditional in Line 1 is the re-evaluation of condition (2) if $\Pi(R_j)$ were to take on the value i. If condition (4) now holds for i, then this reduction in $\Pi(R_j)$ will cause the system to become infeasible; hence Procedure REDUCECEILING(R_j) returns "false" without making the change.

On the other hand, if condition (4) continues to not hold, then $\Pi(R_j)$ may safely be decreased by one. This is done in line 2 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 Fig. 4.

4.2 Reducing RHT's for all resources

In this section, we describe how the algorithms of Sect. 4.1 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 2 *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 Suppose that procedure 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 REDUCECEILING(R_p), we are guaranteed that the sporadic task system is feasible prior to the call to REDUCECEILING(R_j).

Observe from the pseudo-code (Fig. 4) that the success or failure of the call to REDUCECEILING(R_j) is determined by the value of $S_j^{\max}$: the call fails if this exceeds β_i, and succeeds otherwise. Since the values of $S_j^{\max}$ and β_i are 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. □

As a consequence of Lemma 2, it follows that the order in which we choose to make calls to Procedure REDUCECEILING (and hence, to Procedure MINCEILING)

```
REDUCEALL($\tau$)
    ▷ There are $m$ resources, labelled $R_1, \ldots, R_m$
1   for $j \leftarrow 1$ to $m$ MINCEILING($R_j$)
```

Fig. 5 Reducing Preemption ceilings for all resources

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 Fig. 5, which considers the resources in the order in which they are indexed.

Using the modified ceilings given by Procedure REDUCEALL we obtain a scheduling policy that is superior to both the non-preemptive execution of critical sections and normal EDF + SRP. If a resource can be executed non-preemptively, then our algorithm will reduce the resource ceiling to the lowest level, effectively executing the critical section non-preemptively. If instead a non-preemptive execution of the critical section would cause other tasks of the same application to miss their deadlines, our algorithm would allow some degree of preemption from higher priority tasks of the same application: the ceiling is reduced to the lowest possible level that retains the schedulability of the application. If this level equals the original SRP level, then our algorithm will behave in the same way as EDF + SRP, inheriting its optimality (Baruah 2006). If an application cannot be scheduled with our algorithm, then no other policy would find a feasible schedule.

4.3 Computational complexity

As can be seen from the pseudo-code in Fig. 4, condition (2) 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 (2) 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, we can decrease the computational complexity by requiring some additional “book-keeping” operations during the initial pseudo-polynomial feasibility analysis. This is what is done in Procedure ENHANCED PROC. DEMAND TEST(τ), where all blocking tolerances β_i are computed and stored to memory during feasibility-analysis of τ, without increasing the computational complexity by more than a constant factor.

When all blocking tolerances are already available, the complexity of Procedure REDUCECEILING(R_j) decreases, so that the overall complexity of Procedure REDUCEALL(τ) becomes polynomial in the number of tasks.

4.4 Relation to prior work

The method we propose to reduce the resource ceilings presents some analogies with the work proposed by Saksena and Wang (2000). Saksena and Wang's proposed method increases the priority of a task after it starts executing (when possible) to avoid unnecessary preemptions. The priority is raised to the maximum level that doesn't compromise system schedulability. In other words, when a task is scheduled for execution, its priority is raised to a *preemption threshold*, such that (i) all tasks with priority lower than this threshold cannot preempt the executing task, and (ii) all higher priority tasks below the threshold won't miss any deadline due to the blocking suffered for a delayed preemption. In this way, the preemption overhead decreases and the schedulability is preserved.

Additional works by other researchers extend the preemption threshold approach to dynamic priority scheduling. Gai et al. (2001) propose the *stack resource policy with preemption thresholds* (SRPT), a method to uniformly deal with the blocking due to tasks with higher preemption thresholds or to tasks accessing shared resources. Ghattas and Dean (2007) propose a framework that applies to both static-priority and dynamic-priority scheduling, optimizing the selection of preemption thresholds for systems with limited memory and reduced stack size.

This paper differs from previous works on preemption thresholds in several ways. First, we are considering shared resources to reduce the amount of time for which a resource can be held by a task. Our target is to decrease the resource holding time instead of limiting the number of preemptions or the memory usage. Second, our task model is more general, in that it applies as well to systems with deadlines different from periods, while previous works assumed deadlines equal to periods. Finally, the complexity of our algorithm to minimize the resource holding times is lower than the overall complexity of the algorithm proposed in Saksena and Wang (2000) to maximize the preemption thresholds. By exploiting information from initial schedulability analysis, we can optimally reduce the ceilings of all resources in polynomial time, as explained in the previous subsection.

5 Modifying the scheduling framework to reduce RHT

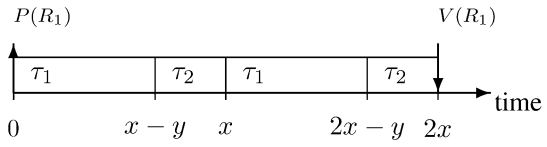
In Sect. 4 above, we restricted ourselves to remain within the EDF + SRP scheduling framework: our objective was to minimize the resource holding times while remaining within this framework. We now eliminate the restriction that we be required to use the EDF + SRP scheduling framework, and explore whether further reductions in RHT's are possible.

5.1 Overview and example

We first provide a broad overview of our proposed approach. We will continue to use EDF as the scheduling algorithm, and will use a resource-access arbitration protocol that is based upon SRP. More specifically, as in SRP we define a system ceiling at each instant in time during run-time; unlike in SRP, however, this is not simply the minimum of the ceilings of all locked resources. Instead, the system ceiling at any point in time depends upon the *remaining* WCET's of each of the critical sections that are currently holding resources. If the remaining WCET of a currently-executing critical section is small enough that continuing to execute it will not cause a higher-priority (earlier deadline) job to miss its deadline, then we block the higher-priority job (equivalently, the system ceiling is set to a smaller value than it would be under "regular" SRP, in that it allows for the blocking of this higher-priority, lower indexed, job), and continue executing the critical section. This allows the critical section to complete, and release the shared resource, sooner than if it were to be preempted by the higher-priority job. We illustrate with an example.

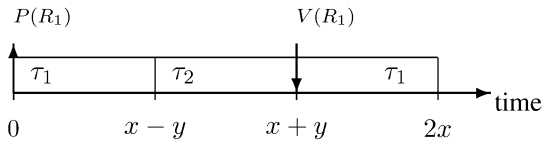
Example 2 Consider a system comprised of two sporadic tasks τ_1 and τ_2, with $\tau_1 = (x-y, x, x)$ and $\tau_2 = (2y, \infty, \infty)$. Task τ_1 does not use any resource other than the processor; task τ_2's jobs use a global resource R_1 throughout their execution.

The ceiling $\Pi(R_1)$ of resource R_1 is equal to 2. It may be verified that since $\beta_1 = x - (x-y) = y$ while $S_1^{\max} = 2y$, Procedure MINCEILING(R_1) of Sect. 4 returns 2; i.e., we cannot use the technique of Sect. 4 above to reduce this ceiling to 1. Consequently the "worst case" scenario from the perspective of maximizing the resource holding time RHT(R_1, τ_2) is as illustrated in the following diagram:



That is, a job of τ_1 arrives immediately after τ_2 acquires a lock on R_1, and subsequent jobs of τ_1 arrive as soon as legally permitted to do so. We can see from the diagram that RHT(R_1, τ_2) is $2x$.

Now if we recognized that at time-instant x the critical section of task τ_2 has only y units of execution remaining, and that this could be accommodated in τ_1's "slack," then it would not be necessary to preempt τ_2. In other words, the system ceiling could be dynamically changed from 2 to 1 at this point in time, yielding the following schedule:



and a resource holding time RHT(R_1, τ_2) of $x+y$.

Thus if $y < x$ the second strategy of dynamically adjusting the system ceiling during run-time results in lower RHT's while retaining feasibility; in the extreme case when $y \ll x$, RHT(R_1, τ_2) is reduced by a factor of almost two. Generalizing the previous example using tasks $\tau_1 = (x-(k-1)y, x, x)$ and $\tau_2 = (ky, \infty, \infty)$, with

$x > (k-1)y$, we can see that there are cases in which the dynamic strategy can reduce the resource holding time by an arbitrarily high factor k.

5.2 The modified resource access protocol

In Example 2 above, observe that τ_1 needed to be able to preempt τ_2's critical section as long as the remaining WCET of the critical section is greater than β_1, task τ_1's blocking tolerance as defined in Definition 2. Once the critical section has executed enough that at most β_1 units of its execution remains, however, this critical section may block jobs of τ_1 without endangering these jobs' ability to meet their deadlines. In other words, the ceiling of resource R_1 can be *changed* from 2 to 1 once there is β_1 units of execution remaining in the critical section.

In our proposed resource access protocol, the ceiling of a resource is *dynamic* rather than static. We denote this dynamic ceiling of resource R_j at the current instant during run-time as $\Pi^D(R_j)$, and describe below how these dynamic ceilings are determined during run-time. Our modified scheduling framework differs from the EDF + SRP scheduling framework presented in Fig. 1 only in the determination of system ceilings (Rule 2 in Fig. 1): at each instant during run-time, we set the system ceiling to equal the minimum *dynamic* ceiling $\Pi^D(R_j)$ of any resource R_j that is currently being held by some job.

To change the value of $\Pi^D(R_j)$ during run-time, we insert short "ceiling changing" commands (in the spirit of the *schedule carrying code*[3] introduced by Henzinger et al. 2003) at specific points into critical sections that lock resource R_j. These commands, denoted CEILCHANGE(R_j, i) change the dynamic priority of R_j by assigning it the value i:

$$\text{CEILCHANGE}(R_j, i) \equiv \Pi^D(R_j) \leftarrow i.$$

The pseudocode in Fig. 6 describes where these ceiling changing commands are inserted in each critical section. Procedure INSERTCEILCHANGE(τ_ℓ, R_j) is called for each critical section of each task τ_ℓ that locks resource R_j. The explanation for this pseudo-code is as follows:

- Since the task set is assumed schedulable with EDF + SRP, all jobs with deadline later than $D_{\Pi(R_j)}$ in the critical-instant arrival sequence can already be blocked (directly or indirectly) by tasks accessing R_j, for the whole duration of the critical section. Therefore, at the start of each critical section accessing R_j, the dynamic ceiling $\Pi^D(R_j)$ is set equal to $\Pi(R_j)$, where $\Pi(R_j)$ denotes the "conventional" (as previously defined in this paper—see Fig. 1) ceiling of resource R_j.
- As proved in Sect. 2.2, all jobs with deadline in $[D_{\Pi(R_j)-1}, D_{\Pi(R_j)})$ in the critical-instant arrival sequence can tolerate $\beta_{\Pi(R_j)-1}$ units of blocking. Remember from previous point that later jobs of $\tau_{\Pi(R_j)-1}$ can already be blocked by critical section accessing R_j. We can then forbid $\tau_{\Pi(R_j)-1}$ from preempting critical sections accessing R_j once we are $\beta_{\Pi(R_j)-1}$ time units from the end of the critical section. We

[3] As in Henzinger et al. (2003), we assume that such schedule carrying code is very short, and hence has zero execution time.

```
INSERTCEILCHANGE($\tau_\ell, R_j$)
    ▷ A critical section (CS) in $\tau_\ell$, locking resource $R_j$. $S_{\ell,j}$ denotes the
      WCET of this CS.
1   insert a CEILCHANGE($R_j, \Pi(R_j)$) command at the start of the CS
2   $X_{\Pi(R_j)} \leftarrow S_{\ell j}$        ▷ $X_1, X_2, \ldots, X_{\Pi(R_j)}$ are temporary variables
3   for $i \leftarrow \Pi(R_j) - 1$ down to 1 do
4       $X_i \leftarrow \min(X_{i+1}, \beta_i)$   ▷ At this point, $X_i$ contains the value
                                  $\min(\beta_i, \beta_{i+1}, \ldots, \beta_{\Pi(R_j)-1}, S_{\ell j})$
5       insert a CEILCHANGE($i$) command $X_i$ execution units from insert
        a CEILCHANGE($R_j, i$) command $X_i$ execution units from he end
        of the CS
6   end for
```

Fig. 6 Ceiling change points

can exploit this fact by changing the dynamic ceiling $\Pi^D(R_j)$ of resource R_j to $(\Pi(R_j) - 1)$ once we are $\beta_{\Pi(R_j)-1}$ time units from the end of the critical section.

- Sequentially repeating the previous step for each task $\tau_i \in \{\tau_{\Pi(R_j)-2}, \ldots, \tau_1\}$, we find that every job of τ_i is able to tolerate at least $X_i = \min(\beta_i, \beta_{i+1}, \ldots, \beta_{\Pi(R_j)-1})$ time-units of blocking. Therefore, it is possible to forbid τ_i from preempting critical sections accessing R_j once we are X_i time units from the end of the critical section. This can be achieved by inserting a CEILCHANGE(R_j, i) command X_i execution units from the end of each CS accessing R_j, as in the for-loop of Procedure INSERTCEILCHANGE(τ_ℓ, R_j).

Theorem 2 *Any task system that is feasible under* EDF + SRP *also meets all deadlines when scheduled using* EDF *scheduling in conjunction with this modified resource access protocol.*

Proof Suppose a task system τ is feasible by EDF + SRP but cannot be scheduled by the modified resource access protocol. Thus, there must exist a set of jobs $\mathcal{J}$ generated by τ that will meet all deadlines for EDF + SRP, but miss a deadline under the modified protocol. Consider the schedule $\mathcal{S}$ for $\mathcal{J}$ under the modified protocol, and let t be the earliest time that some job of $\mathcal{J}$ misses a deadline in $\mathcal{S}$. Let t_0 be the last time prior to t such that there are no pending jobs (with respect to schedule $\mathcal{S}$) with deadlines at or before t. Baker (1991) proved that for any such interval $[t_0, t)$ there can be at most *one* job with deadline $> t$ that executes during $[t_0, t)$ under SRP. An identical proof may be used to show that this property holds for the modified protocol. Thus, all other jobs executed in $\mathcal{S}$ during $[t_0, t)$ must have deadlines at or before t and be pending at some point in $[t_0, t)$; define $\mathcal{J}' \subseteq \mathcal{J}$ to be this set of jobs. Note that every job of $\mathcal{J}'$ must have arrival times also in the interval $[t_0, t)$ (otherwise, this would contradict our choice of t_0) and is, therefore, generated by a task $\tau_\ell \in \tau$ such that $D_\ell \leq t - t_0$. Let τ_i be the task with largest relative deadline not exceeding $t - t_0$ (i.e., $\tau_i \overset{\text{def}}{=} \arg\max_{\tau_i \in \tau : D_i \leq (t-t_0)}\{D_i\}$).

Because τ is feasible under EDF + SRP, (2) implies

$$\text{DBF}(\tau, t - t_0) \leq t - t_0.$$

By definition of demand bound function, $\text{DBF}(\tau, t - t_0)$ is an upper bound on the total execution requirement of $\mathcal{J}'$ over the interval $[t_0, t)$. Thus, the total execution requirement of $\mathcal{J}'$ cannot exceed $t - t_0$. Since a deadline miss occurred at time t and the execution requirements of $\mathcal{J}'$ over $[t_0, t)$ do not exceed $t - t_0$, there must exist *exactly* one blocking job, J_{block}, with deadline $> t$ that executes over $[t_0, t)$. For a deadline miss to occur at time t, the following condition must hold.

$$(\text{Total Execution by } J_{\text{block}} \text{ over } [t_0, t)) + (\text{Execution Requirement of } \mathcal{J}') > t - t_0. \tag{12}$$

Equation (12) and the fact that $\text{DBF}(\tau, t - t_0)$ is an upper bound on the execution requirement of $\mathcal{J}'$ imply that the total execution of J_{block} in $\mathcal{S}$ over $[t_0, t)$ exceeds $[(t - t_0) - \text{DBF}(\tau, t - t_0)]$.

Observe that J_{block} has deadline $> t$; thus, J_{block} must arrive prior to t_0 and hold a lock on some resource R_j with $\Pi^D(R_j) \le i$ at time t_0 (otherwise, J_{block} would not be able to block any jobs of $\mathcal{J}'$ according to the modified protocol). Therefore, J_{block} must have been generated by some task $\tau_k \in \tau$ with $D_k > t - t_0$. Let z_{t_0} be the remaining execution at time t_0 of J_{block} on a critical section locking R_j. By the statement following (12),

$$z_{t_0} > [(t - t_0) - \text{DBF}(\tau, t - t_0)]. \tag{13}$$

According to $\text{INSERTCEILCHANGE}(\tau_k, R_j)$, it must be that either $\Pi(R_j)$ is less than or equal to i or that $\Pi^D(R_j)$ is set to i with X_i execution units remaining in a critical section locking R_j. In the case that $\Pi(R_j)$ is at most i, notice that, according to Definition 1, $B(t - t_0) \ge z_{t_0}$ because $D_i \le t - t_0$ and $D_k > t - t_0$. However, (13) implies that $B(t - t_0) > [(t - t_0) - \text{DBF}(\tau, t - t_0)]$ which contradicts (2) and the fact that τ is feasible by EDF + SRP. Therefore, $\Pi(R_j)$ must exceed i. In the case that $\Pi^D(R_j)$ is set to i with X_i execution units remaining, X_i equals $\min(\beta_i, \beta_{i+1}, \ldots, \beta_{\Pi(R_j)-1}, S_{kj})$. By the value of X_i, the definition of β_i (3), and the fact that $D_i \le t - t_0 < D_k$, it must be that $X_i \le [(t - t_0) - \text{DBF}(\tau, t - t_0)]$. Equation (13) implies that $X_i < z_{t_0}$. However, since $\Pi(R_j)$ must exceed i, $\Pi^D(R_j)$ cannot be $\le i$ with more than X_i units of execution in the critical section. Therefore, $\Pi^D(R_j) > i$ at time t_0. This contradicts the execution of J_{block} over $[t_0, t)$. In both cases, we have derived a contradiction. Hence, our assumption that τ is not schedulable according to the modified resource access protocol is false; the theorem immediately follows. □

The above theorem proves the optimality of our modified resource access protocol with relation to the schedulability of sporadic task sets accessing shared resources. Moreover, since critical sections are less likely to be preempted under the modified resource access protocol than under SRP, it follows that

Lemma 3 *The resource holding time* $\text{RHT}(R_j)$ *of resource* R_j *is never greater, and may be less, under the modified resource access protocol than under* EDF + SRP *scheduling, for all resources* R_j.

Even if Lemma 3 states that the proposed modified resource access protocol is superior to conventional SRP with relation to resource holding times, it is an open question whether resource-holds times can be further minimized. In regards to computational complexity, it is easy to verify that the computation of the ceiling-change points has polynomial run-time complexity, once the blocking tolerances have been computed during the initial feasibility analysis.

5.3 Computing $\mathrm{RHT}(R_j, \tau_i)$ for the modified protocol

The computation of resource-hold times for the modified protocol is closely related to the computation of resource-hold times for EDF + SRP (described in Sect. 3). The major difference for the modified protocol is that the worst-case response times to each "ceiling-change" command must be computed before the overall resource-hold time may be determined; in contrast, for EDF + SRP only the worst-case response time for a task's execution on a critical section for shared resource R_j must be calculated. In this subsection, we will describe a method for computation of $\mathrm{RHT}(R_j, \tau_i)$ under the modified protocol and prove the correctness of the computation.

Assume that task τ_i obtains a lock on shared resource R_j at time t_{ij}. Let the set $\{X_{\Pi(R_j)}, X_{\Pi(R_j)-1}, \ldots, X_1\}$ be the X_ℓ variables computed during the procedure INSERTCEILCHANGE(τ_i, R_j). The following observations are helpful for calculating $\mathrm{RHT}(R_j, \tau_i)$:

1. The maximum response time from t_{ij} to a call to CEILCHANGE(R_j, ℓ) occurs when each task $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ releases a job "immediately" after t_{ij}, and successive jobs are released as soon as legally possible. This arrival situation is identical to the critical-instant arrival sequence described in Theorem 1. We will formally prove this observation in Lemma 4.
2. Task τ_i must execute for exactly $S_{ij} - X_\ell$ time before CEILCHANGE(R_j, ℓ) is executed by τ_i while in its critical section for R_j. This observation follows immediately from the insertion of the ceiling-change commands by INSERT-CEILCHANGE(τ_i, R_j).
3. After CEILCHANGE(R_j, ℓ) has been executed by τ_i, only jobs of tasks $\tau_1, \tau_2, \ldots, \tau_{\ell-1}$ can preempt the execution of τ_i's critical section on shared resource R_j.
4. The function $\mathrm{RBF}(\tau_\ell, \tau_i, t)$ (8) continues to represent an upper bound on the cumulative execution of task τ_ℓ that can preempt τ_i during its critical section access in the interval $(t_{ij}, t_{ij} + t)$. Let t' be the time at which τ_i executes the CEILCHANGE(R_j, ℓ) command. For all t such that $0 < t < t'$, $\mathrm{RBF}(\tau_\ell, \tau_i, t)$ is a tight upper bound on the execution of τ_ℓ that could preempt τ_i's critical section in the interval $(t_{ij}, t_{ij} + t)$. For all $t \geq t'$, $\mathrm{RBF}(\tau_\ell, \tau_i, t')$ is a tight upper bound; this follows from observation (3) that τ_ℓ could not preempt τ_i's critical section after time t'.

Observation (1) implies that the response time to the CeilChange(R_j, ℓ) command from t_{ij} can be determined by restricting attention to the critical-instant arrival sequence. By observation (2), τ_i contributes $S_{ij} - X_\ell$ time to execution up until the call CeilChange(R_j, ℓ). The execution of tasks $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ contributing to the response time of CeilChange(R_j, ℓ) is dependent on the response times of previous commands CeilChange$(R_j, \ell + 1), \ldots,$ CeilChange$(R_j, \Pi(R_j) - 1)$, according to observations (3) and (4). The response time of CEILCHANGE(R_j, ℓ)

can thus be obtained by determining the response time of the previous ceiling change commands. Let $t_i^*(k)$ be the response time of $\textsc{ceilChange}(R_j, k)$ after t_{ij}. The below expression describes the cumulative execution requirements of tasks $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ and the execution of τ_i up to $\textsc{ceilChange}(R_j, \ell)$ over the interval $(t_{ij}, t_{ij} + t)$, assuming the response time for the previous change commands (i.e., $t_i^*(\Pi(R_j) - 1), \ldots, t_i^*(\ell + 1)$) have been determined.

$$W_i^{(\ell)}(t) \overset{\text{def}}{=} (S_{ij} - X_\ell) + \sum_{k=1}^{\ell} \text{RBF}(\tau_k, \tau_i, t) + \sum_{k=\ell+1}^{\Pi(R_j)-1} \text{RBF}(\tau_k, \tau_i, \min(t, t_i^*(k))). \tag{14}$$

Similar to Sect. 3, we may use techniques from Joseph and Pandya (1986); Lehoczky et al. (1989) to find the smallest fixed point of (14). For reasons identical to those stated in Step 5 of Sect. 3, $W_i^{(\ell)}(t)$ is guaranteed to have a fixed point, and an iterative procedure for determining its fixed point is guaranteed to terminate. The next lemma shows that the smallest fixed point for (14) corresponds to the maximum response time to the $(\Pi(R_j) - \ell)$th ceiling change command.

Lemma 4 *Let $t_i^*(\ell)$ be the smallest fixed point of $W_i^{(\ell)}(t)$ (i.e., $W_i^{(\ell)}(t_i^*(\ell)) = t_i^*(\ell)$). The maximum response time from the time τ_i locks resource R_j to the command* $\textsc{ceilChange}(R_j, \ell)$ *is $t_i^*(\ell)$. Furthermore, $t_i^*(\ell)$ is the response time obtained if all tasks $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ simultaneously release a job ϵ time after τ_i locks the resource* (*where ϵ approaches zero*), *and successive jobs are released as soon as legally permitted.*

Proof We will show the lemma by induction on ℓ. First, we prove the base case, $\ell = \Pi(R_j) - 1$; that is, we will show that $t_i^*(\Pi(R_j) - 1)$ is the maximum response time from t_{ij} to $\textsc{ceilChange}(R_j, \Pi(R_j) - 1)$. Note, by observation (2), τ_i executes for $S_{ij} - X_{\Pi(R_j)-1}$ time prior to the call to $\textsc{ceilChange}(R_j, \Pi(R_j) - 1)$. From t_{ij} to this first ceiling-change command there are no other ceiling change command. Notice, because there are no intervening ceiling-change commands between t_{ij} and the first command, the logic of the five steps of Sect. 3 and Theorem 1 applies directly to calculating the response time to the call $\textsc{ceilChange}(R_j, \Pi(R_j) - 1)$. Thus, the maximum response time to $\textsc{ceilChange}(R_j, \Pi(R_j) - 1)$ can be calculated by finding the smallest fixed point of $W_i^{(\Pi(R_j)-1)}(t)$ which by definition is $t_i^*(\Pi(R_j) - 1)$. Furthermore, by the arguments of Theorem 1, the value $t_i^*(\Pi(R_j) - 1)$ corresponds to the response time for $\textsc{ceilChange}(R_j, \Pi(R_j) - 1)$ under the arrival scenario of each task $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ releasing jobs simultaneously ϵ after t_{ij} (where ϵ approaches zero) and subsequent jobs as soon as legally permitted.

For the inductive hypothesis, assume that the smallest fixed points $t_i^*(\Pi(R_j) - 1), \ldots, t_i^*(\ell + 1)$ for the equations $W_i^{(\Pi(R_j)-1)}(t), \ldots, W_i^{(\ell+1)}(t)$, respectively, are the maximum response times from t_{ij} to the respective ceiling-change commands $\textsc{ceilChange}(R_j, \Pi(R_j) - 1), \ldots, \textsc{ceilChange}(R_j, \ell + 1)$. Furthermore, assume for each of these fixed points that the critical-instant arrival sequence achieves the corresponding response time. We must show that the smallest fixed point $t_i^*(\ell)$ for

$W_i^{(\ell)}$ is the maximum response time to the call CEILCHANGE(R_j, ℓ) from t_{ij} and that the critical-instant arrival sequence will result in a $t_i^*(\ell)$ response time.

We first show that the critical-instant arrival sequence results in a response time of $t_i^*(\ell)$ to the call CEILCHANGE(R_j, ℓ) from time t_{ij}. By the inductive hypothesis, the response times of calls CEILCHANGE$(R_j, \Pi(R_j)-1)$, …, CEILCHANGE$(R_j, \ell+1)$ are $t_i^*(\Pi(R_j)-1), \ldots, t_i^*(\ell+1)$ under the critical-instant arrival sequence. Thus, by observation (4), for each $\tau_k \in \{\tau_{\Pi(R_j)-1}, \ldots, \tau_{\ell+1}\}$, the execution requests of τ_k in the interval $(t_{ij}, t_{ij}+t)$ for $t \le t_i^*(k)$ approach RBF(τ_k, τ_i, t); when $t > t_i^*(k)$, the execution requests remains at RBF$(\tau_k, \tau_i, t_i^*(k))$. Clearly, the remaining tasks $\tau_{k'} \in \{\tau_\ell, \ldots, \tau_1\}$ have execution requests approaching RBF$(\tau_{k'}, \tau_i, t)$ under the critical-instant arrival sequence in the interval $(t_{ij}, t_{ij}+t)$, since the command CEILCHANGE(R_j, k') has not been issued by time $t_{ij}+t$. By observation (2), τ_i has execution requirement of $S_{ij} - X_\ell$ from the lock of the resource to the call to CEILCHANGE(R_j, ℓ). Therefore, $W_i^{(\ell)}(t)$ exactly describes the cumulative execution requirements of tasks $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ and task τ_i under the critical-instant arrival sequence for τ_i's lock of resource R_j. The smallest fixed point of $W_i^{(\ell)}$ is the first time-instant at which the processor can complete $S_{ij} - X_\ell$ units of τ_i's critical section and all the execution requests of tasks $\tau_1, \ldots, \tau_{\Pi(R_j)-1}$ that can preempt τ_i's critical section. Thus, the CeilChange(R_j, ℓ) command will be executed at time $t_{ij} + t_i^*(\ell)$ in the critical-instant arrival sequence.

We now show that $t_i^*(\ell)$ represents the "worst-case" response time to the call CEILCHANGE(R_j, ℓ) for task τ_i. To show this statement, the reasoning closely follows the proof of Theorem 1. If any task $\tau_k \in \{\tau_1, \ldots, \tau_{\Pi(R_j)-1}\}$ releases its first job later than $t_{ij} + \epsilon$ (where ϵ approaches zero), then the amount of execution requests that τ_i can contribute to the interval $(t_{ij}, t_{ij} + t_i^*(\ell))$ cannot increase. Moving the activation of the first job of a task of τ_k before t_{ij} would either contradict the property that τ_k did not have an active job at time t_{ij} (Step 3), or would result in a lower interference (if the job completed its execution at time t_{ij}). Therefore, under the arrival sequence where all tasks except τ_k release jobs according to the critical-instant arrival scenario, the interference in the interval $(t_{ij}, t_{ij} + t_i^*(\ell))$ is at most $W_i^{(\ell)}(t_i^*(\ell))$. We have just shown that any deviation from the critical-instant arrival sequence results in a lower or equal response time. By inductively applying this same logic, it can be shown that any other tasks deviating from the critical-instant arrival sequence will also not increase the response time to call CeilChange(R_j, ℓ). The maximum response time is, therefore, $t_i^*(\ell)$. □

By the previous lemma, the maximum response time to the final ceiling-change call in τ_i's critical section for R_j (i.e., CEILCHANGE$(R_j, 1)$) is the smallest fixed point for function $W_i^{(1)}(t)$ (i.e., $t_i^*(1)$). After the call to CEILCHANGE$(R_j, 1)$, the dynamic ceiling, $\Pi^D(R_j)$, is set to one; the implication is that, after time $t_{ij} + t_i^*(1)$, τ_i may execute its critical section non-preemptively until it releases shared resource R_j. By observation (2), the amount of non-preemptive execution after $t_{ij} + t_i^*(1)$ is $S_{ij} - X_1$. The next theorem below immediately follows from the previous statements.

Theorem 3 *The maximum resource-holding time,* RHT(R_j, τ_i), *of resource* R_j *by task* τ_i *under the modified protocol is equal* $t_i^*(1) + S_{ij} - X_1$.

6 Conclusion

Recent developments in two emerging areas of real-time systems—open environments and multicore platforms—indicate that it is important to minimize the amount of time that individual task systems keep specific resources locked (thereby denying other independent, co-located task systems access 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 task systems that can be modeled 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, our technique is general enough that it may be adapted, with minimal changes, to other scheduling frameworks (such as, for example, deadline-monotonic (DM) scheduling with resource arbitration done according to the priority ceiling protocol (PCP), as proved in Bertogna et al. 2007).
- 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, our technique is easily adapted to apply to systems scheduled using other scheduling frameworks (such as DM + SRP Bertogna et al. 2007).
- We have considered the problem of RHT-minimization of resource-sharing sporadic task systems when we are not constrained to using the EDF + SRP framework. To this end, we have modified SRP to come up with a more general resource access protocol, and have presented and proved the correctness of an algorithm for further decreasing RHT's when a system is scheduled using this modified resource access protocol in conjunction with EDF. Furthermore, we have described how to compute RHT's for the modified protocol.

As a future work, we intend to show how the techniques here described can be efficiently applied in the context of open environments (in the model described in Fisher et al. (2007a, 2007b)), as well as in the multiprocessor domain. Reducing the amount of time for which a re source can be held will allow to overcome the major limitations that these systems have when global resources can be shared among different task sets.

Acknowledgements We are extremely grateful to the anonymous reviewers of both the conference and journal versions of this paper for their help in improving the quality of the paper. In particular, we are grateful to the reviewer who suggested an improvement in the resource-hold time calculations of Sect. 3.

References

Baker TP (1991) Stack-based scheduling of real-time processes. Real-Time Syst Int J Time-Crit Comput 3(1):67–100

Baruah S (2006) Resource sharing in EDF-scheduled systems: A closer look. In: Proceedings of the IEEE real-time systems symposium, Rio de Janeiro, Brazil, December 2006. IEEE Comput Soc, Los Alamitos, pp 379–387

Baruah S, Mok A, Rosier L (1990) Preemptively scheduling hard-real-time sporadic tasks on one processor. In: Proceedings of the 11th real-time systems symposium, Orlando, Florida, 1990. IEEE Comput Soc, Los Alamitos, pp 182–190

Behnam M, Shin I, Nolte T, Nolin M (2006) Real-time subsystem integration in the presence of shared resource. In: Proceedings of the real-time systems symposium—work-in-progress session, Rio de Janerio, Brazil, December 2006, pp 9–12

Behnam M, Shin I, Nolte T, Nolin M (2007) SIRAP: a synchronization protocol for hierarchical resource sharing in real-time open systems. In: EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software, New York, NY, USA, October 2007. Assoc Comput Mach, New York, pp 279–288

Bertogna M, Fisher N, Baruah S (2007) Resource-locking durations in static-priority systems. In: Proceedings of the workshop on parallel and distributed real-time systems, Long Beach, California, April 2007. IEEE Comput Soc, Los Alamitos, pp 1–8,

Davis RI, Burns A (2005) Hierarchical fixed priority pre-emptive scheduling. In: Proceedings of the IEEE real-time systems symposium, Miami, Florida, IEEE Comput Soc, Los Alamitos, pp 389–398

Davis RI, Burns A (2006) Resource sharing in hierarchical fixed priority pre-emptive systems. In: Proceedings of the IEEE real-time systems symposium, Rio de Janeiro, Brazil, December 2006. IEEE Comput Soc, Los Alamitos, pp 257–267

Deng Z, Liu J (1997) Scheduling real-time applications in an open environment. In: Proceedings of the eighteenth real-time systems symposium, San Francisco, California, December 1997. IEEE Comput Soc, Los Alamitos, pp 308–319

Dertouzos M (1974) Control robotics: the procedural control of physical processors. In: Proceedings of the IFIP congress, pp 807–813

Feng XA, Mok A (2002) A model of hierarchical real-time virtual resources. In: Proceedings of the IEEE real-time systems symposium, Austin, Texas, December 2002. IEEE Comput Soc, Los Alamitos, pp 26–35

Fisher N, Bertogna M, Baruah S (2007a) The design of an EDF-scheduled resource-sharing open environment. In: Proceedings of the IEEE real-time systems symposium, Tucson, Arizona, December 2007. IEEE Comput Soc, Los Alamitos, pp 83–92

Fisher N, Bertogna M, Baruah S (2007b) The design of an EDF-scheduled resource-sharing open environment. Technical report, Department of Computer Science, The University of North Carolina at Chapel Hill. Available at http://www.cs.unc.edu/~fishern/pubs.html

Fisher N, Bertogna M, Baruah S (2007c) Resource-locking durations in EDF-scheduled systems. In: Proceedings of the 13th IEEE real-time and embedded technology and applications symposium, Bellevue, Washington, April 2007. IEEE Comput Soc, Los Alamitos

Gai P, Lipari G, di Natale M (2001) Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In: Proceedings of the IEEE real-time systems symposium, London, England, December 2001. IEEE Comput Soc, Los Alamitos, pp 73–83

Ghattas R, Dean AG (2007) Preemption threshold scheduling: Stack optimality, enhancements and analysis. In: Proceedings of the IEEE real-time technology and applications symposium (RTAS), Bellevue, Washington, April 2007. IEEE Comput Soc, Los Alamitos, pp 147–157

Henzinger TA, Kirsch CM, Matic S (2003) Schedule carrying code. In: Alur R, Lee I (eds), Embedded software, third international conference, EMSOFT 2003, Philadelphia, PA, USA, October 13–15, 2003. Lecture Notes in Computer Science, vol 2855, Springer, Berlin, pp 241–256

Joseph M, Pandya P (1986) Finding response times in a real-time system. Comput J 29(5):390–395

Kuo T-W, Li C-H (1999) A fixed priority driven open environment for real-time applications. In: Proceedings of the IEEE real-time systems symposium, Phoenix, Arizona, December 1999. IEEE Comput Soc, Los Alamitos, pp 256–267

Lehoczky J, Sha L, Ding Y (1989) The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In: Proceedings of the real-time systems symposium, Santa Monica, California, USA, Dec. 1989. IEEE Comput Soc, Los Alamitos, pp 166–171

Lipari G, Bini E (2003) Resource partitioning among real-time applications. In: Proceedings of the EuroMicro Conference on real-time systems, Porto, Portugal, July 2003. IEEE Comput Soc, Los Alamitos, pp 151–160

Lipari G, Buttazzo G (2000) Schedulability analysis of periodic and aperiodic tasks with resource constraints. J Systems Archit 46(4):327–338

Liu C, Layland J (1973) Scheduling algorithms for multiprogramming in a hard real-time environment. J ACM 20(1):46–61

Mok AK (1983) Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology. Available as Technical Report No MIT/LCS/TR-297

Pellizzoni R, Lipari G (2005) Feasibility analysis of real-time periodic tasks with offsets. Real-Time Syst Int J Time-Crit Comput, 30(1–2):105–128

Saewong S, Rajkumar R, Lehoczky JP, Klein MH (2002) Analysis of hierarchical fixed-priority scheduling. In: Proceedings of the EuroMicro conference on real-time systems, Vienna, Austria, June 2002. IEEE Comput Soc, Los Alamitos, pp 173–181

Saksena M, Wang Y (2000) Scalable real-time system design using preemption thresholds. In Proceedings of the IEEE real-time systems symposium, Los Alamitos, California, November 2000. IEEE Comput Soc, Los Alamitos, pp 256–267

Sha L, Rajkumar R, Lehoczky JP (1990) Priority inheritance protocols: An approach to real-time synchronization. IEEE Trans Comput 39(9):1175–1185

Shin I, Lee I (2003) Periodic resource model for compositional real-time guarantees. In: Proceedings of the IEEE real-time systems symposium, Cancun, Mexico, December 2003. IEEE Comput Soc, Los Alamitos, pp 2–13

Sprunt B, Sha L, Lehoczky JP (1989) Aperiodic task scheduling for hard real-time systems. Real-Time Syst 1:27–69

Spuri M, Buttazzo G (1996) Scheduling aperiodic tasks in dynamic priority systems. Real-Time Syst Int J Time-Crit Comput 10(2):179–210

Marko Bertogna is currently completing his Ph.D. in Computer Science at the Scuola Superiore S. Anna. He graduated magna cum laude in Telecommunication Engineering at the University of Bologna in 2002. During 2001 he was visiting student at the Technical University of Delft (Holland). His main research interests include Real-Time Operating Systems, reconfigurable devices and the scheduling analysis for multiprocessor platforms and FPGA.

Nathan Fisher is an Assistant Professor in the Department of Computer Science at Wayne State University. He received his Ph.D. from the University of North Carolina at Chapel Hill in 2007, his M.S. degree from Columbia University in 2002, and his B.S. degree from the University of Minnesota in 1999, all in computer science. His research interests are in real-time and embedded computer systems, parallel and distributed algorithms, resource allocation, and approximation algorithms. His current research focus is on multiprocessor scheduling theory and composability of real-time applications.

Sanjoy Baruah is a professor in the Department of Computer Science at the University of North Carolina at Chapel Hill. He received his Ph.D. from the University of Texas at Austin in 1993. His research and teaching interests are in scheduling theory, real-time and safety-critical system design, and resource-allocation and sharing in distributed computing environments.