The SRP Resource Sharing Protocol for Self-Suspending Tasks

Geoffrey Nelissen† and Alessandro Biondi*
†CISTER, ISEP, Polytechnic Institute of Porto, Portugal
*Scuola Superiore Sant'Anna, Pisa, Italy

***Abstract*—Motivated by the increasingly wide adoption of real-time workload with self-suspending behaviors, and the relevance of mechanisms to handle mutually-exclusive shared resources, this paper takes a new look at locking protocols for self-suspending tasks under uniprocessor fixed-priority scheduling. Pitfalls when integrating the widely-adopted Stack Resource Policy (SRP) with self-suspending tasks are firstly illustrated, and then a new fine-grained SRP analysis is presented. Next, a new locking protocol, named SRP-SS, is proposed to overcome the limitations of the original SRP. The SRP-SS is a generalization of the SRP to cope with the specificities of self-suspending tasks. It therefore reduces to the SRP under some configurations and hence theoretically dominates the SRP. It also ensures backward compatibility for applications developed specifically for the SRP. The SRP-SS comes with its own schedulability analysis and configuration algorithm. The performances of the SRP and SRP-SS are finally studied by means of large-scale schedulability experiments.**

I. INTRODUCTION

Self-suspending tasks are tasks that suspend their execution to: synchronize with other tasks running on the same or other cores by means of semaphores or waiting barriers; wait for data produced by other tasks, co-processors or obtained through I/O devices; wait for timing events or external interrupts; and/or synchronize accesses to hardware shared resources, citing but a few examples. Clearly, self-suspending tasks model a wide variety of execution behaviors witnessed in actual applications. However, as evidenced by a string of misconceptions that propagated in the state-of-the-art on real-time scheduling [1]–[3], the analysis of self-suspending tasks cannot be performed with standard techniques conceived for regular sporadic tasks. The analysis of self-suspending tasks poses significant challenges in the derivation of safe, yet tight, response-time bounds.

A common feature implemented in real-time embedded systems is the protection of so-called *critical sections* by means of locking protocols. Those critical sections may be segments of code that must execute non-preemptively for performance or safety reasons, or they may encapsulate read or write operations on shared software or hardware resources. In the latter case, the lock protecting the critical section prevents other tasks to modify the same resource at the same time in an undeterministic manner. The *Stack Resource Policy* (SRP) [4] is one of the most widespread locking protocols for uniprocessor real-time system, also used as a building block to develop influential locking protocols for multiprocessors (e.g., the MSRP [5]). In this work we show that, when scheduling a set of self-suspending tasks, the SRP may lead to large blocking times, losing its core property that guarantees that a task can be blocked by at most one critical section. This strongly penalizes the system schedulability and affects the system robustness in the presence of CPU overloads. Additionally, since it was not conceived to handle self-suspending tasks, the original SRP analysis fails to account for the extra blocking that a self-suspending task may suffer. Therefore, there is a need for **(i)** devising a new analysis for SRP that considers self-suspensions, and **(ii)** developing a new resource sharing protocol that is better suited to self-suspending tasks.

Note that studying self-suspending real-time tasks in the presence of resource sharing has also a high industrial relevance. Indeed, the AUTOSAR [6] automotive standard mandates the use of the SRP[1] to regulate the access to mutually-exclusive shared resources. Furthermore, it supports task self-suspensions by means of the AUTOSAR *events* mechanism that allow tasks to suspend their execution until a specific event (be it a timing event, a data reception event, or any external or internally defined event) is triggered. These two standardized mechanisms are likely combined in realistic applications, e.g., [8]. For instance, automotive software developers are now facing parallel workload with precedence constraints [9] (typically originated by data causality) that execute on multicore platforms. Synchronization between tasks and cores is performed by means of waiting barriers, which force the waiting task(s) to self-suspend [10]. Local resource accesses are themselves protected with the SRP.

Paper contributions. Motivated by the need of a better understanding of SRP-based locking in the presence of self-suspending tasks, this paper makes the following contributions:

- After showing that existing results related to the SRP are not compatible with self-suspending tasks, we present a *schedulability analysis* for a set of self-suspending tasks scheduled with a task level fixed priority scheduling policy and that share resources protected by the SRP.
- We propose a new locking protocol, named SRP-SS, that generalizes and hence dominates the SRP. The SRP-SS improves the schedulability of self-suspending tasks accessing locks.
- We present a schedulability analysis for the SRP-SS.
- We propose a technique for configuring the SRP-SS with the aim of enhancing the system schedulability.

Experimental results are also presented to assess the performance of **(i)** the proposed schedulability analyses, and **(ii)** the new locking protocol SRP-SS and its configuration algorithm.

II. SYSTEM MODEL

This paper considers the problem of scheduling a set $\Gamma = \{\tau_1, \dots, \tau_n\}$ of n sporadic real-time tasks upon a single processor. Each task τ_i is characterized by a *worst-case*

---

[1] More specifically, it mandates the use of the Immediate Priority Ceiling protocol [7], which is equivalent to the SRP for the considered setting (i.e., fixed priority scheduling on a single core platform) [4].

execution time (WCET) C_i, a *minimum inter-arrival time* T_i, and a *relative deadline* $D_i \leq T_i$. Tasks release an infinite sequence of *jobs*. A task τ_i is said to be *active* at time t if a job of τ_i started executing at or before t and did not yet complete its execution at time t.

Tasks are scheduled according to *task level fixed-priority scheduling*, where each task τ_i has a unique priority $\pi_i \geq 1$ (larger values indicate higher priorities). We denote the set of tasks with higher priority than τ_i as $hp(i) \subset \Gamma$. Analogously, $lp(i) \subset \Gamma$ denotes be set of tasks with lower priority than τ_i. For simplicity, we assume that no two tasks have the same priority. However, the results of this paper can easily be extended to the case where more than one task share the same priority by adding one term in the response time equation to account for the interference that same-priority tasks generate on each other.

Tasks can *self-suspend* their execution, e.g., due to I/O operations or to use hardware accelerators [11]. The total suspension time a job of τ_i is upper-bounded by S_i. No information is assumed about the actual task structure, therefore suspensions can occur at any time during the execution of a job. In the related literature, the model described above is known as the *dynamic self-suspending task model* [12]. The only restriction we pose on the tasks' dynamic self-suspension behavior is that tasks cannot self-suspend within critical sections[2].

The tasks share a set of n_r single-unit resources $Q = \{\ell_1, \ldots, \ell_{n_r}\}$. Each resource must be accessed in mutual exclusion. Each job of task τ_i accesses resource ℓ_k at most $N_{i,k}$ times by means of *critical sections* of a duration upper-bounded by $L_{i,k}$. If a task τ_i does not access a resource ℓ_k, then $L_{i,k} = N_{i,k} = 0$. This work assumes that critical sections cannot be nested[3].

This work considers two different resource sharing policies. Section VI analyzes the case where the access to shared resources is regulated by the popular and widely implemented *stack resource policy* (SRP) [4]. Then, a new variant of SRP (called SRP-SS) specifically designed to handle self-suspending is proposed and detailed in Section VII.

III. BACKGROUND

A. The Stack Resource Protocol

According to the SRP, each resource ℓ_k is assigned a *resource ceiling* $\pi(\ell_k)$. Whenever a task locks a resource ℓ_k, a system-wide parameter Π, called *system ceiling*, is raised to $\pi(\ell_k)$. The system ceiling Π is then restored to its previous value when the currently executing task releases a resource.

The SRP also modifies the *preemption rule* of standard fixed-priority scheduling:

SRP preemption rule – a task τ_j can preempt the execution of another task τ_i if $\pi_j > \pi_i$ *and* $\pi_j > \Pi$.

Formally, when the SRP is used together with a fixed job priority scheduling algorithm, it can be implemented using three queues: the *Ready Queue* Q_r, the *Blocked Queue* Q_b and the *Suspended Queue* Q_{ss}. At any time instant, the scheduler executes the highest priority job in the ready queue Q_r. The content of Q_r is updated whenever scheduling event happens as described in Algorithm 4 reported in Appendix A.

The SRP is typically implemented as a stack (hence its name). Whenever a resource is locked, its ceiling priority is pushed on top of the stack (Line 27 in Algorithm 4), and whenever a resource is freed, its ceiling priority is removed from the top of the stack (Line 32). The system ceiling Π is always equal to the resource ceiling priority on top of the stack (Lines 28 and 33). The preemption rule stated above is then implemented by updating the tasks in the ready queue at Lines 5 to 9 and Lines 34 to 37.

SRP's timing analysis and the configuration of resource ceilings are briefly reviewed in Section III-B. Note that the original analysis of the SRP assumes that tasks do not self-suspend. We address this limitation in Section VI.

B. Original Analysis for the SRP

The analysis of blocking times introduced by the SRP was originally derived for regular periodic/sporadic tasks (i.e., without self-suspensions) [4]. As it will be discussed in Section IV, the original analysis for the SRP is not valid for self-suspending tasks. Nevertheless, that analysis is recalled in this section to make the paper self-sufficient.

In his seminal paper, Baker [4] showed that when single-unit resources are protected by the SRP, a job can be blocked at most once, and this blocking is due to a single critical section accessed by a lower priority task. At the analysis stage, this phenomenon is reflected as a blocking time B_i for each task τ_i, which accounts for the largest critical section related to a resource (i) accessed by a lower priority task, and (ii) whose ceiling can prevent τ_i to execute, that is

$$B_i = \max_{\tau_j \in lp(i), \ell_k \in Q} \{L_{j,k} \mid \pi(\ell_k) \geq \pi_i\}_0 \tag{1}$$

where $\max\{\cdot\}_0 = 0$ if the set on which the max operation is applied is empty.

For *non-self-suspending* tasks, the worst-case response time (WCRT) of a task τ_i is then given by the smallest positive fixed-point solution to the following recursive equation.

$$R_i = B_i + C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j. \tag{2}$$

Another key property of the SRP is that blocking time is incurred at the release of a job, and not when attempting to lock a resource: for this reason, the blocking factor B_i is also known as *arrival blocking*.

Finally, it is worth mentioning that the SRP comes with a rule for configuring the resource ceilings, which mandates to assign ceilings with the highest priority among the tasks that use that resource, i.e., $\pi(\ell_k) = \max_{\tau_i \in \Gamma}\{\pi_i \mid N_{i,k} > 0\}$.

---

[2] This restriction could be lifted using similar techniques than Brandenburg in Appendix F of the extended version of [13].

[3] In practice, nested resources could be handled using group-locks (as in SRP). It may be possible to consider finer-grained nesting, but further investigation would be required to understand the impact on blocking.

C. Response time analysis of self-suspending tasks

Several analyses have been developed over the years to compute the WCRT of dynamic self-suspending tasks. We recall three of them.

Suspension oblivious analysis. This approach [12] models suspension time as execution time. Each task τ_i is replaced by an artificial task τ_i' with WCET $C_i' = C_i + S_i$ and suspension time $S_i' = 0$. The usual response time analysis (RTA) [14] for sequential tasks may then be used, that is,

$$R_i = (C_i + S_i) + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (C_j + S_j). \tag{3}$$

The suspension oblivious analysis is the simplest but also the most pessimistic analysis for self-suspending tasks.

Blocking-based analysis. In her book [15], Liu models the extra interference suffered by tasks due to self-suspension by introducing an artificial blocking term G_i in the RTA. More specificaly, it was stated in [15] and [16] and then proven in [12] that:

Lemma 1. *The worst-case contribution of a self-suspending task τ_j to the interference suffered by a lower priority task τ_i in an interval of length t is upper-bounded by*

$$\min\{C_j, S_j\} + \left\lceil \frac{t}{T_j} \right\rceil C_j.$$

Summing the contribution of every higher priority task, the WCRT of a task τ_i is then given by the smallest positive fixed point solution to

$$R_i = G_i + C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{4}$$

where $G_i = S_i + \sum_{\tau_j \in hp(i)} \min\{C_j, S_j\}$.

Note that despite having been published in a book in 2000 [15] and used in [16] in 1988, there was no proof of correctness for this analysis until [17] and [12] were published in 2016.

Jitter-based analysis. This analysis models suspension related interference as a jitter term in the RTA [3]. Specifically,

Lemma 2. *The worst-case contribution of a self-suspending task τ_j to the interference suffered by a lower priority task τ_i in an interval of length t is upper-bounded by*

$$\left\lceil \frac{t + J_j}{T_j} \right\rceil C_j$$

where $J_j = \overline{R}_j - C_j$ and $\overline{R}_j$ is a safe upper-bound on the WCRT of τ_j.

Then, the WCRT of a task τ_i is given by the least positive fixed point of the following equation.

$$R_i = (C_i + S_i) + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j. \tag{5}$$

A hybrid approach has been presented in [12]. It combines ideas of Liu's work and the jitter-based analysis. It was proven that the analysis in [12] dominates the three analyses presented above. However, we do not discuss it here as it is much more complex than the other three and the focus of this paper is rather on resource sharing policies than the RTA of dynamic self-suspending tasks.

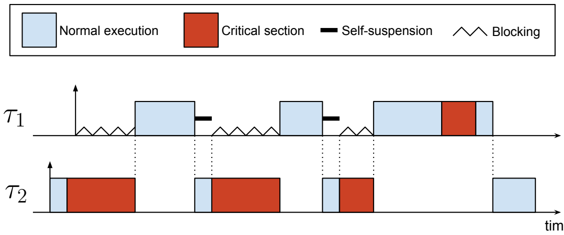


Figure 1. Scheduling example of two self-suspending tasks sharing a resource protected with the SRP. Up-arrows denote the release of a job.

All the analyses mentioned above are compatible with the resource sharing policies discussed in this paper. Yet, for the sake of conciseness, the jitter-based analysis is assumed hereafter[4].

IV. MOTIVATIONAL EXAMPLE

The impact of self-suspensions on blocking time can be easily explained with an example. Consider two tasks τ_1 and τ_2, with $\pi_1 > \pi_2$, both sharing a resource ℓ with ceiling $\pi(\ell) = \pi_1$. Suppose also that τ_2 includes at least three critical sections to access ℓ. Now, consider the schedule of Figure 1, where τ_1 self-suspends *two* times for a total of $S_i = 2$ time units. As shown in Figure 1, contrary to what is assumed in the analysis of the SRP for non-self-suspending tasks, a single job of τ_1 can be blocked more than once by the lower-priority task τ_2. Specifically, it can be blocked **(i)** when it is released, which is the only scenario considered in the original analysis of the SRP, and **(ii)** at the end of each of its self-suspensions. As seen in Figure 1, whenever τ_1 self-suspends the lower-priority task τ_2 can execute, lock a resource, and contextually raise the system ceiling to a point (in this case $\Pi = \pi_1$) that forbids the execution of τ_1 when it completes its self-suspension.

From the example above, it becomes clear that the amount of blocking a task can suffer is *strictly* dependent on the number of times the task self-suspends. If such a number is not available, a safe analysis must assume that the number of self-suspensions is as large as possible (virtually infinite), with the consequence that in the worst-case, *every* conflicting critical section executed by a lower-priority task can lead to blocking every higher priority task, i.e., a task may always self-suspend and then resume its execution an arbitrary small amount of time after a low-priority task locked a resource. Clearly, this approach would be too pessimistic to be useful.

V. EXTENDING THE SELF-SUSPENDING TASK MODEL

The discussion in the previous section suggests that the dynamic self-suspending task model lacks information for an accurate analysis when resources are shared with the SRP.

---

[4] According to our experiments, there is almost no difference in terms of schedulability performance between (4) and (5) when they are combined with the resource sharing blocking analysis. The suspension oblivious analysis, i.e., (3), always performs worse than (4) and (5), with very few task sets detected as being schedulable.

Similar to what was recently proposed in [18] and [19], we overcome this limitation introducing a new task parameter, namely, the *maximum number of times* a task can self-suspend. We denote that maximum number of suspensions by the task parameter X_i. Therefore, in this new model, the self-suspension behaviour of each task τ_i is characterized by its maximum suspension time S_i and the maximum number of suspensions X_i over which suspension time may be spent.

Note that the value of X_i can easily be extracted from the task's source code by identifying and counting the maximum number of events that may trigger a self-suspension (e.g., specific system calls) on a single execution path. In the presence of mutually-exclusive execution paths (e.g., due to conditional statements), the code analysis must keep track of the paths with the largest number of suspensions. Such analysis requires at most one pass through the code of each task.

For the sake of completeness, it is worth mentioning that prior work already considered another self-suspending task model that transitively included such an information. It is the case of the (so called) *segmented* self-suspending task model [1]. This model explicitly accounts for a structure where the task alternates execution phases with suspension phases, where each of them is characterized by a WCET or maximum suspension time. However, the analysis of the segmented self-suspending task model has been shown [1] to suffer of a considerable conceptual complexity due to a number of non-trivial scheduling phenomena. To date, a precise analysis for such a model is only enabled by means of mixed-integer linear programming [1] or iterative search [20]. Both techniques have large run-time and do not scale well. Furthermore, the derivation of a safe segmented self-suspending task model in the presence of multi-path code is far from being obvious as discussed in [21]. These facts led us to stick with the dynamic self-suspending task model instead, which can benefit of a compatibility with classical and efficient response-time analysis techniques (provided that a careful estimation of the interference is adopted) as reviewed in Section III-C.

VI. ANALYSIS OF SELF-SUSPENDING TASKS UNDER THE SRP

We can infer from the observations made in Section IV that the original SRP analysis cannot be applied to self-suspending tasks as it would lead to optimistic response time bounds. For instance, consider the example of Figure 1: in this case, task τ_1 is blocked three times, while the original SRP analysis (see Section III-B) would account for a single blocking event at the job release. Therefore, in this section, we propose a new analysis that can cope with self-suspending tasks.

A. Simple blocking analysis

First, we formalize the property that was intuitively introduced in Section V, namely, that under the SRP, a self-suspending task can incur blocking multiple times.

Lemma 3. *Under the SRP, a job of a self-suspending task τ_i can be blocked at most $X_i + 1$ times. Each blocking can be caused by a single critical section.*

Proof. As it has been recalled in Section III-B, under the SRP a task can be blocked only when it attempts to preempt a lower-priority task that is holding a shared resource with a conflicting ceiling. Since, under task level fixed priority scheduling, a preemption may occur only when an execution segment is released, i.e., at the beginning of a job or after a self-suspension, and because τ_i comprises at most X_i self-suspensions, the task can be blocked at most $X_i + 1$ times.

According to the SRP preemption rule, a task τ_i with a pending preemption that is blocked by a lower-priority task τ_j can proceed to execute as soon as τ_j releases the shared resource that generates the blocking. Since there are no nested critical sections, each blocking event is related to a single critical section. The lemma follows. □

From Lemma 3, a blocking bound can be derived by accounting for the largest critical section that can block τ_i multiplied by the number of times the task can be blocked.

Theorem 1. *Under the SRP, the maximum blocking time incurred by a job of a self-suspending task τ_i is bounded by*

$$B_i = (X_i + 1) \times \max_{\tau_j \in lp(i), \ell_k \in Q} \{L_{j,k} \mid \pi(\ell_k) \geq \pi_i\}. \qquad (6)$$

Proof. Under the SRP, a task τ_i can be blocked only by critical sections of lower-priority tasks whose corresponding resource ℓ_k has a conflicting ceiling with τ_i, i.e., $\pi(\ell_k) \geq \pi_i$. Consequently, $\max_{\tau_j \in lp(i), \ell_k \in Q}\{L_{j,k} \mid \pi(\ell_k) \geq \pi_i\}$ upper-bounds the length of any critical section that can block τ_i. The theorem follows after recalling Lemma 3. □

Given the task set parameters, the blocking term provided by the above theorem is a constant, and therefore can seamlessly be integrated at the stage of response-time analysis to obtain a safe schedulability test. However, this approach can clearly be very pessimistic. For instance, in the presence of one low priority task τ_j with a single, but very large critical section $\mathcal{C}$, this approach would always consider the case where $\mathcal{C}$ blocks each job J_i of τ_i $(X_i + 1)$ times, even though this may be totally impossible in practice, i.e., independently of the actual number of jobs of τ_j that can overlap with the execution of J_i. For this reason, a more accurate blocking analysis is developed in the following section.

B. More accurate blocking analysis

The analysis proposed in this section is built upon the following three steps: **(i)** *explicitly* identify all critical sections that can possibly overlap with a given time interval; **(ii)** select the largest $(X_i + 1)$ critical sections within the interval of interest; **(iii)** perform (i) and (ii) concurrently with the RTA of all tasks in Γ.

As expressed by (i), we first identify all critical sections that can execute in a time interval of length t and potentially block the execution of task τ_i. Let $\mathcal{CS}_i(t)$ be the multiset containing all those critical sections, and $\Delta_i(t)$ the multiset containing their worst-case duration. Finally, assume a vector $\overline{\mathbf{R}}$ of safe response-time bounds for all tasks in Γ is given, Lemma 4 explains how $\Delta_i(t)$ may be built.

Lemma 4. *The worst-case durations of the critical sections that can block τ_i in an interval of length t are included into* *the multiset $\Delta_i(t, \overline{\mathbf{R}})$ defined as follows*[5]

$$\Delta_i(t, \overline{\mathbf{R}}) = \biguplus_{\tau_j \in lp(i)} \biguplus_{\ell_k \in Q} \{L_{j,k} \mid \pi(\ell_k) \geq \pi_i\} \otimes \big(N_{j,k} \times \eta_j(t, \overline{\mathbf{R}})\big)$$

where

$$\eta_j(t, \overline{\mathbf{R}}) = \left\lceil \frac{t + \overline{R}_j}{T_j} \right\rceil$$

and $\overline{R}_j$ is the component of $\overline{\mathbf{R}}$ for τ_j (i.e., an upper-bound on τ_j's response time).

Proof. Under the SRP, task τ_i can be blocked only by critical sections of a lower-priority task τ_j (iterated over with the first multiset union) related to a resource ℓ_k with ceiling $\pi(\ell_k) \geq \pi_i$ (iterated over with the second multiset union). Furthermore, τ_j may execute at most $\eta_j(t, \overline{\mathbf{R}})$ different jobs in any time interval of length t [22]. By definition of $N_{j,k}$, each job of τ_j has at most $N_{j,k}$ critical sections related to resource ℓ_k. Hence, for each pair (τ_j, ℓ_k), there are at most $N_{j,k} \times \eta_j(t, \overline{\mathbf{R}})$ critical sections with a worst-case duration $L_{j,k}$ that can block τ_i. □

Following principle (ii) and building upon the above lemma, it is finally possible to derive a tighter blocking bound for self-suspending tasks under the SRP. To simplify the presentation of the following result, the notation $\Sigma(x, \mathcal{S})$ is introduced to denote the sum of the x *largest* elements of a multiset $\mathcal{S}$. If the size of the multiset $\mathcal{S}$ is smaller than x, then $\Sigma(x, \mathcal{S})$ returns the sum of all elements in $\mathcal{S}$.

Theorem 2. *Consider a self-suspending task τ_i. Under the SRP, the maximum blocking time incurred by a job of τ_i during an interval of length t is bounded by*

$$B_i(t, \overline{\mathbf{R}}) = \Sigma\left(X_i + 1, \Delta_i(t, \overline{\mathbf{R}})\right).$$

Proof. By Lemma 3, a job of τ_i can be blocked by at most $X_i + 1$ critical sections. By Lemma 4, the worst-case duration of all the critical sections that can block a job of τ_i during an interval of length t are included into $\Delta_i(t, \overline{\mathbf{R}})$. Hence, the sum of the largest $X_i + 1$ elements in $\Delta_i(t, \overline{\mathbf{R}})$ yields a safe blocking bound for τ_i. □

The next section explains how to exploit the result of Theorem 2 to implement a safe schedulability test for the considered task model.

C. Analysis algorithm

We first extend the results of Section III-C to cope with the blocking bound derived in Theorem 2.

Theorem 3. *Let R_i be the least positive fixed-point of the following recursive equation (if it exists)*

$$R_i = (C_i + S_i) + B_i(R_i, \overline{\mathbf{R}}) + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i + \overline{R}_j - C_j}{T_j} \right\rceil C_j. \tag{7}$$

If $R_i \leq D_i$, then R_i is a safe upper-bound on τ_i's worst-case response time and τ_i is schedulable .

Proof. The response time of task τ_i is composed of three terms, **(1)** τ_i's execution and suspension time, **(2)** the amount of time τ_i may be blocked by lower priority tasks due to the SRP, and **(3)** the amount of time τ_i's execution is interfered by a higher-priority task.

1) If τ_i's response time is upper-bounded by D_i (i.e., $R_i \leq D_i$), then term (1) is upper-bounded by the sum of τ_i's WCET and worst-case suspension time, i.e., $C_i + S_i$.
2) Theorem 2 bounds term (2), i.e., during an interval of length R_i, τ_i can be blocked for at most $B_i(R_i, \overline{\mathbf{R}})$ time units.
3) Finally, it was proven in [3] that Lemma 2 yields a safe bound on the higher priority interference incurred by τ_i during an interval of length R_i.

Therefore, summing the three bounds discussed above, we get Equation 7, and the existence of a fixed-point $R_i \leq D_i$ (remember the assumption at point 1)) for Equation (7) implies that R_i is a safe response-time bound for τ_i. □

With the above theorem in place, it is possible to derive an algorithm that allows checking the system schedulability. The major issue with Theorem 3 is that it takes as input the vector $\overline{\mathbf{R}}$ of safe response-time bounds for all tasks, which clearly introduces a sort of circular dependency in Equation (7), i.e., it requires the response-time of all tasks to obtain the response-time of each task. This issue can be solved by adopting a simple iterative scheme, as reported in Algorithm 1.

The idea is to construct a *non-increasing* sequence of safe response-time bounds by starting from initializing $\overline{\mathbf{R}}$ with the tasks' deadlines, i.e., $\forall \tau_i \in \Gamma$, $\overline{R}_i = D_i$ (line 2 of Algorithm 1). This choice is supported by the following rationale. Suppose to dispose of a run-time mechanism $\mathcal{M}$ that aborts a job that did not complete by its deadline: in this way, each job of each task τ_i is guaranteed to terminate within D_i time units, and hence the latter constitutes a valid response-time bound to configure $\overline{\mathbf{R}}$. Then, if Theorem 3 yields a response-time bound for each task that is lower than or equal to the corresponding deadlines (line 13), then it means that no deadlines is ever violated. Consequently, the run-time mechanism $\mathcal{M}$ is never triggered, which implies that the system schedulability is not affected if $\mathcal{M}$ is not deployed.

If only some of the response-time bounds provided by Theorem 3 are lower than the corresponding deadlines, the system cannot be deemed schedulable, but such bounds can in turn be used to further reduce the response-time bounds of the other tasks. Note that, since the right-hand-side of Equation (7) is monotone in $\overline{R}_j$, by reducing at least one component of $\overline{\mathbf{R}}$, the response-time bound provided by Theorem 3 cannot increase. When no response-time bounds can be further reduced during the iterative loop, the algorithm terminates by signaling that the task set cannot be deemed schedulable.

VII. THE SRP-SS PROTOCOL

As seen in Section IV and analyzed in Section VI, a job of a self-suspending task may be blocked multiple times by lower priority tasks when the SRP is used. In fact, the number

---

[5] The operator $\uplus$ represents the union between multisets, e.g., $\{1, 1\} \uplus \{1, 2\} = \{1, 1, 1, 2\}$, and the product operator $\otimes$ multiplies the number of instances of every element in the multiset to which it is applied, e.g., $\{1, 2, 3\} \otimes 3 = \{1, 1, 1, 2, 2, 2, 3, 3, 3\}$.

Algorithm 1 Algorithm for checking the schedulability of a set Γ of self-suspending tasks under the SRP.

```
procedure isSchedulable($\Gamma$)
    $\forall \tau_i \in \Gamma,\ \overline{R}_i \leftarrow D_i$
    atLeastOneUpdate ← TRUE
    while (atLeastOneUpdate=TRUE) do
        atLeastOneUpdate ← FALSE
        for all $\tau_i \in \Gamma$ do
            $R_i \leftarrow$ Theorem 3
            if $R_i < \overline{R}_i$ then
                $\overline{R}_i \leftarrow Ri$
                atLeastOneUpdate ← TRUE
            end if
        end for
        if $\forall \tau_i \in \Gamma,\ R_i \leq D_i$ then
            return TRUE
        end if
    end while
    return FALSE
end procedure
```

of times a job of a self-suspending task τ_i may be blocked by lower priority tasks is dependent on the number of times τ_i suspends. This defeats the original goal of the SRP, namely, to ensure that each job of a task τ_i may be blocked *at most once* by a lower priority task, and that this blocking may happen only at the job release.

We propose an extension of the SRP to support self-suspending tasks. This new locking protocol is referred to as the *stack resource policy for self-suspending tasks* or SRP-SS in short. The SRP-SS can be configured to trade blocking suffered by higher priority tasks against interference suffered by lower priority ones. On one end of the configuration spectrum, the SRP-SS ensures that tasks will not suffer more than a single blocking by lower priority tasks (see Corollary 2 in Section VII-C), thereby bringing back the desirable property initially sought by the SRP. This configuration however, may drastically increase the interference suffered by the lowest priority tasks and therefore negatively impact the schedulability of the system. On the other end of the configuration spectrum, the SRP-SS behaves exactly as the SRP (Corollary 1 in Section VII-C). In such a configuration, higher priority self-suspending tasks suffer more blocking (see Theorem 2) but their interference on lower priority tasks is reduced (see Equation (7) for the WCRT analysis). This flexibility in the protocol configuration ensures retro-compatibility of the SRP-SS with systems developed for the SRP. Any configuration of the protocol that falls between those two extremes balances the effect of blocking and interference.

A. Main idea

We introduce the main idea behind the SRP-SS with an example.

Consider three tasks τ_1, τ_2 and τ_3 with $\pi_1 > \pi_2 > \pi_3$. Assume that τ_1 and τ_3 share a resource ℓ_1 with ceiling $\pi(\ell_1) = \pi_1$, and that τ_2 does not access any shared resource. Now, consider the schedule depicted in Fig. 2. Task τ_1 suspends two times. If the SRP is used for synchronising resource accesses, τ_1 may be blocked up to three times by τ_3 (see Fig. 2(a)).

To prevent this to happen, the SRP-SS introduces a new system parameter called *system priority* and denoted by Π^{ss}. A task can execute only if its priority is larger than Π^{ss}.

Coming back to our example, assume that, when task τ_1 starts executing, it raises the system priority Π^{ss} to π_3, and later reduces it to 0 when it completes. As seen on Fig. 2(b), τ_1 can still be blocked by τ_3 upon arrival, but τ_3 cannot resume its execution when τ_1 self-suspends. It results that τ_1 cannot be blocked anymore when its suspension ends. The reduction of τ_1's blocking time has however been at the cost of increasing the interference suffered by τ_3. Task τ_3 cannot execute during τ_1's self-suspension time and its response time is thus increased by as much. Note also that despite having a lower priority than τ_1, the response time of τ_2 is not impacted since its priority is still larger than Π^{ss}.

The system priority Π^{ss} may recall the notion of preemption threshold introduced in [23]–[25]. However, the technique introduced in this paper and the preemption threshold mechanism are quite different in nature as discussed in Section X.

B. The protocol

The SRP-SS introduces a *system priority* Π^{ss} and defines a new parameter π_i^{ss} for each task τ_i (see Cor. 1 and 2 or Section VIII for a discussion on how to assign a value to π_i^{ss}). At any time instant t, it holds that

$$\Pi^{ss} = \max_{\tau_j \in Act(t)} \{\pi_j^{ss}\}, \tag{8}$$

where $Act(t)$ is defined as the set of active tasks (i.e., those that started executing and did not yet complete) at time t.

The SRP-SS changes SRP's scheduling invariant to rely on the new system priority:

The SRP-SS scheduling invariant – At any time t, a task τ_i is *eligible* to execute if $\pi_i > \Pi^{ss}$.
The scheduler always executes the highest priority eligible task in the ready queue.

Algorithm 2 summarizes how a scheduling algorithm based on the SRP-SS can be implemented. Algorithm 2 only shows the differences with Algorithm 4. A new scheduling event is considered in Algorithm 2 in comparison to Algorithm 4, namely, the start of a job execution. Whenever a job starts executing, the set of active tasks Act is updated, and so is the system priority Π^{ss}. Similarly, Act and Π^{ss} are updated when a job completes its execution. All other scheduling events are treated as in Algorithm 4 and are not repeated here.

Note that the preemption rule is kept identical to that of the SRP. Therefore, with the above scheduling invariant, the SRP-SS reduces to the SRP if Π^{ss} is always equal to 0 (i.e., if $\pi_i^{ss} = 0$ for all tasks). In that particular case, all ready tasks are always eligible for execution and the highest priority ready task is always picked by the scheduler, same as the SRP.

The schedulability analysis under the SRP-SS is presented next. There are two effects that must be analyzed and incorporated into the schedulability analysis, the more explicit blocking in which a high-priority job is waiting for a lower-priority job to complete a critical section, and a more implicit blocking, also referred to as interference, that is caused by the system priority ceiling preventing an otherwise ready job from

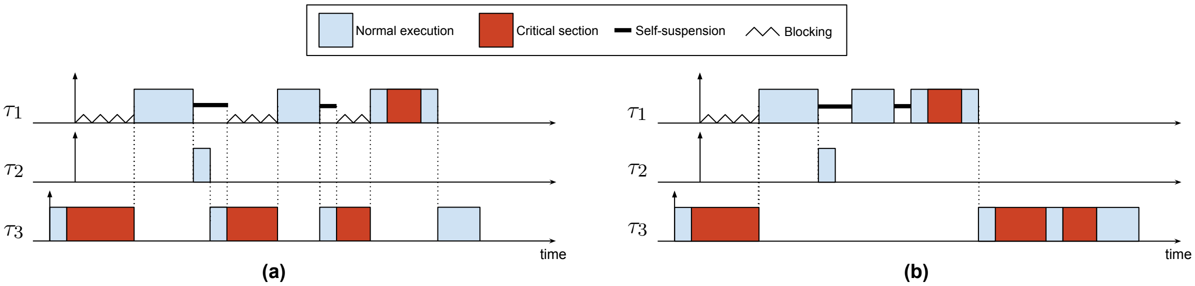


Figure 2. Scheduling example of three self-suspending tasks under (a) the SRP and (b) the SRP-SS. Up-arrows denote the release of a job.

Algorithm 2 Scheduling under the SRP-SS (only the changes w.r.t. Algorithm 4 are shown)

```
Init: $Q_r \leftarrow$ IdleTask; $Q_b \leftarrow \emptyset$; $Q_{ss} \leftarrow \emptyset$; $L \leftarrow 0$; $\Pi \leftarrow 0$; $\Pi^{ss} \leftarrow 0$;
Schedule invariant: RunTask $\leftarrow \operatorname{argmax}_{\tau_i \in Q_r}\{\pi_i \mid \pi_i > \Pi^{ss}\}$;

procedure StartExecute($\tau_j$)
    Add $\tau_j$ to $Act$
    $\Pi^{ss} \leftarrow \max_{\tau_j \in Act(t)}\{\pi_j^{ss}\}$
end procedure

procedure Complete($\tau_j$)
    Remove $\tau_j$ from $Q_r$
    Remove $\tau_j$ from $Act$
    $\Pi^{ss} \leftarrow \max_{\tau_j \in Act(t)}\{\pi_j^{ss}\}$
end procedure
```

executing. The former is discussed in Section VII-C, whilst the latter is treated in Section VII-D.

C. Blocking analysis

To analyze the the worst-case blocking time that a task τ_i may suffer under the SRP-SS, we first determine the set of tasks with lower priority than τ_i that can block τ_i after it self-suspended.

Lemma 5. *Under the SRP-SS, only tasks in the set* $mp(i) \overset{\text{def}}{=} \{\tau_j \in \Gamma \mid \pi_i > \pi_j > \pi_i^{ss}\}$ *may block* τ_i *after it self-suspended.*

Proof. Only tasks with lower priorities than τ_i may block the execution of τ_i with a critical section. Therefore, as for the SRP, only a task τ_j such that $\pi_i > \pi_j$ may block τ_i.

Moreover, because τ_i raises the system priority Π^{ss} to at least π_i^{ss} as soon as it starts executing, a task τ_j with priority $\pi_j \leq \pi_i^{ss}$ cannot execute anymore after τ_i started to execute. Therefore, it cannot execute during τ_i's suspension and cannot block τ_i when it resumes its execution. We conclude that a task τ_j may block τ_i after it suspended only if $\pi_j > \pi_i^{ss}$. □

Similar to the approach adopted for the analysis of the SRP, Lemma 6 explicitly identifies the set of critical sections that may block a task τ_i when it resumes after one of its suspensions.

Lemma 6. *The worst-case durations of the critical sections that can block* τ_i *after a self-suspension in an interval of length t, are included into the multiset* $\Delta_i^{ss}(t, \overline{\mathbf{R}})$ *defined as follows*

$$\Delta_i^{SS}(t, \overline{\mathbf{R}}) = \biguplus_{\tau_j \in mp(i)} \biguplus_{\ell_k \in Q} \{L_{j,k} \mid \pi(\ell_k) \geq \pi_i\} \otimes \big(N_{j,k} \times \eta_j(t, \overline{\mathbf{R}})\big)$$

where $\eta_j(t, \overline{\mathbf{R}}) = \left\lceil \frac{t + \overline{R}_j}{T_j} \right\rceil$.

Proof. By Lemma 5, only tasks within $mp(i)$ may block τ_i after one of its self-suspensions. Hence the first multiset union is restricted to tasks within $mp(i)$. Further, only the critical sections with a ceiling priority $\pi(\ell_k) \geq \pi_i$ may block τ_i for a maximum duration of $L_{j,k}$ time units, and it was already proven in Lemma 4 that each such critical section may be executed at most $(N_{j,k} \times \eta_j(t, \overline{\mathbf{R}}))$ times by τ_j within an interval of length t. The lemma follows. □

We can now derive an upper-bound on the blocking time suffered by a task τ_i under the SRP-SS.

Theorem 4. *Under the SRP-SS, the maximum blocking time incurred by a job of* τ_i *during an interval of length t is bounded by*

$$\begin{aligned} B_i^{SS}(t, \overline{\mathbf{R}}) = \max\{\ & \Sigma\left(X_i + 1,\, \Delta_i^{SS}(t, \overline{\mathbf{R}})\right); \\ & B_i^{lp} + \Sigma\left(X_i,\, \Delta_i^{SS}(t, \overline{\mathbf{R}})\right)\ \} \end{aligned} \tag{9}$$

where $B_i^{lp} = \max\limits_{\substack{\tau_j \in lp(i) \setminus mp(i) \\ \ell_k \in Q}} \{L_{j,k} \mid \pi(\ell_k) \geq \pi_i\}_0$.

Proof. Two execution scenarios may happen:

1) All critical sections blocking τ_i are from tasks in $mp(i)$. Since $\Delta_i^{SS}(t, \overline{\mathbf{R}})$ contains the worst-case duration of all critical sections accessed by tasks in $mp(i)$ in an interval of length t (Lemma 6), and because τ_i may be blocked at most $(X_i + 1)$ times (Lemma 3), the sum of the $(X_i + 1)$ largest elements in $\Delta_i^{SS}(t, \overline{\mathbf{R}})$ yields an upper-bound on $B_i^{SS}(t, \overline{\mathbf{R}})$ for that case.
2) Not all critical sections blocking τ_i are from tasks in $mp(i)$. In that case, at least one critical section of a task $\tau_j \in lp(i) \setminus mp(i)$ blocks τ_i. According to Lemma 5, tasks in $lp(i) \setminus mp(i)$ may only block τ_i's first execution segment (i.e., it cannot block τ_i after one of its suspensions). Further, each execution segment of τ_i may be blocked by a single critical section (Lemma 3). Therefore, at most one critical section of all the tasks within $lp(i) \setminus mp(i)$ may block τ_i's first execution segment. This single blocking is thus upper-bounded by B_i^{lp}. Since the tasks in $lp(i) \setminus mp(i)$ block τ_i at most once, all the

X_i other blockings (remember that a job of τ_i may be blocked at most $(X_i + 1)$ times) must be caused by tasks in $mp(i)$. This blocking is maximized by summing the X_i largest elements in $\Delta_i^{SS}(t, \overline{\mathbf{R}})$. The sum of B_i^{lp} and $\Sigma\left(X_i + 1,\, \Delta_i^{SS}(t, \overline{\mathbf{R}})\right)$ therefore yields an upper-bound on $B_i^{SS}(t, \overline{\mathbf{R}})$ for that case.

Taking the maximum blocking among those two scenarios yields an upper-bound on $B_i^{SS}(t, \overline{\mathbf{R}})$. □

Using the blocking bound proven in Theorem 4, we derive two properties of the SRP-SS w.r.t. the value of the parameter π_i^{ss} of each task τ_i.

Corollary 1. *If $\pi_i^{ss} = 0$, then $B_i^{SS}(t, \overline{\mathbf{R}}) = B_i(t, \overline{\mathbf{R}})$.*

Proof. If $\pi_i^{ss} = 0$, then $mp(i) = lp(i)$. Therefore, $\Delta_i^{SS}(t, \overline{\mathbf{R}}) = \Delta_i(t, \overline{\mathbf{R}})$. Further, $lp(i) \setminus mp(i) = \emptyset$ implying that $B_i^{lp} = 0$. From Theorem 4, it must therefore hold that $B_i^{SS}(t, \overline{\mathbf{R}}) = \Sigma\left(X_i + 1,\, \Delta_i^{SS}(t, \overline{\mathbf{R}})\right) = \Sigma\left(X_i + 1,\, \Delta_i(t, \overline{\mathbf{R}})\right) = B_i(t, \overline{\mathbf{R}})$. □

Corollary 2. *If $\pi_i^{ss} \geq \max_{\tau_j \in lp(i)} \{\pi_j \mid \exists \ell_k,\, N_{j,k} > 0 \wedge \pi(\ell_k) \geq \pi_i\}$, then τ_i may be blocked at most once by a lower-priority task, i.e., $B_i^{SS}(t, \overline{\mathbf{R}}) = B_i^{lp}$.*

Proof. If $\pi_i^{ss} \geq \max_{\tau_j \in lp(i)} \{\pi_j \mid \exists \ell_k,\, N_{j,k} > 0 \wedge \pi(\ell_k) \geq \pi_i\}$, then none of the tasks in $mp(i)$ accesses a resource ℓ_k with a ceiling $\pi(\ell_k) \geq \pi_i$ (i.e., for all tasks $\tau_j \in mp(i)$ and all resource $\ell_k \in Q$ such that $\pi(\ell_k) \geq \pi_i$, there is $N_{j,k} = 0$). Therefore, according to Lemma 6, we have $\Delta_i^{SS}(t, \overline{\mathbf{R}}) = \emptyset$ and $\Sigma\left(X_i + 1,\, \Delta_i^{SS}(t, \overline{\mathbf{R}})\right) = \Sigma\left(X_i,\, \Delta_i^{SS}(t, \overline{\mathbf{R}})\right) = 0$. It results that $B_i^{SS}(t, \overline{\mathbf{R}}) = B_i^{lp}$. □

Corollaries 1 and 2 prove two key properties of the SRP-SS, namely, that it reduces to the SRP if $\pi_i^{ss} = 0$ for all tasks in Γ, and that each task τ_i may be blocked by at most one critical section if $\pi_i^{ss} = \max_{\tau_j \in lp(i)} \{\pi_j \mid \exists \ell_k,\, L_{j,k} > 0 \wedge \pi(\ell_k) \geq \pi_i\}$. The former property proves that *the SRP-SS dominates the SRP*, and the latter proves that the SRP-SS may be configured to bring back the initial property of the SRP, that is that lower priority tasks may block higher priority tasks at most once.

D. Schedulability analysis

Theorem 4 upper-bounds the maximum blocking time incurred by a task τ_i under the SRP-SS. We use that result to derive a bound on the WCRT of τ_i that can then be injected in Algorithm 1 to check the schedulability of task set Γ.

We first decompose the set of higher priority tasks into two subsets $hp^{ob}(i)$ and $hp^{aw}(i)$ such that $hp(i) = hp^{ob}(i) \cup hp^{aw}(i)$. The set $hp^{ob}(i)$ is composed of all tasks with higher priority than τ_i that have their parameter π_j^{ss} greater than or equal to τ_i's priority. Formally, $hp^{ob}(i) = \{\tau_j \in hp(i) \mid \pi_j^{ss} \geq \pi_i\}$. The set $hp^{aw}(i)$ is then defined as $hp(i) \setminus hp^{ob}(i)$. In other words, $hp^{ob}(i)$ is the set of tasks with higher priority than τ_i that prevent τ_i to execute during their suspension time, while $hp^{aw}(i)$ is composed of the higher priority tasks that do not prevent τ_i to execute during their suspension time.

In Lem. 7 and 8, we derive an upper-bound on the interference caused by each task in each of those two subsets, and then integrate those results in τ_i's response time in Th. 5.

Lemma 7. *The worst-case contribution of a task $\tau_j \in hp^{ob}(i)$ to the interference suffered by a lower priority task τ_i in an interval of length t is upper-bounded by $\left\lceil \frac{t}{T_j} \right\rceil (C_j + S_j)$.*

Proof. Since τ_j prevents τ_i to execute during its suspension time, task τ_j is seen as a task with execution time $(C_j + S_j)$ and zero suspension time by τ_i. According to Lemma 1, the contribution of τ_j to τ_i's interference is then bounded by $\min\{C_j + S_j,\, 0\} + \left\lceil \frac{t}{T_j} \right\rceil (C_j + S_j)$, proving the lemma. □

Lemma 8. *The worst-case contribution of a task $\tau_j \in hp^{aw}(i)$ to the interference suffered by a lower priority task τ_i in an interval of length t is upper-bounded by*

$$\left\lceil \frac{t + \overline{R}_j - C_j}{T_j} \right\rceil C_j.$$

Proof. This directly follows from Lemma 2. □

Theorem 5. *Let R_i be the least positive fixed-point of the following recursive equation (if it exists)*

$$\begin{aligned} R_i = (C_i + S_i) + B_i^{SS}(R_i, \overline{\mathbf{R}}) + \sum_{\tau_j \in hp^{ob}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (C_j + S_j) \\ + \sum_{\tau_j \in hp^{aw}(i)} \left\lceil \frac{R_i + \overline{R}_j - C_j}{T_j} \right\rceil C_j. \end{aligned} \quad (10)$$

If $R_i \leq D_i$, then the worst-case response time of τ_i is upper-bounded by R_i and τ_i is schedulable.

Proof. The lemma directly follows from the summation of the bounds proven in Theorem 4 and Lemmas 7 and 8. □

The schedulabilty of a task set Γ under the SRP-SS can therefore be implemented using Algorithm 1 and replacing the call to Theorem 3 at Line 7 by a call to Theorem 5 instead.

VIII. CONFIGURING THE SRP-SS

The new protocol presented in the previous section requires to specify a new parameter π_i^{ss} for each task τ_i. Corollary 1 indicates a limit-case configuration for parameters π_i^{ss} such that the SRP-SS behaves as the SRP, that is, a task can be blocked every time it is resumed from a self-suspension. Corollary 2 instead corresponds to the other extreme of the configuration spectrum in which each task can be blocked at most once. All other configurations of the parameters π_i^{ss} allow trading low-priority blocking with high-priority interference.

Unfortunately, due to circular dependencies between tasks introduced by the response-time analysis proposed in Section VII-D, the computation of an optimal configuration for the SRP-SS is far from obvious. Nevertheless, this section presents a simple greedy algorithm to configure parameters π_i^{ss} with the aim of maximizing the system schedulability.

Algorithm 3 Algorithm for configuring the SRP-SS.

```
1:  procedure isSchedulableWithConfig($\Gamma$)
2:      $\pi_i^{ss} = 0, \forall \tau_i \in \Gamma$.
3:      while (TRUE) do
4:          if isSchedulable($\Gamma$) then
5:              return TRUE
6:          else
7:              $\tau_u \leftarrow argmax_{\tau_i \in \Gamma}\{\pi_i \mid R_i > D_i\}$
8:              $pSet \leftarrow \{\pi_l \mid \tau_l \in mp(u)\}$
9:              if $pSet$==$\emptyset$ then
10:                 return FALSE
11:             else
12:                 $\pi_u^{ss} = \min\{pSet\}$
13:             end if
14:         end if
15:     end while
16: end procedure
```

The proposed solution is reported in Algorithm 3. As a first step, all parameters π_i^{ss} are initialized to zero, i.e., the SRP-SS behaves as the SRP. Subsequently, the algorithm comprises a loop in which the schedulability of a task set Γ is tested by means of Algorithm 1 modified to handle the SRP-SS as indicated in the previous section. If the task set is schedulable, then the algorithm terminates by returning TRUE; otherwise, it tries to improve the task set schedulability by increasing the parameter π_i^{ss} of a task. The idea is to reduce the blocking incurred by a unschedulable task, hopefully making it schedulable. To this end, the algorithm identifies the highest-priority[6] task τ_u that is *not* schedulable (line 7). Then, it computes the set $pSet$ of priorities of low-priority tasks (with respect to τ_u) that can block τ_u, i.e., tasks $\tau_l \in mp(u)$ (line 8). If such a set is empty, then no possible improvements are possible and the task set is deemed unschedulable by returning FALSE. Otherwise, π_u^{ss} is increased to the minimum priority in $pSet$, i.e., to the priority of the lowest priority task τ_l that may block τ_u. After this update, τ_l cannot block τ_u anymore. This action may reduce the response time of τ_u at the cost of an increase of the response time of τ_l.

IX. EXPERIMENTAL RESULTS

The analyses and configuration techniques proposed in this paper were evaluated with an experimental study based on synthetic workload. This section reports and discusses the results obtained during our study.

Five different analyses/configuration techniques have been evaluated: **(i)** `SRP`, which corresponds to the fine-grained analysis for the SRP provided by Algorithm 1; **(ii)** `SRP-coarse`, which corresponds to a coarse-grained analysis for the SRP where Theorem 1 is used to bound the blocking time of each task; **(iii)** `SRP-optimistic`, which denotes an incorrect analysis for the SRP where the original SRP blocking analysis is used (Eq. (1)); **(iv)** `SRP-SS-cor2`, which denotes the analysis for the SRP-SS configured by applying Corollary 2 to all tasks; **(v)** `SRP-SS-config`, which denotes the analysis for the SRP-SS configured with Algorithm 3.

1) Workload generation: Given a target utilization U, task sets Γ composed of n tasks have been generated using the Emberson et al.'s generator [26]. Periods are randomly picked in the range $[1, 1000]$ms with log-uniform distribution. For each task τ_i, the relative deadline D_i, the number of suspensions x_i and the total suspension time S_i have been randomly generated with uniform distributions from the intervals $[C_i + \beta(T_i - C_i), T_i]$, $[X^{min}, X^{max}]$, and $[\sigma^{min} D_i, \sigma^{max} D_i]$, respectively, where β, X^{min}, X^{max}, σ^{min} and σ^{max} are generation parameters. Given a target number of shared resources n_r, critical sections have been generated as follows. First, for each resource ℓ_q, n^* tasks have been randomly selected to share ℓ_q, where n^* is randomly selected in the range $[2, \lfloor rsf \cdot n \rfloor]$ with uniform distribution. The parameter rsf denotes the *resource sharing factor* and allows controlling the "amount" of resource sharing. Second, for each task τ_i selected to access ℓ_q, $N_{i,q}$ and $L_{i,q}$ have been randomly selected from the ranges $[N^{min}, N^{max}]$ and $[L^{min}, L^{max}]$, respectively, both with uniform distribution. N^{min}, N^{max}, L^{min}, and L^{max} are other generation parameters. To avoid generating unrealistic task sets, the generation enforces that all critical sections must fit within the task's WCET, i.e., $\sum_{\ell_q} N_{i,q} \cdot L_{i,q} \leq C_i, \forall \tau_i \in \Gamma$: critical sections that violate this constraints are discarded and re-generated from scratch up to 10^6 times. If no proper configuration could be generated after 10^6 attempts the task set is skipped. During the reported experiments, we ensured that no more than 1% of the generated task sets were skipped.

2) Experiment 1: The first experiment tries to be representative of applications managed by AUTOSAR/OSEK operating systems [6] in which the `RES_SCHEDULER` resource is used. `RES_SCHEDULER` is a virtual resource *implicitly* shared by all tasks that is essentially used to lock the scheduler, i.e., once such a resource is locked, no tasks can preempt the lock holder. This feature is commonly used to implement non-preemptable execution sections or to integrate legacy software and third-party software whose accessed resources are unknown when the operating system is configured. The major issue in using `RES_SCHEDULER` is that all tasks (except the one with lowest priority) can incur blocking. Clearly, this may be harmful for high-priority latency sensitive tasks, especially when self-suspensions are present. This experiment mainly aims at assessing whether the schedulability of AUTOSAR/OSEK applications can be improved by replacing the SRP with the SRP-SS. Task sets using `RES_SCHEDULER` are generated by first adopting the workload generation presented earlier, and then enforcing that each task τ_i accesses one particular resource ℓ_q for $\max\{1, N_{i,q}\}$ times.

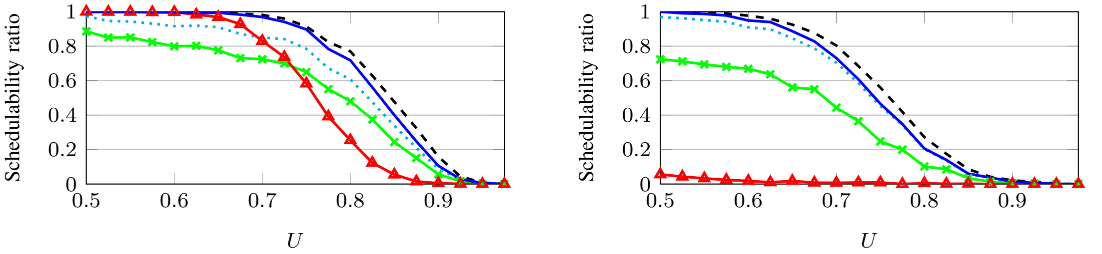
A multidimensional exploration of the generation parameters has been performed whose results are partially reported in Fig. 3. Three major trends emerged: **(i)** the SRP-SS definitively allows improving the system schedulability in comparison to the SRP, especially in the presence of large critical sections ($> 300\mu s$) and tasks with constrained deadlines ($\beta = 0.75$); **(ii)** configuring the SRP-SS such that each task can be blocked by at most one critical section using Corollary 2, is convenient only for task sets with low utilization, while it generally leads to very low schedulability in the presence of a high utilization; **(iii)** our new fine-grained schedulability analysis for the SRP is quite accurate, since the gain in schedulabilty in comparison to the coarse analysis is very large (between 10 and 30%), and

---

[6] Choosing the highest-priority unschedulable task is arbitrary but simplifies the configuration algorithm.

(a) $n = 10$, $n_r = 4$, $N^{max} = 3$,$L^{min} = 10$, $L^{max} = 500$, $\sigma^{min} = 0$, $\sigma^{max} = 0.05$, $X^{min} = 2$, $X^{max} = 5$

(b) $n = 10$, $n_r = 6$, $N^{max} = 1$,$L^{min} = 10$, $L^{max} = 500$, $\sigma^{min} = 0$, $\sigma^{max} = 0.2$, $X^{min} = 2$, $X^{max} = 5$



(c) $n = 10$, $n_r = 4$, $N^{max} = 3$,$L^{min} = 10$, $L^{max} = 500$, $\sigma^{min} = 0$, $\sigma^{max} = 0.1$, $X^{min} = 2$, $X^{max} = 5$

(c) $n = 15$, $n_r = 8$, $N^{max} = 1$,$L^{min} = 10$, $L^{max} = 200$, $\sigma^{min} = 0$, $\sigma^{max} = 0.1$, $X^{min} = 4$, $X^{max} = 8$

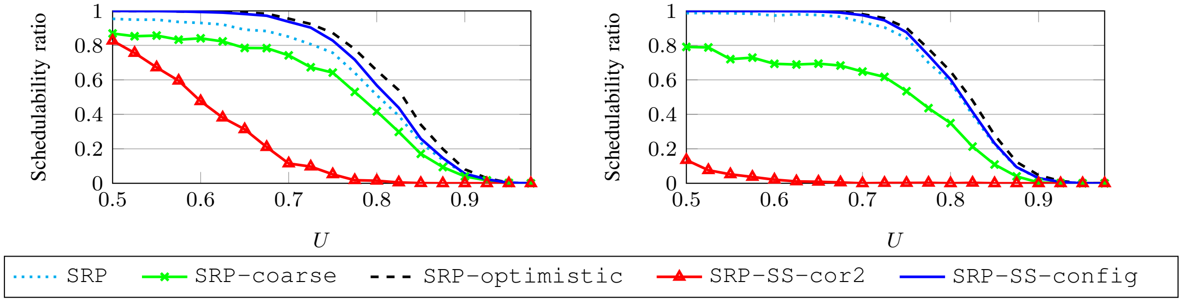


Figure 3. Experimental results for Experiment 1 (insets (a) and (b)) and Experiment 2 (insets (c) and (d)). All the graphs are related to configurations with $\beta = 0.75$. The other configuration parameters are reported in the captions above the graphs.

the distance from the optimistic, hence unsafe analysis is very limited (from 2 to 10%) for many configurations.

Figures 3(a) and (b) report the results of two representative configurations (generation parameters are reported above each graph), which have been obtained by varying the utilization U from 0.5 to 0.975 and testing 1000 task sets for each utilization value. As it can be observed from Figure 3(a), the `SRP-SS-config` shows a considerable improvement in comparison to the fine-grained analysis for the SRP, exhibiting a performance gap of up to 12% for $U = 0.7$. Furthermore, note that the fine-grained analysis for the SRP also shows a consistent improvement over `SRP-coarse`, with a performance gap of 14%. The `SRP-SS-cor2` approach is beneficial up to $U = 0.7$, and then tends to show very low schedulability performance as the system utilization increases. Figure 3(b) reports the results when the maximum suspension time is doubled, the number of resources is increased, and the number of critical sections per task is decreased. In this configuration, the performance gap between `SRP-SS-config` and `SRP` is reduced to about 4%, while the improvement of `SRP` upon `SRP-coarse` is more than doubled. The reduction in the effectiveness of the SRP-SS was expected since due to the larger task suspension times, the increased interference generated by suspension regions when π_i^{ss} increases, dominates the gain in terms of blocking time. In most cases, Algorithm 3 will not find a better configuration than the SRP.

Finally, it is worth observing that the `SRP-optimistic` curve also corresponds to the ideal case in which each task is blocked by at most one critical section and there is no extra-interference generated by self-suspension regions. The experimental results show that the schedulability performance of `SRP-SS-config` are not that far from the one of `SRP-optimistic` (about 3% of difference), hence quantifying the quality of the solution provided by Algorithm 3.

3) Experiment 2: This experiment considered the same generation scheme assumed for Experiment 1 but without the presence of `RES_SCHEDULER`, and therefore aims at being representative of task sets with typical shared resources. The results of two configurations are reported in Figures 3(c) and 3(d). The graphs show similar trends to those emerged in Experiment 1, with a slight reduction on the performance gap between `SRP` and `SRP-optimistic` and a consequent reduction of the gap between `SRP` and `SRP-SS-config`. Such a gap further reduces with shorter critical sections and larger number of tasks (Figure 3(d)). Furthermore, such graphs confirm that the `SRP-SS-cor2` approach tends to degrade its performance as the number of resources and the suspension times increase. Finally, Figure 3(d) shows the benefits of adopting the proposed fine-grained analysis for the SRP, in comparison to the coarse blocking bound, specially in presence of a larger number of tasks and resources ($n = 15$ and $n_r = 8$), achieving a schedulability gain of up to 30%.

X. RELATED WORK

This work is not about the RTA of self-suspending tasks but rather on resource sharing protocols. Therefore, we do not discuss this topic in this section: the interested reader can refer to the survey in [2] and Section III already provides several references to relevant work on that topic.

Despite the fundamental relevance of classical real-time locking protocols, such as the SRP [4], and the priority ceiling and priority inheritance protocols [7], to the best of our knowledge efforts spent on their integration with self-suspending tasks were limited at best. In [16], Rajkumar et al. modelled blocking due to resources globally shared in multi-processor platforms as self-suspensions. One of the terms in their blocking bound is similar to the coarse-grained analysis proposed in this paper. Similarly, Brandenburg derives a coarse-grained analysis of blocking dues to self-suspension for the FLMP+ resource sharing protocol [19]. We propose a tighter response time analysis, tailored to self-suspending tasks.

State-of-the-art analyses for multiprocessor locking protocols have adopted a similar technique to our more accurate blocking analysis of the SRP, to characterize the set of critical sections that can effectively block higher priority tasks [22], [27]: this approach is typically referred to as inflation-free analysis, and was first proposed by Brandenburg in [13].

Works on semaphore-based multiprocessor locking protocols [13], [28] also considered the integration of blocking with self-suspensions, but under the case in which suspensions are originated by waiting times for resources locked on a remote processor or when suspensions happen within a critical section (e.g., due to I/O accesses). To the best of our records, there is no work that attempts to mitigate the additional blocking originating from self-suspension, other then by using spinlocks [22], effectively replacing suspension by computation.

The FMLP+ family of algorithms [19], [29] uses priority boosting to limit blocking time and avoid starvation for self-suspending tasks running on multicore platforms. However, priority boosting in FLMP+ works only for lock-related suspensions. It was not designed and does not help in the occurrence of suspensions that are not the result of waiting for a lock. On the other hand, the SRP-SS provides a fine-grained control of the blocking originated by locking-unrelated suspensions. Furthermore, FMLP+ enforces a suspension when a lock is busy, while the SRP-SS does not introduce additional suspensions.

The new system priority Π^{ss} introduced by the SRP-SS resembles the preemption threshold mechanism first proposed in [23] and then extended in [24], [25] to improve the schedulability of non-self-suspending tasks under task level fixed priority scheduling. Beside relying on a system priority in both cases, this work and those on preemption thresholds have drastically different goals. Preemption threshold aims at reducing the higher priority interference by deferring preemptions, while the SRP-SS reduces the lower priority interference by preventing blocking. The task parameter π_i^{ss} is thus assigned to suspension regions and is set to a smaller value than π_i, where a preemption threshold parameter would be assigned to execution segments and would be set to a higher value than π_i. Those works are thus incomparable but mixing the two mechanisms may be worth investigating in the future.

XI. CONCLUSION

In this work, we have presented two schedulability analyses for the SRP when tasks are allowed to self-suspend. The most accurate of the two has shown very good results during the experimental evaluation with up to 30% increase in the task set schedulability ratio in comparison to the second (coarse-grained) one. Additionally, we proposed the SRP-SS, a generalization of the SRP, together with its own schedulability analysis and a greedy configuration algorithm, to cope with the specificities of self-suspending tasks. Beside the theoretical dominance over the SRP, the experimental results showed that the SRP-SS may increase the schedulability (+10 to 12%) of task sets with large critical sections ($> 300\mu s$) as it may happen in real applications, specially in the presence of non-preemptive or partially non-preemptive workload. Further, we expect the SRP-SS protocol to be sufficiently close to the SRP to allow for an easy integration in existing operating systems, with limited overheads.

APPENDIX A

Algorithm 4 Pseudo-code of a scheduler using the SRP

```
Init: $Q_r \leftarrow$ IdleTask; $Q_b \leftarrow \emptyset$; $Q_{ss} \leftarrow \emptyset$; $L \leftarrow 0$; $\Pi \leftarrow 0$;
Schedule invariant: RunningTask $\leftarrow \text{argmax}_{\tau_i \in Q_r}\{\pi_i\}$;

procedure RELEASE($\tau_j$)
    if $\pi_j > \Pi$ then
        Add $\tau_j$ to $Q_r$
    else
        Add $\tau_j$ to $Q_b$
    end if
end procedure

procedure COMPLETE($\tau_j$)
    Remove $\tau_j$ from $Q_r$
end procedure

procedure SUSPEND($\tau_j$)
    Remove $\tau_j$ from $Q_r$
    Add $\tau_j$ to $Q_{ss}$
end procedure

procedure RESUME($\tau_j$)
    Remove $\tau_j$ from $Q_{ss}$
    Add $\tau_j$ to $Q_r$
end procedure

procedure LOCK($\tau_j$, $\ell_k$)
    Push $\pi(\ell_k)$ on $L$
    $\Pi \leftarrow \pi(\ell_k)$
end procedure

procedure UNLOCK($\tau_j$, $\ell_k$)
    Pop $\pi(\ell_k)$ from $L$
    $\Pi \leftarrow L.top$
    for all $\tau_j \in Q_b$ s.th. $\pi_j > \Pi$ do
        Remove $\tau_j$ from $Q_b$
        Add $\tau_j$ to $Q_r$
    end for
end procedure
```

ACKNOWLEDGMENT

This work has been partially supported by the RETINA Eurostars Project E10171 and by National Funds through FCT (Portuguese Foundation for Science and Technology) within the CISTER Research Unit (CEC/04234).

REFERENCES

[1] G. Nelissen, J. Fonseca, G. Raravi, and V. Nelis, "Timing analysis of fixed priority self-suspending sporadic tasks," in *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, July 2015.

[2] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, Neil, Audsley, R. Rajkumar, and D. de Niz, "Many suspensions, many problems: A review of self-suspending tasks in real-time systems," Faculty of Informatik, TU Dortmund, Tech. Rep. 854, 2016.

[3] K. Bletsas, N. C. Audsley, W.-H. Huang, J.-J. Chen, and G. Nelissen, “Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions,” *Leibniz Transactions on Embedded Systems*, vol. 5, no. 1, pp. 2:1–2:20, 2018.

[4] T. P. Baker, “Stack-based scheduling for realtime processes,” *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, April 1991.

[5] P. Gai, G. Lipari, and M. D. Natale, “Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip,” in *Proceedings of IEEE Real-Time Systems Symposium*, 2001.

[6] The AUTOSAR standard, version 4.3. [Online]. Available: http://www.autosar.org

[7] L. Sha, R. Rajkumar, and J. Lehoczky, “Priority inheritance protocols: An approach to real-time synchronization,” *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, September 1990.

[8] A. Biondi, M. Di Natale, Y. Sun, and S. Botta, “Moving from single-core to multicore: initial findings on a fuel injection case study,” in *SAE Technical Paper, SAE Conference, Detroit, USA*, April 2016.

[9] M. Lowinski, D. Ziegenbein, and S. Glesner, “Splitting tasks for migrating real-time automotive applications to multi-core ecus,” in *11th IEEE Symposium on Industrial Embedded Systems (SIES 2016)*, May 2016.

[10] J. Fonseca, G. Nelissen, V. Nelis, and L. M. Pinho, “Response time analysis of sporadic dag tasks under partitioned scheduling,” in *11th IEEE Symposium on Industrial Embedded Systems (SIES)*, May 2016.

[11] A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo, “A framework for supporting real-time applications on dynamic reconfigurable FPGAs,” in *Real-Time Systems Symposium (RTSS), 2016 IEEE*. IEEE, 2016, pp. 1–12.

[12] J.-J. Chen, G. Nelissen, and W.-H. Huang, “A unifying response time analysis framework for dynamic self-suspending tasks,” in *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, July 2016.

[13] B. B. Brandenburg, “Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling,” in *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2013.

[14] M. Joseph and P. Pandya, “Finding response times in a real-time system,” *The Computer Journal*, vol. 29, no. 5, pp. 390–395, 1986.

[15] J. W. S. W. Liu, *Real-Time Systems*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[16] R. Rajkumar, L. Sha, and J. P. Lehoczky, “Real-time synchronization protocols for multiprocessors,” in *Proceedings of the 9th IEEE Real-Time Systems Symposium (RTSS)*, 1988, pp. 259–269.

[17] J.-J. Chen, W.-H. Huang, and G. Nelissen, “A note on modeling self-suspending time as blocking time in real-time systems,” Tech. Rep., 2016, http://arxiv.org/abs/1602.07750.

[18] G. v. d. Brüggen, W.-H. Huang, and J.-J. Chen, “Hybrid self-suspension models in real-time embedded systems,” in *23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*, Hsinchu, Taiwan, August 16-18 2017, pp. 1–9.

[19] B. B. Brandenburg, “The fmlp+: An asymptotically optimal real-time locking protocol for suspension-aware analysis,” in *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*. IEEE, 2014, pp. 61–71.

[20] M. Mohaqeqi, P. Ekberg, and W. Yi, “On fixed-priority schedulability analysis of sporadic tasks with self-suspension,” in *Proc. of 24th International Conference on Real-Time Networks and Systems (RTNS)*, 2016.

[21] K. Bletsas, “Worst-case and best-case timing analysis for real-time embedded systems with limited parallelism,” Ph.D. dissertation, University of York, 2007.

[22] M. Yang, A. Wieder, and B. Brandenburg, “Global real-time semaphore protocols: A survey, unified analysis, and comparison,” in *Proceedings of the 36th Real-Time Systems Symposium (RTSS)*, 2015.

[23] Y. Wang and M. Saksena, “Scheduling fixed-priority tasks with preemption threshold,” in *Real-Time Computing Systems and Applications, 1999. RTCSA’99. Sixth International Conference on*. IEEE, 1999, pp. 328–335.

[24] U. Keskin, R. J. Bril, and J. J. Lukkien, “Exact response-time analysis for fixed-priority preemption-threshold scheduling,” in *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*. IEEE, 2010, pp. 1–4.

[25] R. J. Bril, M. M. van den Heuvel, and J. J. Lukkien, “Improved feasibility of fixed-priority scheduling with deferred preemption using preemption thresholds for preemption points,” in *Proceedings of the 21st International conference on Real-Time Networks and Systems*. ACM, 2013, pp. 255–264.

[26] P. Emberson, R. Stafford, and R. I. Davis, “Techniques for the synthesis of multiprocessor tasksets,” in *WATERS Workshop*, 2010, pp. 6–11.

[27] A. Wieder and B. Brandenburg, “On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks,” in *Proceedings of the 34th IEEE Real-Time Systems Symposium (RTSS)*, December 2013, pp. 45–56.

[28] P. Patel, I. Baek, H. Kim, and R. R. Rajkumar, “Analytical enhancements and practical insights for mpcp with self-suspensions,” in *Proceedings of the 24th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, December 2013, pp. 45–56.

[29] B. B. Brandenburg, “Scheduling and locking in multiprocessor real-time operating systems,” Ph.D. dissertation, University of North Carolina at Chapel Hill, 2011.