Refining SIRAP with a Dedicated Resource Ceiling for Self-Blocking

Moris Behnam, Thomas Nolte
Mälardalen Real-Time Research Centre
P.O. Box 883, SE-721 23
Västerås, Sweden
{moris.behnam,thomas.nolte}@mdh.se

Reinder J. Bril
Technische Universiteit Eindhoven (TU/e)
Den Dolech 2, 5612 AZ
Eindhoven, The Netherlands
r.j.bril@tue.nl

ABSTRACT
In recent years, several synchronization protocols for resource sharing have been presented for use in a Hierarchical Scheduling Framework (HSF). An initial comparative assessment of existing protocols revealed that none of the protocols is superior to the others and that the performance of a protocol heavily depends on system parameters. In this paper, we aim at efficiency improvements of the synchronization protocol SIRAP [?] and its associated schedulability analysis, where efficiency refers to calculated CPU resource needs. The contribution of the paper is threefold. Firstly, we present an improvement of the schedulability analysis for SIRAP, which makes SIRAP more efficient. Secondly, we generalize SIRAP by distinguishing separate resource ceilings for self-blocking and resource access. Using a separate resource ceiling for self-blocking enables a reduction of the interference from lower priority tasks, which can result in efficiency improvements. The efficiency improvement depends on both subsystem characteristics and the value selected for the resource ceiling for self-blocking, however. The third contribution of this paper is therefore an algorithm that given a subsystem selects for each globally shared resource an optimal value in terms of efficiency for its resource ceiling for self-blocking. The efficiency improvement gained by the algorithm compared to the original SIRAP approach is evaluated by means of simulation.

Categories and Subject Descriptors
D.4.1 [OPERATING SYSTEMS]: Process Management—Scheduling; Synchronization; D.4.7 [OPERATING SYSTEMS]: Organization and Design—Real-time systems and embedded systems.

General Terms
Algorithms, Design.

*The work in this paper is supported by the Swedish Foundation for Strategic Research (SSF), via the research programme PROGRESS.

Keywords
Hierarchical scheduling, Real-time open systems, Real-time subsystem integration, Resource-sharing, SIRAP, Synchronization protocol.

1. INTRODUCTION
The Hierarchical Scheduling Framework (HSF) has been introduced to support hierarchical CPU sharing among applications under different scheduling services [?]. The HSF can be represented as a tree of nodes, where each node represents an application with its own scheduler for scheduling internal workloads (e.g., tasks), and resources are allocated from a parent node to its children nodes.

The HSF provides means for decomposing a complex system into well-defined parts called subsystems. In essence, the HSF provides a mechanism for timing-predictable composition of course-grained subsystems. In the HSF a subsystem provides an introspective interface that specifies the timing properties of the subsystem precisely [?]. This means that subsystems can be independently developed and tested, and later assembled without introducing unwanted temporal interference. Temporal isolation between subsystems is provided through budgets which are allocated to subsystems.

Motivation: Research on HSFs started with the assumption that subsystems are independent, i.e., inter-subsystem resource sharing other than the CPU fell outside their scope. In some cases [?, ?], intra-subsystem resource sharing is addressed using existing synchronization protocols for resource sharing between tasks, e.g., the Stack Resource Policy (SRP) [?]. Recently, three SRP-based synchronization protocols for inter-subsystem resource sharing have been presented, i.e., HSRP [?], BROE [?], and SIRAP [?]. Although all three protocols are SRP-based, they rely on different mechanisms to deal with inter-subsystem resource sharing and depletion of budgets. In particular, HSRP is based on a so-called overrun mechanism, whereas both BROE and SIRAP are based on a so-called skipping approach. Moreover, their constituting frameworks are based on different assumptions. As an example, scheduling (of subsystems as well as of tasks) in the frameworks of HSRP and SIRAP is based on FPPS, whereas it is based on EDF for BROE. Finally, unlike BROE and SIRAP, HSRP does not support local schedulability analysis, and the local schedulability analysis in BROE as described in [?] is incomplete. An initial comparative assessment of these three synchronization protocols [?] revealed that none of them was superior to the
others and that the performance of a protocol heavily depends on the system parameters. A comparative evaluation of the mechanisms overrun and skipping using a single framework can be found in [?]. In this paper, we focus on SIRAP and aim at improving the efficiency of the protocol and its associated schedulability analysis, where efficiency refers to calculated CPU resource needs of a subsystem.

SIRAP is based on a skipping mechanism to prevent deple- tion of a subsystem budget during global shared resource access. That is, whenever a task tries to lock a global shared resource and the remaining budget is insufficient to complete the access, the task experiences self-blocking during the remainder of the current budget period and is guaranteed to access the resource during the next budget period. The contribution of this paper is threefold. Firstly, we remove some pessimism from SIRAP by improving its associated schedulability analysis. Secondly, we generalize SIRAP by distinguishing separate resource ceilings for self-blocking and resource access. Using a dedicated resource ceiling for self-blocking enables a reduction of the interference from lower priority tasks which may reduce the calculated resource needs of the subsystem whilst guaranteeing the schedulability of all its internal tasks. Thirdly, we propose an algorithm to select the optimal value per global shared resource for this novel resource ceiling for a subsystem with given characteristics, resulting in the lowest calculated resource needs for that subsystem. The efficiency of the algorithm is evaluated by comparing its calculated resource needs with those of the original SIRAP protocol in a simulation.

2. RELATED WORK
Over the years, there has been a growing attention to hierarchical scheduling of real-time systems [?, ?, ?, ?, ?, ?, ?, ?, ?, ?]. Deng and Liu [?] proposed a two-level HSF for open systems, where subsystems may be developed and validated independently. Kuo and Li [?] presented schedulability analysis techniques for such a two-level framework with the Fixed-Priority Scheduling (FPS) global scheduler. Lipari and Baruah [?, ?] presented schedulability analysis techniques for Earliest Deadline First (EDF) global schedulers. Mok et al. [?, ?] proposed the bounded-delay virtual processor model to achieve a clean separation in a multi-level HSF. In addition, Shin and Lee [?] introduced the periodic virtual processor model (to characterize the periodic CPU allocation behaviour), and many studies have been proposed on schedulability analysis with this model under FPS [?, ?, ?] and under EDF scheduling [?, ?]. However, a common assumption shared by all above studies is that tasks are independent.

Recently, three SRP-based synchronization protocols for inter-subsystem resource sharing have been presented, i.e., HSRP [?], BROE [?], and SIRAP [?]. A comparative assessment of these three synchronization protocols [?] revealed that none of them was superior to the others and that the performance of a protocol heavily depends on system parameters.

3. SYSTEM MODEL AND BACKGROUND
This paper focuses on scheduling of a single node or a single network link, where each node (or link) is modeled as a system \( S \) consisting of one or more subsystems \( S_s \in S \). The system is scheduled by a two-level HSF as shown in Figure ???. During runtime, the system level scheduler (global scheduler) selects, at all times, which subsystem will access the common (shared) CPU resource.

Subsystem model. A subsystem \( S_s \) consists of a set \( T_s \) of \( n_s \) tasks and a local scheduler. Once a subsystem is assigned the processor (CPU), its scheduler will select which of its tasks will be executed. With each subsystem \( S_s \), a subsystem timing interface \( S_s(P_s, Q_s, X_s) \) is associated, where \( Q_s \) is the subsystem budget that the subsystem \( S_s \) will receive every subsystem period \( P_s \), and \( X_s \) is the maximum time that a subsystem internal task may lock a shared resource. Finally, both the local scheduler of a subsystem \( S_s \) as well as the global scheduler of the system \( S \) is assumed to implement the fixed priority preemptive scheduling policy. Let \( \mathcal{R} \) be the set of \( m \) global shared resources accessed by \( S_s \).

Task model. The task model considered in this paper is the deadline-constrained sporadic hard real-time task model \( \tau_i(T_i, C_i, D_i, (c_{i,j})) \), where \( T_i \) is a minimum separation time between arrival of successive jobs of \( \tau_i \), \( C_i \) is their worst-case execution-time, and \( D_i \) is an arrival-relative deadline \( (0 < C_i \leq D_i \leq T_i) \) before which the execution of a job must be completed. Each task is allowed to access one or more shared logical resources, and each element \( c_{i,j} \in (c_{i,j}) \) is a critical section execution time that represents a worst-case execution-time requirement within a critical section of a global shared resource \( R_j \). It is assumed that all tasks belonging to the same subsystem are assigned unique static priorities and are sorted according to their priorities in the order of increasing priority. Without loss of generality, it is assumed that the priority of a task is equal to the task ID number after sorting, and the greater a task ID number is, the higher its priority is. The same assumption is made for
the subsystems. The set of shared resources accessed by \( \tau_i \) is denoted \( \{R^i\} \). Let \( hp(i) \) return the set of local tasks that belong to a subsystem with priorities higher than that of \( \tau_i \) and \( lp(i) \) return the set of local tasks with priorities lower than that of task \( \tau_i \). Table 1 shows the summary of the notations used in this paper. For each subsystem, we assume that the subsystem period is selected such that \( 2P_s \leq T_{\min} \), where \( T_{\min} \) is the task with the shortest period. The motivation for this assumption is that higher \( P_s \) will require more CPU resources [?]. In addition, this assumption simplifies the presentation of the paper (evaluating \( X_s \)).

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>System</td>
</tr>
<tr>
<td>( S_k )</td>
<td>Subsystem</td>
</tr>
<tr>
<td>( T_s )</td>
<td>Sporadic task set</td>
</tr>
<tr>
<td>( n_s )</td>
<td>Number of local tasks.</td>
</tr>
<tr>
<td>( P_s )</td>
<td>Subsystem period</td>
</tr>
<tr>
<td>( Q_s )</td>
<td>Subsystem budget</td>
</tr>
<tr>
<td>( X_s )</td>
<td>Maximum time that ( S_s ) locks a global shared resource.</td>
</tr>
<tr>
<td>( R_s )</td>
<td>Set of global shared resources accessed by ( S_s )</td>
</tr>
<tr>
<td>( m_s )</td>
<td>Cardinality of ( R_s )</td>
</tr>
<tr>
<td>( \tau_i )</td>
<td>Sporadic task</td>
</tr>
<tr>
<td>( T_i )</td>
<td>Period of ( \tau_i )</td>
</tr>
<tr>
<td>( C_i )</td>
<td>Worst case execution time of ( \tau_i )</td>
</tr>
<tr>
<td>( D_i )</td>
<td>Relative deadline of ( \tau_i )</td>
</tr>
<tr>
<td>( c_{i,j} )</td>
<td>Critical section execution times of ( \tau_i ) accessing ( R_j )</td>
</tr>
<tr>
<td>( {R^i} )</td>
<td>Set of critical section execution times of ( \tau_i ) accessing ( R_j )</td>
</tr>
<tr>
<td>( hp(i) )</td>
<td>Set of local tasks with priorities higher than that of ( \tau_i )</td>
</tr>
<tr>
<td>( lp(i) )</td>
<td>Set of local tasks with priorities lower than that of ( \tau_i )</td>
</tr>
<tr>
<td>( rc_{i,j} )</td>
<td>Internal resource ceiling of ( R_j )</td>
</tr>
<tr>
<td>( RX_j )</td>
<td>External resource ceiling</td>
</tr>
<tr>
<td>( SC )</td>
<td>System ceiling</td>
</tr>
<tr>
<td>( s_{sc_s} )</td>
<td>Subsystem ceiling</td>
</tr>
<tr>
<td>( rc_{LWB}^j )</td>
<td>Lower bounds for ( rc_{i,j} )</td>
</tr>
<tr>
<td>( RX_{LWB}^j )</td>
<td>Lower bound for ( RX_j )</td>
</tr>
<tr>
<td>( sbf_s(t) )</td>
<td>Supply bound function</td>
</tr>
<tr>
<td>( rbf_{sp}(i,t) )</td>
<td>Request bound function of ( \tau_i )</td>
</tr>
<tr>
<td>( I_S(i) )</td>
<td>Self blocking of ( \tau_i )</td>
</tr>
<tr>
<td>( I_H(i,t) )</td>
<td>Interference from tasks with priority higher than that of ( \tau_i )</td>
</tr>
<tr>
<td>( I_L(i) )</td>
<td>Interference from tasks, with priority lower than that of ( \tau_i )</td>
</tr>
<tr>
<td>( src_{i,j} )</td>
<td>Self blocking ceiling of ( R_j )</td>
</tr>
</tbody>
</table>

Table 1: Summary of notations.

\( RX \) and \( SC \) are defined as \( RX_{LWB} = \max \{ s | S_s \text{ accesses } R_j \} \) and \( SC = \max \{ s | S_s \text{ accesses } R_j \} \), respectively.

### System/subsystem ceiling

The system/subsystem ceilings \((SC/sc_s)\) are dynamic parameters that change during execution. The system/subsystem ceiling is equal to the highest external/internal resource ceiling of a currently locked resource in the system/subsystem.

Under SRP, a task \( \tau_k \) can preempt the currently executing task \( \tau \) (even inside a critical section) within the same subsystem, only if the priority of \( \tau_k \) is greater than its corresponding subsystem ceiling. The same reasoning applies for subsystems from a global scheduling point of view. An attractive property of SRP is that it allows tasks within a subsystem to share a common stack.

### 4. SIRAP

The SIRAP [?] protocol can be used for independent development of subsystems and it supports subsystem integration in the presence of globally shared logical resources. It uses a periodic resource model [?] to abstract the timing requirements of each subsystem. SIRAP uses the SRP protocol to synchronize access to global shared resources in both local and global scheduling. SIRAP applies a skipping approach to prevent the budget expiration inside critical section problem. The mechanism works as follows: when a job wants to enter a critical section, it enters the critical section at the earliest instant such that it can complete the critical section execution before the subsystem budget expires. This can be achieved by checking the remaining budget before granting the access to globally shared resources; if there is sufficient remaining budget then the job enters the critical section, and if there is insufficient remaining budget, the local scheduler delays the critical section entering of the job (i.e., the job blocks itself and its state becomes self blocking) until the next subsystem budget replenishment and guarantees access to the resource during the next subsystem budget period. In addition, it sets the subsystem ceiling equal to the internal resource ceiling of the resource that the self blocked job wanted to access, to prevent the execution of all tasks that have a priority at most equal to the ceiling of the resource until the job releases the resource.

**Local schedulability analysis**. The local schedulability analysis under FPS is as follows [?, ?]:

---

**Shared resources**. The presented HSF allows for sharing of logical resources between arbitrary tasks, located in arbitrary subsystems, in a mutually exclusive manner. To access a resource \( R_i \), a task must first lock the resource, and when the task no longer needs the resource it is unlocked. The time during which a task holds a lock is called a critical
where $sbf_s(t)$ is the supply bound function based on the periodic resource model presented in \cite{X} that computes the minimum possible CPU supply to $S_s$ for every interval length $t$, and $rbf_{fp}(i,t)$ denotes the request bound function of a task $\tau_i$. $sbf_s(t)$ can be calculated as follows:

$$sbf_s(t) = \begin{cases} \frac{t}{k} + (k+1)(P_s - Q_s) & \text{if } t \in V^{(k)} \\ (k-1)Q_s & \text{otherwise,} \end{cases}$$

(2)

where $k = \max(\left(\frac{(t - (P_s - Q_s))}{P_s}\right), 1)$ and $V^{(k)}$ denotes an interval $[(k+1)P_s - 2Q_s, (k+1)P_s - Q_s]$.

Note that, for Eq. (2), $t$ can be selected within a finite set of scheduling points \cite{X}. The request bound function $rbf_{fp}(i,t)$ of a task $\tau_i$ is given by:

$$rbf_{fp}(i,t) = C_i + I_S(i) + I_H(i,t) + I_L(i),$$

(3)

$$I_S(i) = \sum_{R_s \in \{R_1\}} X_{i,k},$$

(4)

$$I_H(i,t) = \sum_{\tau_j \in \Phi(i)} \left[ \frac{t}{T_j} \right] (C_j + \sum_{R_s \in \{R_1\}} X_{j,k}),$$

(5)

$$I_L(i) = \max_{\tau_j \in \Phi(i)} \left( \max_{\forall R_s \in \{R_1\}} (X_{j,j}) \right),$$

(6)

where $I_S(i)$ is the self blocking of task $\tau_i$, $I_H(i,t)$ is the interference from tasks with priority higher than that of $\tau_i$, and $I_L(i)$ is the interference from tasks, with priority lower than that of $\tau_i$, that access shared resources.

**Subsystem budget.** In this paper, it is assumed that the subsystem period is given while the minimum subsystem budget should be computed so that the system will require lower CPU resources. Given a subsystem $S_s$ and $P_s$, let $\text{calculateBudget}(S_s, P_s)$ denote a function that calculates the smallest subsystem budget $Q_s$ that satisfies Eq. (7) (the function is similar to the one presented in \cite{X}). Hence,

$$Q_s = \text{calculateBudget}(S_s, P_s).$$

(7)

**Calculating $X_s$.** Given a subsystem $S_s$, its critical section execution time $X_s$ represents the maximum time that a subsystem internal task may lock a shared resource. Note that any task $\tau_i$ accessing a resource $R_j$ can be preempted by tasks with priority higher than $rc_{j1}$. Note that SIRAP prevents subsystem budget expiration inside a critical section of a global shared resource. When a task experiences self-blocking during a subsystem budget period it is guaranteed access to the resource during the next period. A sufficient condition to provide this guarantee is

$$Q_s \geq X_s.$$  

(8)

We now derive $X_s \leq Q_s < P_s$ and since we assume that $2P_s \leq T_{\text{min}}$ then all tasks that are allowed to preempt while $\tau_i$ accesses $R_j$ will be activated at most one time from the time that self blocking happens until the end of the next subsystem period. Then $X_{i,j}$ which represents the maximum time that $\tau_i$ locks $R_j$, can be computed as follows,

$$X_{i,j} = c_{i,j} + \sum_{k = r_{c_{j1}} + 1}^{n_s} C_k.$$  

(9)

Let $X_j = \max\{X_{i,j}\}$ for all $\tau_i \in T_s$ accessing $R_j$, then $X_s = \max\{X_j\}$ for all $R_j \in R_s$.

**Internal resource ceiling.** Looking at Eq. (7), assigning internal resource ceilings according to SRP may make the value of $X_s$ very high which causes the subsystem to require more CPU resources. One way to handle this problem is by preventing the preemption inside the subsystem when a task is accessing a shared resource as proposed in \cite{X} so $X_{i,j} = c_{i,j}$. It can be implemented using SRP by assigning the resource ceiling of all resources equal to the maximum task priority $rc_{j1} = n_s$ where $n_s$ is the task ID number of the highest priority task. However, Bertogna et al. \cite{X} showed that preventing preemption while accessing a global shared resource may violate the local schedulability of the subsystem and proposed an algorithm based on increasing the ceiling of all resources in steps as much as possible without violating the local schedulability. Finally, Shin et al. \cite{X} showed that there is a tradeoff between decreasing the value of $X_s$ and the minimum subsystem budget required to guarantee the schedulability of the subsystem.

The result of this paper does not depend on any of the discussed methods to set the internal resource ceiling. So we assume that the internal ceiling of resource $R_j$ can be selected within the following range $n_s \geq r_{c_{j1}} \geq r_{c_{j1}}^{\text{LWB}}$.

5. **IMPROVED SIRAP ANALYSIS**

In this section we will show that Eq. (7) is pessimistic and can be improved such that the subsystem budget may decrease. Each task $\tau_i$ that shares a global resource $R_j$ with a lower priority task $\tau_{i'}$ can be blocked by $\tau_{i'}$ due to (i) self blocking of $\tau_{i'}$ and in addition due to (ii) access of $R_j$ by $\tau_{i'}$. The maximum blocking times of (i) and (ii) are given by the self blocking time $X_{i,j}$, and the maximum execution time $c_{i,j}$ of $\tau_i$ inside a critical section of $R_j$, respectively. Note that preemption of tasks with priority higher than $rc_{j1}$ can be excluded from the resource access of $R_j$ by $\tau_i$, because those preemptions are already incorporated in $I_H(i,t)$ (in Eq. (7)). The worst-case blocking is the summation of the blocking from these two scenarios, as shown in Eq. (7).
Since $c_{f,j} \leq X_{f,j}$, the interference $I_L(i)$ of tasks with a priority lower than that of task $\tau_i$, based on (8), is at most equal to that of (6). As a result, $r_{bffp}(i, t)$ may decrease, and the corresponding subsystem budget $Q_s$ may therefore decrease as well.

6. IMPROVED SIRAP PROTOCOL

In this section, we present a generalization of SIRAP, providing options for efficiency improvements of the protocol. First, we consider a dedicated setting for the subsystem ceiling during self-blocking. Next, we show that the efficiency of the protocol depends on both that setting and the subsystem parameters. Selecting an optimal setting is the topic of the next section.

6.1 Subsystem ceiling for self-blocking

Looking at Eq. (??), one way to reduce the subsystem budget $Q_s$ is by decreasing $r_{bffp}(i, t)$ for tasks that require highest subsystem budget. In Section ??, we have described one way to decrease $r_{bffp}(i, t)$ for higher priority tasks that share resources by decreasing $I_L(i)$. In this section we propose a method that allows for a further reduction of $I_L(i)$. According to SIRAP, when a task $\tau_i$ wants to enter a critical section of $R_j$, it first checks if the remaining budget is enough to release the shared resource before the budget expiration. If there is not enough budget remaining, then the task $\tau_i$ blocks itself and changes only the subsystem ceiling to be equal to $r_{c_j}$. This prevents the execution of all tasks $\{\tau_i\}$ that have priority higher than that of $\tau_i$ and at most equal to the ceiling of $R_j$ (i.e., $r_{c_j} \geq k > i$) that will be released after the self-blocking instance.

The new method called E-SIRAP is based on allowing tasks in $\{\tau_k\}$ to execute during the self-blocking time of $\tau_i$. This can be achieved by setting the subsystem ceiling equal to the priority of $\tau_i$ upon self blocking of task $\tau_i$ and raising the subsystem ceiling to the resource ceiling when $\tau_i$ actually accesses the resource. The main difference between SIRAP and E-SIRAP is the setting of subsystem ceiling when a task $\tau_i$ enters self blocking (wants to access a shared resource $R_j$ and there is not enough budget left). In SIRAP, the subsystem ceiling is set to $r_{c_j}$, i.e., the resource ceiling of $R_j$ (the resource that caused the self blocking). While using E-SIRAP the subsystem ceiling is set to $i$, i.e., the priority of $\tau_i$, which is at most equal to $r_{c_j}$. By choosing $i$ during self-blocking, we allow a maximum number of tasks to execute while preserving the attractive property of SRP that we can use a single stack for all tasks of a subsystem.

When using E-SIRAP, the maximum interference from lower priority tasks $I_L(i)$ will be decreased compared to Eq. (??), and can be calculated as:

$$I_L(i) = \max_{\tau_j \in \mathcal{P}(i)} \left( \max_{R_j | r_{c_j} \geq i} (c_{f,j}) \right).$$

According to the original SIRAP approach, if $\tau_i$ blocks itself, it should enter the critical section at the next subsystem budget replenishment. However, using E-SIRAP there is no guarantee that $\tau_i$ will enter the critical section at the next subsystem activation, since tasks with priority higher than that of $\tau_i$ and less than or equal to the ceiling of $R_j$ are also allowed to execute in the next subsystem activation. To guarantee that $\tau_i$ will enter its critical sections at the next subsystem budget replenishment, the subsystem budget should be big enough to include the execution of those tasks. When using E-SIRAP, the sufficient condition (??) has to be revised to:

$$Q_s \geq X_{i,j} + \sum_{k \in \{i+1, \ldots, r_{c_j}\}} C_k.$$

Hence, the minimum amount of budget needed for E-SIRAP may increase compared to SIRAP.

Since we assume that $2P_s \leq T_{min}$ then all higher priority tasks will be activated at most one time during the time $t \in [t_{rep}, t_{rep} + P_i]$, where $t_{rep}$ is the subsystem replenishment time after self blocking of task $\tau_i$.

Note that to evaluate $X_{i,j}$, Eq. (??) can be used without modification since E-SIRAP changes the behavior of SIRAP only within the self blocking time, and during the self blocking the task that caused self blocking is not allowed to access the shared resource. The only effect of using E-SIRAP is on the subsystem budget, hence efficiency can be defined exclusively in terms of $Q_s$.

Comparing Eq. (??) with Eq. (??), $I_L(i)$ may decrease significantly and that may decrease the subsystem budget. However, Eq. (??) is a stronger condition than Eq. (??), which may require a higher subsystem budget. Given these opposite forces, we conclude that E-SIRAP will not always decrease the minimum subsystem budget and therefore will not always give better results than the original SIRAP. We will illustrate this by the following example.

Example 1: Consider a subsystem $S_j$ that has three tasks and two of them share resource $R_1$ as shown in Table ??.

<table>
<thead>
<tr>
<th>Task</th>
<th>$C_j$</th>
<th>$T_i$</th>
<th>$R_j$</th>
<th>$c_{i,j}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>2</td>
<td>30</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>1</td>
<td>32</td>
<td>$R_1$</td>
<td>1</td>
</tr>
<tr>
<td>$\tau_3$</td>
<td>4</td>
<td>80</td>
<td>$R_1$</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2: Task set parameters of Example 1.

Let the subsystem period be equal to $P_s = 15$. Using the original SIRAP, we derive $X_{1} = X_{1,1} = 6$ and $Q_s = 9.34$. Using E-SIRAP, we derive $X_{1} = X_{1,1} = 6$ and $Q_s = 7$. This latter value satisfies Eq. (??), i.e., $Q_s \geq X_{1,1} + C_2 = 7$. In this case, E-SIRAP decreases the subsystem budget, hence requires less CPU resources. Conversely, for $C_2 = 5$, we derive $Q_s = 10.67$ for the original SIRAP and derive $Q_s \geq X_{1,1} + C_2 = 11$ by applying Eq. (??) for E-SIRAP. In this case, the original SIRAP outperforms E-SIRAP.

6.2 Subsystem ceiling upon self-blocking

As described in the previous section, the subsystem ceiling using E-SIRAP is equal to the priority of the task that enters
self blocking state during the self blocking time. However, using this setting for all shared resources during the self blocking of tasks may limit the performance improvement of E-SIRAP in terms of decreasing the subsystem budget as shown in the following example.

**Example 2:** Consider a subsystem $S_i$ that has four tasks as shown in Table ?? and the subsystem period $p_i = 50$.

<table>
<thead>
<tr>
<th>$T$</th>
<th>$C_i$</th>
<th>$T_i$</th>
<th>$R_i$</th>
<th>$c_{ij}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>14.7</td>
<td>100</td>
<td>$R_1, R_2, R_3$</td>
<td>0.1, 0.1, 0.1</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>5</td>
<td>250</td>
<td>$R_2$</td>
<td>4</td>
</tr>
<tr>
<td>$\tau_3$</td>
<td>5</td>
<td>300</td>
<td>$R_2$</td>
<td>3</td>
</tr>
<tr>
<td>$\tau_4$</td>
<td>10</td>
<td>500</td>
<td>$R_1$</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Table 3: Task set parameters of Example 2.

Using original SIRAP, $Q_s = 23$, $X_s = X_3 = 4$ and $rc_1 = rc_2 = rc_3 = 4$. Using E-SIRAP the minimum budget that satisfies Eq. (??) is $Q_s = 19$, however, to satisfy the condition in Eq. (??) when $\tau_1$ access $R_1$, the subsystem budget should be $Q_s \geq X_{i,1} + C_2 + C_3 + C_4 = 24.8$, when $\tau_2$ access $R_2$ then $Q_s \geq 22.7$ and for $\tau_3$ access $R_3$ then $Q_s \geq 18.7$. This means that the subsystem budget should be $Q_s = 24.8$.

If we use SIRAP setting for $R_1$ and E-SIRAP setting for the other shared resources then $Q_s = 19$ to satisfy Eq. (??) and to satisfy the condition in Eq. (??) for $\tau_1$ access $R_1$, then $Q_s \geq 0.1$, when $\tau_2$ access $R_2$ then $Q_s \geq 22.7$ and for $\tau_3$ access $R_3$ then $Q_s \geq 18.7$. This means that the subsystem budget should be $Q_s = 22.7$.

Finally, if we set the subsystem ceiling equal to 3 when $\tau_2$ block itself after trying to lock $R_2$ then $Q_s = 19$ to satisfy Eq. (??), and to satisfy the condition in Eq. (??) for $\tau_3$ access $R_3$ should be $Q_s \geq 18.7$ and $\tau_2$ the subsystem budget should be $Q_s \geq 17.7$. The subsystem budget for this case should be $Q_s = 19$.

It is clear that combining SIRAP and E-SIRAP gives better results than each alone but the last setting gives even better results (lowest subsystem budget) which is the combination of SIRAP (for $\tau_1$ access $R_1$) and E-SIRAP (for $\tau_3$ access $R_3$) and in between (for $\tau_2$ access $R_2$). However, there are two problems associated with this approach. First, each task access a shared resource should have its own setting for subsystem ceiling during the self blocking time and that means we need $n_s \times m_s$ extra memory space to save these values as a worst case. The second problem is finding the best setting of subsystem ceiling for each task access a global shared resource. In Section ?? we present an algorithm that finds the best setting of the subsystem ceiling to decrease the subsystem budget $Q_s$.

To solve the first problem, we introduce self blocking ceiling $src_j$ as the ceiling of a global shared resource $R_j$ during the self blocking time of all tasks that access this resource. The value of the self blocking ceiling should be within $src_j \in [k, rc_j]$ where $k$ is the index of the lowest priority task that access $R_j$ i.e., $k = \min\{\forall \tau_i \in T, \tau_i$ accesses $R_j\}$.

The self blocking ceiling will be used in assigning the subsystem ceiling $sc_a$ value during the self blocking, e.g., when $\tau_i$ blocks itself after failing to lock $R_j$, the following assignment takes place:

$$sc_a = \max(src_j, i). \quad (13)$$

The max function in Eq. (13) is used to prevent all lower priority tasks $\tau_j$ that have $src_j \leq k < i$, from being executed during the self blocking of $\tau_i$. One of the advantages of using self blocking ceiling is that decreases the memory space required to save the setting during the self blocking of tasks to $m_s$, however, it increases the runtime overhead since it uses the max function.

Note that if it is required to use SIRAP setting for $R_j$ then it is simply achieved by setting $src_j = rc_j$ and if it is required to use E-SIRAP instead then $src_j = k$, so using self blocking ceiling generalizes this version of SIRAP to include original SIRAP, first E-SIRAP and in between.

Eq. (??) and (??) should be changed to include the self blocking ceiling which has a great effect on them. The interference from lower priority tasks on $\tau_i$ depends on $src_j$. During self-blocking of a lower priority task $\tau_j$ that tried to access $R_j$, task $\tau_i$ is allowed to execute if $src_j < i$. Hence, $\tau_i$ will not be blocked during the self blocking of $\tau_i$ on $R_j$ when $src_j < i$. The interference from lower priority tasks can be calculated as follows:

$$I_z(i) = \max_{\forall R_j \in R_s, src_j \geq i} \left( A(i, j) \times X_{f, j} + c_{f, j} \right), \quad (14)$$

where

$$A(i, j) = \begin{cases} 0 & \text{if } src_j < i \\ 1 & \text{otherwise.} \end{cases} \quad (15)$$

$src_j$ should also be included in Eq. (??) as shown below:

$$Q_s \geq X_{i, j} + \sum_{k \in \{\max(i, src_j) + 1, \ldots, rc_j\}} C_k. \quad (16)$$

7. SELECTION ALGORITHM

In this section, we will present an algorithm that finds the best setting of the self blocking ceilings that minimize the subsystem budget $Q_s$. The algorithm searches for the best values for $\{src_j\}, \forall R_j \in R_s$ through iteration steps (see Figure ??). The algorithm is explained as follows:

**Input and output.** $S_s$, $R_s$ and $\{rc_j\}, \forall R_j \in R_s$ are the inputs to the algorithm, and the outputs from the algorithm are $Q_s$ and $\{src_j\}$.

**Initialization.** In the beginning, the algorithm sets the self blocking ceiling of resources equal to the resource ceiling (line 1 in Figure ??) which is equivalent to SIRAP. In this case, the interference from lower priority tasks will be the highest and is counted using Eq (??).
**Iteration step.** In line 4, the algorithm calculates the subsystem budget $Q_s$ and it checks the condition in Eq. (??) to guarantee the correctness of SIRAP/E-SIRAP (lines 5–6). In line 14 it finds the task $\tau_s$ that requires the highest CPU resource, that the value of $Q_s$ was selected according to the CPU resource demand of this task. Then the algorithm finds the resource $R_h$ that cause the maximum blocking on task $\tau_s$ (line 15). Finally, it sets the $src_h$ to be less than the priority of $\tau_s$ in line 19 ($src_h = h - 1$). The interference from lower priority tasks that access $R_h$, on task $\tau_s$ will be lower and will be computed according to Eq. (??) (with $A(h,b) = 0$) which decreases $rbf_{fp}(h, t)$ and can decrease the subsystem budget $Q_s$. Finally, the algorithm computes the subsystem budget after the changes of the self blocking ceiling and repeat the operation.

**Iteration termination.** The algorithm terminates if one of the following conditions becomes true:

1. The self blocking ceiling of the resource $R_h$ is lower than the priority of the task $\tau_s$ (line 16). In this case, lowering the self blocking ceiling will not decrease $Q_s$ because the maximum blocking on $\tau_s$ can not be decreased, i.e., maximum does not contain the term $X_{h,b}$ in Eq. (??).

2. If there is not a resource that block the task $\tau_s$, i.e., $I_L(h) = 0$ in Eq. (??) (for example the lowest priority task).

3. If the current budget is greater than the one that is evaluated in the previous iteration (lines 8, 12). Note that the budget may increase only due to the condition in Eq. (??). The reason is that in each iteration the self blocking ceiling of $R_h$ is decreased which can decrease the required $Q_s$ that schedule $\tau_s$. On the other hand, decreasing $src_h$ will increase the right hand side of Eq. (??) which may require higher $Q_s$, and continuing to decrease $src_h$ will increase $Q_s$ even more.

**Complexity and runtime overhead.** During an $i$-th iteration, the algorithm only decreases the self blocking ceiling of $R_h$. Then, it can repeat at most $O(n_t \times m_s)$ iterations for a subsystem that its lowest priority task accesses all shared resources and the algorithm decreased the self blocking ceiling of the shared resources to the priority of that task. During runtime, the improved E-SIRAP adds some runtime overhead compared with the original SIRAP since it uses a max function in Eq. (??) when assigning the value of the subsystem ceiling when entering self blocking state. In addition, it requires more memory to save the self blocking ceiling of shared resources compared with SIRAP as explained in section ??.

**Improvement compared to SIRAP.** The resulting $Q_s$ when using this algorithm is always less than or equal to the subsystem budget when using the original SIRAP. The algorithm initializes the self blocking ceiling according to SIRAP (i.e., $src_j = rc_j$) and it will continue iterating as long as there is a possibility to decrease $Q_s$. It stops if the value of $Q_s$ starts to increase or the algorithm can not decrease it anymore. Then we can conclude that the algorithm will give same or better results compared with SIRAP.

**Algorithm’s functions**

- The $\text{findTaskMaxQ}$ function returns the index of one task. In case there are more than one task that require at least $Q_s$ then the algorithm will handle one subsystem at each iteration and the order of handling them does not affect the results of the algorithm. The same holds for the function $\text{findResourceMaxB}$ where it returns the index of one resource and it might happen that more than one shared resource cause the maximum blocking.

- We will explain the function $\text{findTaskMaxQ}$; for each task $\tau_r$, lets define $\text{slack}_r$ as the maximum positive difference between the supply bound function and the request bound function of $\tau_r$, $\text{slack}_r = \max_{t \in [0,D_r]} (\text{sbf}_r(t) - \text{rbf}_{fp}(i,t))$ where $t$ can be selected within a finite set of points $[7]$. Then the function $\text{findTaskMaxQ} = \{ \text{such that } \text{slack}_r = \min_{t \in [0,D_r]} \text{slack}_s \}$

**Example.** We will explain the operation of the algorithm using the example in Table ??.

1. First, the algorithm initializes the values of self blocking such that $src_1 = src_2 = src_3 = 4$, then it finds the minimum budget required to guarantee the schedulability $Q_s = 23$. At line 14, it finds the task that requires maximum CPU resources which is $\tau_4$. It tries to decrease the $rbf_{fp}(4,t)$ by decreasing the interference from lower priority tasks, looking at Eq. (??) at this step, $A(4,1) = A(4,2) = A(4,3) = 1$, i.e., the maximum blocking from each shared resource. At line 15 the algorithm finds $R_3$ as the shared resource that imposes the maximum blocking on $\tau_4$ so it decreases the self blocking ceiling of $R_3$ such that $src_3 = 3$ which makes $A(4,3) = 0$.

2. The algorithm calculates the new budget $Q_s = 21$ and check the condition in Eq. (??) which is $Q_s \geq 18.7$ so final$Q_s = 21$. It finds the task that requires maximum CPU resources, and it is task $\tau_4$. After that its finds the resource that imposes maximum blocking which is $R_2$. Then it sets $src_2 = 3$ and by this $A(4,2) = 0$ in Eq. (??).

3. The new subsystem budget will be $Q_s = 19$ and the condition in Eq. (??) then $Q_s \geq 18.7$, final$Q_s = 19$. After that the algorithm finds the task that requires maximum CPU, and it is still task $\tau_4$, and it finds that $R_2$ is imposing the maximum blocking. But $src_3 < 4$ (at line 16) the blocking from this resource is the minimum and can not be minimized more (case 1 in the iteration termination). So the algorithm stops and returns the subsystem budget final$Q_s = 19$ and the
We will explain the effect of the mentioned parameters by means of the simulation in the following section.

8.1 Simulation settings

The simulation is performed by applying the algorithm on 1000 different randomly generated subsystems where each subsystem consists of 5 tasks, and then we have increased the number of tasks to 8 tasks to investigate the effect of changing the number of task on the algorithm performance. The internal resource ceilings of the globally shared resources are assumed to be equal to the highest task priority in each subsystem (i.e., \( r_{Cj} = n_s \)) and we assume that \( T_i = D_i \) for all tasks. For the subsystems that contain 8 tasks, 2-6 tasks access globally shared resources and 1-4 tasks access global shared resources for the subsystems that contains 5 tasks. The worst-case critical section execution time of a task \( \tau_i \) is set to a value between 0.3\( C_i \) and 0.8\( C_i \). A task is assumed to access at most one globally shared resource. For each simulation study the following settings are changed and a new 1000 subsystems is generated except when changing the subsystem period where the same subsystems are used:

1. Number of tasks – the number of tasks in subsystems.
2. Task set utilization \( U_s \) – the task set utilization is the summation of the utilization of all tasks in the subsystem, is specified to a desired value.
3. The subsystem period – the subsystem period is specified to a desired value.

The task set utilization is divided randomly among the tasks that belong to that subsystem. Task periods are selected within the range of 200 to 1000. Since the task period is generated to a value within the interval as specified, the execution time is derived from the desired task utilization. All randomized subsystem parameters are generated following uniform distributions.

8.2 Simulation results

Tables ??-?? show the results of 3 different simulation studies performed to measure the performance of the algorithm. The tables present four main measures. Firstly, the percentage of subsystems for which the subsystem budget decreased when the algorithm was applied is presented in the row labeled by ”Improvement”. Secondly, the absolute improvement, i.e. decrement, \( Q_s^{Dec} \) of the subsystem budget is computed, which is defined as \( Q_s^{Dec} = Q_s^{SIRAP} - Q_s^{alg} \), where \( Q_s^{SIRAP} \) and \( Q_s^{alg} \) are the subsystem budget using SIRAP and using the selection algorithm, respectively. The tables present both the average decrement and the maximum decrement in rows labeled by ”Avg. \( Q_s^{Dec} \)” and ”Max. \( Q_s^{Dec} \)”, respectively. Thirdly, the relative improvement of the subsystem utilization \( U_{sImp} \) is computed, which is defined as

\[
U_{sImp} = \frac{(U_s^{SIRAP} - U_s^{alg})}{U_s^{SIRAP}},
\]

where \( U_s^{SIRAP} = Q_s^{SIRAP} / P_s \) and \( U_s^{alg} = Q_s^{alg} / P_s \) denote the subsystem utilization using SIRAP and using the selection algorithm, respectively. Similar to the improvement of the subsystem budget, the tables present both the average decrement (”Avg. \( U_{sImp} \)”) and maximum decrement (”Max. \( U_{sImp} \)”) of the subsystem utilization. Finally, the maximum

---

8. ALGORITHM EVALUATION

In this section, we evaluate the performance of the presented algorithm, in terms of requiring less CPU resource than using original SIRAP.

Based on Eqs. (??), (??), (??) and (??), we can distinguish two parameters that have great effect on the performance of the algorithm:

- \( X_{i,j} \) – since the algorithm decreases the interference of the lower priority tasks by \( X_{i,j} \) compared with SIRAP (see Eq. (??)) then higher values of \( X_{i,j} \) can decrease the subsystem budget more.

- The difference between \( P_s \) and \( T_{min} \) – the lower the difference is the better results the algorithm will give. The reason behind this is that if \( P_s \) is much lower than \( T_{min} \), then the subsystem budget using SIRAP will be lower and because of the condition in Eq. (??) the algorithm may not be able to decrease the subsystem budget.

---

Figure 2: The selection algorithm.

self blocking ceilings of the shared resources \( src_1 = 4, src_2 = 3, src_3 = 3 \).
number of iterations that the algorithm needed to find the lowest subsystem budget is determined and presented in the row labeled by "Max. iterations".

- **Study 1** is specified having task utilizations $U^T_s$ of 5%, 10% and 20%, number of tasks $n_s$ equals to 5, task periods between 200 and 1000, and subsystem period $P_s$ is 100.

- **Study 2** changes the subsystem period $P_s$ (compared to Study 1) to 75, 70 and 65 and keeps $U^T_s = 5\%$. As mentioned previously we use the same 1000 subsystems in Study 1 that have $U^T_s = 5\%$ and only change the subsystem period.

- **Study 3** increase the number of tasks (compared to Study 1) to 8 tasks.

Looking at the results in Table ??, it is clear that for some subsystems the algorithm can decrease the required budget significantly (a maximum decrease "Max. $Q_s^{Dec}$ of 17.4 and maximum relative subsystem utilization improvement $Max. U_s^{Imp}$ of 35\%). It also shows that increasing $U^T_s$ decreases the number of subsystems for which the algorithm can improve their budgets compared with SIRAP. The reason is that increasing $U^T_s$ will increase $C_i$ of the tasks and will increase the required budget that satisfy Eq. (??). However, it will also increase $c_{i,j}$ which is clear from observing the "Avg." and "Max." rows in the table.

Looking at Table ???, it is clear that when the subsystem period is decreased, the number of the subsystems that the algorithm can improve will be decreased. However, the decrement in the subsystem budget will be more significant for the subsystem utilization when the subsystem period is lower since $U_s = Q_s / P_s$ (see the "Ave." and "Max." rows in Table ??). For the case $P_s = 75$ when $P_s = 65 < T_{min}/3$ then the algorithm can not improve any subsystem as explained in the beginning of this section. This can be seen as a limitation of the algorithm and it could be better to decrease the subsystem period instead of using the algorithm to decrease the subsystem utilization $U_s = Q_s / P_s$. However, this is not always true, first, from the simulation results we have compared the subsystem utilization $U_s$ when $P_s = 100$ and $P_s = 65$ and we have found that when $P_s = 100$, 97 out of 410 subsystems that the algorithm improved, require less or equal subsystem $U_s$ than when using only SIRAP with $P_s = 65$. The reason is that when the algorithm is able to improve the subsystem budget, then the request bound function of the task that needs maximum CPU resources will be lower than the case when using original SIRAP with lower subsystem period, and this affects Eq. (??). The second issue of reducing the subsystem budget is that it increases the context switch overhead because the subsystem budget will be lower. Finally, as showed in [?] decreasing the subsystem period may increase $U_s$ to satisfy the condition $Q_s \geq X_s$ of SIRAP.

In Study 3 we have increased the number of tasks to 8 tasks in each subsystem, and the results in Table ?? shows that increasing the number of tasks does not change the effect of $U^T_s$. However, increasing the number of tasks will increase the number of subsystems that the algorithm can improve (compare the "Improve" row in Table ?? and Table ??). The reason is that there are more task and more shared resources in each subsystem so the algorithm can improve the subsystem more before it stops. The maximum number of iterations for 8 tasks is 6 iterations while for 5 tasks it is 4. In addition, increasing the number of tasks and keeping $U^T_s$ the same, decreases the utilization of each task which improves the algorithm performance in terms of the number of the subsystems that the algorithm can improve as explained previously.

<table>
<thead>
<tr>
<th>Improve</th>
<th>$U^T_s=5%$</th>
<th>$U^T_s=10%$</th>
<th>$U^T_s=20%$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. $Q_s^{Dec}$</td>
<td>2</td>
<td>3.7</td>
<td>5.3</td>
</tr>
<tr>
<td>Avg. $U_s^{Imp}$</td>
<td>16.4%</td>
<td>15.3%</td>
<td>12.4%</td>
</tr>
<tr>
<td>Max. $Q_s^{Dec}$</td>
<td>5</td>
<td>10</td>
<td>17.4</td>
</tr>
<tr>
<td>Max. $U_s^{Imp}$</td>
<td>36.6%</td>
<td>37.2%</td>
<td>35%</td>
</tr>
<tr>
<td>Max. iterations</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 4: Measured results of Study 1.

<table>
<thead>
<tr>
<th>Improve</th>
<th>$P_s=75$</th>
<th>$P_s=70$</th>
<th>$P_s=65$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. $Q_s^{Dec}$</td>
<td>1.9</td>
<td>0.55</td>
<td>0</td>
</tr>
<tr>
<td>Avg. $U_s^{Imp}$</td>
<td>20%</td>
<td>8%</td>
<td>0%</td>
</tr>
<tr>
<td>Max. $Q_s^{Dec}$</td>
<td>5</td>
<td>1.2</td>
<td>0</td>
</tr>
<tr>
<td>Max. $U_s^{Imp}$</td>
<td>41.9%</td>
<td>16.6%</td>
<td>0%</td>
</tr>
<tr>
<td>Max. iterations</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5: Measured results of Study 2.

<table>
<thead>
<tr>
<th>Improve</th>
<th>$U^T_s=5%$</th>
<th>$U^T_s=10%$</th>
<th>$U^T_s=20%$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. $Q_s^{Dec}$</td>
<td>2</td>
<td>3.3</td>
<td>5</td>
</tr>
<tr>
<td>Avg. $U_s^{Imp}$</td>
<td>14.3%</td>
<td>13.6%</td>
<td>11%</td>
</tr>
<tr>
<td>Max. $Q_s^{Dec}$</td>
<td>5</td>
<td>10</td>
<td>14.6</td>
</tr>
<tr>
<td>Max. $U_s^{Imp}$</td>
<td>34.5%</td>
<td>35.4%</td>
<td>33.5%</td>
</tr>
<tr>
<td>Max. iterations</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 6: Measured results of Study 3.

9. SUMMARY
In this paper, we presented an improved schedulability analysis for the synchronization protocol SIRAP. The improved analysis may decrease the minimum subsystem budget while still guaranteeing the schedulability of all tasks in a subsystem. We also presented a generalization of SIRAP, which distinguishes separate resource ceilings for self blocking and for actual resource access, with the aim to reduce the required CPU resource for each subsystem by reducing the interference from lower priority tasks. Because the efficiency of the protocol depends on both the setting of the resource ceilings and the subsystem parameters, we presented an algorithm that finds the best settings for resource ceilings during the self blocking for each shared resource in order to minimize the required subsystem budget. The simulation results shows that the algorithm can significantly reduce the CPU resource needs of a subsystem, but that the effectiveness of the algorithm heavily depends on the tasks parameters and the subsystem period.
Our future work includes further improvements of SIRAP in two directions: I) Applying runtime mechanisms to decrease the value of $X_{i,j}$ that is used to check if there is enough remaining budget before accessing a shared resource, based on the arrival time of the higher priority tasks, in order to improve the average response time of tasks. II) Investigating the case of allowing lower priority tasks to execute during the self blocking in order to reduce the interference from higher priority tasks. For this improvement, a runtime mechanism may be required to decide the correct execution order of tasks during the next activation subsystem period. III) Finally, showing the advantages of the proposed algorithm on real industrial systems.