Extended Support for Limited Preemption Fixed Priority Scheduling for OSEK/AUTOSAR-Compliant Operating Systems

Matthias Becker*, Nima Khalilzad*, Reinder J. Bril†*, Thomas Nolte*
*MRTC / Mälardalen University, Västerås, Sweden
{matthias.becker, nima.m.khalilzad, thomas.nolte}@mdh.se
†Technische Universiteit Eindhoven, Eindhoven, Netherlands
r.j.bril@tue.nl

Abstract—Fixed Priority Scheduling (FPS) is the de facto standard in industry and it is the scheduling algorithm used in OSEK/AUTOSAR. Applications in such systems are compositions of so-called runnables, the functional entities of the system. Runnables are mapped to operating system tasks during system synthesis. In order to improve system performance it is proposed to execute runnables non-preemptively while varying the tasks threshold between runnables. This allows simpler resource access, which can reduce the stack usage of the system and improve the schedulability of the task sets. FPS*, as a special case of fixed-priority scheduling with deferred preemptions, executes subjobs non-preemptively and preemption points have preemption thresholds, providing exactly the proposed behavior. However OSEK/AUTOSAR-conform systems cannot execute such schedules. In this paper we present an approach allowing the execution of FPS* schedules. In our approach we exploit pseudo resources in order to implement FPS*. It is further shown that our optimal algorithm produces a minimum number of resource accesses. In addition, a simulation-based evaluation is presented in which the number of resource accesses as well as the number of required pseudo-resources by the proposed algorithms are investigated. Finally, we report the overhead of resource access primitives using our measurements performed on an AUTOSAR-compliant operating system.

I. INTRODUCTION

Since the seminal work of Liu and Layland [1], Fixed-Priority Scheduling (FPS) has been widely studied in the real-time scheduling community. Fully preemptive FPS provides a higher ratio of schedulable systems than non-preemptive FPS. However, fully preemptive FPS increases the memory requirement of systems and it introduces preemption overheads. Also, it complicates the software development. This is because, in fully preemptive scheduling, preemptions can occur at any given time. Thus, access to shared resources becomes non-trivial. While, non-preemptive FPS does not have the aforementioned problems. In order to combine the benefits of the two approaches, two schemes have been proposed for FPS with limited preemption. Tasks may increase their priorities after acquiring the CPU. Therefore, fewer tasks will be able to preempt the running task. This approach is known as fixed-priority scheduling with deferred preemptions (FPDS) [8], [9], [10].

Both models are a generalization of FPS, where the two improvements are orthogonal to each other. In [11], Bril et al. combined the ideas of FPTS and FPDS into a general form FPVS, and later in [12] the authors refined the model to fixed-priority scheduling with varying preemption thresholds (FPVS). In the same work FPDS* is presented. It is a specialization of FPVS where tasks are only allowed to be preempted at specific points known as preemption points. The preemption points may have higher priorities than the tasks’ original priorities at dispatch.

A. Motivation for using FPDS*

In the following we present two examples to demonstrate the benefits of limited preemptive fixed-priority scheduling techniques. Next, we show that FPVS (generalizing both FPTS and FPDS) and FPDS* have a higher ratio of schedulable task sets than FPTS and FPDS.

Example 1. The task set in the first example consist of two tasks $\tau_1$ and $\tau_2$. The characteristics of the task set used in this example is presented in Table I. The worst-case response time of tasks using preemptive FPS (FPPS), non-preemptive FPS (FPNS) and FPDS are denoted using $WR^P_i$, $WR^N_i$, and $WR^D_i$ respectively. This task set is not schedulable by either FPNS or FPPS. Figure 1 depicts the execution trace of the task set under FPDS. The arrows mark the release of a job, and the gray colored boxes highlight the executing jobs. If $\tau_2$ is divided into two subjobs, then the task set becomes indeed schedulable with FPDS, since the second subjob of $\tau_2$ cannot be preempted by $\tau_1$. Let us consider scheduling the task set with FPTS. We can either execute the jobs of $\tau_2$ with a threshold level equal to the priority of $\tau_2$ or equal to the priority of $\tau_1$. The resulting schedule is then equivalent to FPSS or FPNS respectively. In both cases the task set is not schedulable (see Table I). Thus FPTS does not dominate FPDS.

Example 2. In this example we consider a task set consisting

<table>
<thead>
<tr>
<th>$T_i = D_i$</th>
<th>$C_i$</th>
<th>$\pi_i$</th>
<th>$WR^P_i$</th>
<th>$WR^N_i$</th>
<th>$WR^D_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

The research leading to these results has received funding from the Swedish Research Council (Vetenskapsrådet), the Swedish Knowledge Foundation (KKS), and the ARTEMIS Joint Undertaking project EMC2 (grant agreement 621429).
of four tasks \( \{\tau_1, \ldots, \tau_4\} \) presented in Table II. In contrast to the first example, in this example the deadlines are not equal to the periods. The task set is schedulable by FPTS and the preemption threshold of a task \( \tau_i \) is represented by \( \Theta_i \). Figure 2 depicts a possible execution trace, where \( \tau_4 \) is released just before all other tasks. Since the preemption threshold of \( \tau_4 \) is equal to the priority of \( \tau_3 \), \( \tau_3 \) cannot preempt \( \tau_4 \) and it experiences its worst-case response time. However, it still stays schedulable. Both FPNS and FPPS can not make the task set schedulable (see Table II). Since \( \tau_1 \) has its deadline equal to its execution time it is easy to see that any attempt to make the task set schedulable using FPDS will result in a deadline miss of \( \tau_1 \). Therefore we can conclude that FPDS does not dominate FPTS.

The above two examples show that neither of FPTS and FPDS algorithms dominates the other. Therefore, it is interesting to see the average success ratio of each algorithm. To this end, we present the simulation results published by Bril et al. in [12]. Figure 3 depicts the schedulability ratio of randomly generated task sets with deadlines equal to periods. The figure clearly shows the improvements of FPTS and FPDS over the traditional fixed priority scheduling schemes. Additionally, the figure illustrates that FPDS* and FPVS have a higher success ratio in comparison to all other fixed-priority algorithms considered in the experiment. In order to make use of this high schedulability ratio in OSEK/AUTOSAR-compliant operating systems, in this paper, we present an approach to implement the FPDS* algorithm.

B. Support for FPDS* in OSEK/AUTOSAR

AUTOSAR [13] is the de facto standard for automotive electric/electronic architectures. The real-time Operating System (OS) of AUTOSAR is based on the OSEK standard [14]. Software applications in AUTOSAR consist of software components which in turn are composed out of runnables, where a runnable is a functional entity. During the system synthesis all runnables are mapped to tasks of the system which are then scheduled by FPS. Additionally, AUTOSAR provides the concept of internal resources which allows tasks to raise their priority during the execution of jobs. Thus, the implementation of non-preemptive groups [2] and a limited version of preemption thresholds is possible. Recently, Zeng et al. [15] proposed several algorithms targeting the design synthesis of AUTOSAR systems with focus on minimizing the stack memory usage. They provide algorithms to assign runnables to tasks and they also assign thresholds to runnables. A runnable, a functional entity, could be considered non-preemptable. This allows further reduction of stack space and simple resource access without the need for complex resource access protocols [15], [16]. However, more elaborate scheduling schemes, such as FPDS*, which could exploit the characteristics of such task models cannot be straightforwardly implemented in an OSEK-compliant operating system. In [17] Gai et al. proposed exploiting resource access protocols for assigning preemption thresholds to tasks. They observed that the Stack Resource Policy (SRP) [18] can be used to raise the priority of a task to the respective ceiling of the resource. This, in turn, is equal to raising the tasks’ priorities to a threshold. Thus, task priorities can be dynamically changed by locking appropriate resources at right times. Note that this technique is not exclusive for SRP. The Immediate Priority Ceiling Protocol (IPCP) [19] combined with FIFO execution of tasks with the same priority can also be used for this purpose.

In this paper, we investigate implementing FPDS* in an OSEK-compliant OS. Two algorithms are presented. Both algorithms use pseudo-resources [17] in combination with a resource access protocol to adapt the task priorities at runtime. Both algorithms generate the necessary calls to the resource
access primitives in order to raise the tasks priority to the respective threshold values. We prove that our first algorithm always results in an OSEK-conform sequence of resource lockings while resulting in desired priority values. The second proposed algorithm is an extension of the first algorithm which generates a minimum number of resource accesses.

II. BACKGROUND

This section describes the theoretic foundation assumed in [12] as well as the necessary background knowledge about scheduling and resource access protocols within an OSEK/AUTOSAR-compliant OS.

A. Task model

We assume a task set \( T = \{\tau_1, \ldots, \tau_N\} \), consisting of \( N \) independent tasks \( \tau_i \). Tasks are scheduled on a single processor. A task \( \tau_i \) has a worst-case execution time \( C_i \) and arbitrary deadline \( D_i \). We assume arbitrary phasing. Tasks are activated periodically with a period \( T_i \). Thus a task is described by the following tuple \( \{C_i, D_i, T_i\} \). Each task consists of \( m_i \) subjobs, where \( m_i \geq 1 \). We denote the \( a \)-th subjob of a task \( i \) with \( \tau_{i,a} \). Each task has a priority \( \pi_i \). For our convenience we assume priorities equal to task indices. To coincide with the OSEK/AUTOSAR standard [14], a higher value constitutes a higher priority, with \( N \) as the highest task priority in the system.

FPDS\(^*\), as described by Bril et al. in [12], executes all subjobs in a run-to-completion manner. Preemption is allowed between subjobs at so called preemption points. These points are defined by the system developer at design time. We use natural numbers in the range \([0, m_i]\) for referring to the preemption points. The \( a \)-th preemption point, with \( 0 < a < m_i \), is located between subjob \( \tau_{i,a} \) and \( \tau_{i,a+1} \). Point zero is the task start while the end of task is point \( m_i \). Each preemption-point has a preemption threshold \( \theta_{i,a} \), where \( \pi_i \leq \theta_{i,a} \leq \pi_N \), and \( \theta_{i,0} = \theta_{i,m_i} = \pi_i \).

B. OSEK/AUTOSAR

We use the OSEK operating system description [14] as the basis of our work. This is sufficient since AUTOSAR is a superset of OSEK [13]. OSEK only allows static priorities to improve the efficiency. Priority boosting may only happen by the resource sharing protocols [14]. The priorities can be divided into three groups: (i) interrupts with the highest priority; (ii) scheduling priorities that are below the interrupt priorities (iii) user task priorities that are lower than the scheduling operations. According to the specification, three possible scheduling models are supported.

1) Fixed Priority Fully-Preemptive Scheduling;
2) Fixed Priority Non-Preemptive Scheduling;
3) Fixed Priority Mixed Preemptive Scheduling.

All three scheduling schemes assume fixed priorities, assigned at design time. For the fixed priority non-preemptive scheduling AUTOSAR prohibits calls to reschedule the tasks while a resource is held. Therefore we limit our approach to the fixed priority fully-preemptive scheduling model.

The scheduler searches all tasks in the run and ready queues, where it selects the tasks with the highest priority.

For tasks with a same priority, ties are broken based on the age of the oldest job. The age of preempted jobs is based on their first activation time, whereas the age of jobs released after a wait() command is based on their reactivation time. In order to avoid problems such as priority inversion and deadlocks, resource sharing mechanisms are supported by the standard. The Immediate Priority Ceiling Protocol (IPCP) [19] is implemented in an adapted version by the operating system [14]. The standard defines the OSEK Priority Ceiling Protocol (OPCP) as follows:

1) Tasks may only access resources that are defined in their task descriptors. The runtime engine assigns a ceiling priority to a resource. The ceiling priority is statically assigned the priority of the highest priority task accessing the resource.
2) If a task requests a resource, its priority is raised to the ceiling priority of the resource.
3) If the resource is released again the priority of the task is changed to the priority it had before the resource usage.

OSEK also provides the scheduler as lockable resource. By locking the scheduler, tasks can create non-preemptive regions. Therefore, the OSEK standard supports FPDS. The OSEK standard provides the following two functions for accessing shared resources: (i) GetResource(\( r_i \)) which locks resource \( r_i \); (ii) ReleaseResource(\( r_i \)) which is used for unlocking \( r_i \). The OSEK standard also defines how resources should be nested, if nesting is necessary. If a second critical section needs to be entered, the critical sections must be nested. The case shown in Figure 4a where the critical sections overlap is therefore called improper nesting and it is not allowed. Nesting shown as in Figure 4b is proper nesting and it is allowed.

Lemma 1. In order for task \( \tau_i \) to be able to elevate its priority at preemption points and return back to its nominal priority \( \pi_i \) at the end of each job, it has to access the pseudo-resources through proper nesting (as recommended by the standard).

Proof: The OPCP protocol is implemented as follows. When task \( \tau_i \) locks a resource \( r_j \), first the resource stores...
the active priority of the task \((r_j \rightarrow \text{old_task_prio} = \tau_i \rightarrow \text{activePriority})\). Thereafter, the active priority of the task is set to the ceiling of the resource \((\tau_i \rightarrow \text{activePriority} = r_j \rightarrow \text{ceiling_priority})\). After releasing the resource, the active priority of the task is set back to the old priority stored in the resource \((\tau_i \rightarrow \text{activePriority} = r_j \rightarrow \text{old_task_prio})\).

After releasing the resource, the active priority of the task is set to the ceiling of the resource \((\tau_i \rightarrow \text{activePriority} = r_j \rightarrow \text{ceiling_priority})\). After releasing the resource, the active priority of the task is set back to the old priority stored in the resource \((\tau_i \rightarrow \text{activePriority} = r_j \rightarrow \text{old_task_prio})\).

The proof follows from the implementation of the PCP protocol.

Suppose we have improper nesting, i.e., task \(\tau_i\) first locks resource \(r_j\) then \(r_k\), while it first releases \(r_j\) and then \(r_k\).

After locking we have: \(r_j \rightarrow \text{old_task_prio} = \tau_i,\ r_k \rightarrow \text{old_task_prio} = j\) and \(\tau_i \rightarrow \text{activePriority} = k\). After unlockings we will have: \(\tau_i \rightarrow \text{activePriority} = j\). While the task has to return back to its nominal priority \(\tau_i\) after releasing all of its resources.

A. Deriving the required pseudo-resources

In order to have the pseudo-resources needed to lift a task’s priority to the threshold given by the FPDS\(^*\) schedule, we define a set \(R\) containing one pseudo-resource for each threshold priority used by the tasks in \(T\), where \(r_p\) denotes the pseudo-resource with ceiling \(p\). Each task \(\tau_i \in T\) uses a subset of \(R\). Therefore, for elevating the threshold at preemption point \(a\) we need to lock \(r_{\theta^a_i}\). On the other hand, the operating system determines the ceiling of a resource based on the tasks that are using such a resource. Therefore, the resource usage of \(r_p\) is inserted to the declaration of the following tasks:

- \(\tau_p\), because we want the ceiling priority of \(r_p\) to be assigned to \(\pi_p\), i.e., \(r_p = \pi_p\).

Algorithm 1 finds the different pseudo resources and it adds the necessary entries to the task declaration. The algorithm iterates over all tasks in \(T\) (Line 3), and it further iterates over all preemption points of that task (Line 4). It may be the case that \(\tau_i\) is the only task with a preemption point with a preemption threshold equal to \(\pi_i\). In this case we do not need to add \(r_i\) to \(R\), which we check in Line 5. In Line 6 the algorithm checks if the resource with ceiling value equal to the current preemption point does not yet exist in \(R\). It is assumed that each resource can only be added once. If the resource is not already in \(R\), then the algorithm generates a new pseudo-resource. This resource is added to \(R\) and an entry is added to the declaration of the task with priority equal to the desired ceiling of that resource. Finally the resource is added to the declaration of the current task. Let \(M = \max_{\tau_i, m_i} T\). The algorithm has a complexity of \(O(N \times M)\) and it is executed offline.

Algorithm 1: FindPseudoResources

\begin{align*}
1: & \quad \textbf{function} \quad \text{FindPseudoResources} (T) \\
2: & \quad \quad R \leftarrow \emptyset \\
3: & \quad \quad \text{for } \forall \tau_i \in T \text{ do} \\
4: & \quad \quad \quad \text{for } \theta^a_i \mid 1 < a < m_i \text{ do} \\
5: & \quad \quad \quad \quad \quad \text{if } \theta^a_i \neq \pi_i \text{ then} \\
6: & \quad \quad \quad \quad \quad \quad \text{if } \exists r_{\theta^a_i} \in R \text{ then} \\
7: & \quad \quad \quad \quad \quad \quad \quad \quad \text{New resource } r_{\theta^a_i} \text{ and } R \leftarrow R \cup \{r_{\theta^a_i}\} \\
8: & \quad \quad \quad \quad \quad \quad \quad \quad \text{Add resource to } r_{\theta^a_i} \\
9: & \quad \quad \quad \quad \quad \quad \quad \quad \text{end if} \\
10: & \quad \quad \quad \quad \quad \quad \quad \quad \text{Add resource to } r_{\theta^a_i} \\
11: & \quad \quad \quad \quad \quad \quad \quad \quad \text{end if} \\
12: & \quad \quad \quad \quad \quad \quad \quad \quad \text{end for} \\
13: & \quad \quad \quad \quad \quad \quad \quad \text{end for} \\
14: & \quad \quad \quad \quad \quad \quad \text{end function}
\end{align*}

B. Runtime mechanism

The preemption point configurations of tasks are static and they do not change during execution time. Information on what resources need to be locked/unlocked in order to boost the tasks priority as defined by the FPDS\(^*\) schedule can thus be provided in a table for the runtime access. We use a two dimensional array of size \([m_i \times (N-i)]\) to store all information needed for a task \(\tau_i\), where \(m_i\) represents the number of preemption points and the second dimension represents the maximum possible number of calls to the resource sharing primitives for either GetResource or ReleaseResource.

A task with priority \(\pi_i\) has \(N-i\) higher priority tasks with possible pseudo-resources of same priority. Therefore, at most \(N-i\) resources can be locked or unlocked at a given
time. We use two of the described arrays per task: \( L^R \) for all calls to ReleaseResource; and \( L^G \) for all calls to GetResource. If \( L^G[a,p] \) is marked true, then \( \tau_i \) has to obtain \( r_p \) at preemption point \( a \).

Task \( \tau_i \), at each preemption point \( a \), calls \( Yield(a, i) \) (Algorithm 2). The \( Yield \) function works as follows. At a preemption point \( a \) of task \( \tau_i \) the algorithm releases all resources specified in column \( a \) of \( L^R \) followed by acquiring all resources specified in column \( a \) of \( L^G \). The algorithm thus has a complexity of \( O(N - i) \) for each task \( \tau_i \).

Algorithm 2: \( Yield \) function

1: function \( Yield(a, i) \)
2: \( index = 0 \)
3: while \( index \leq m_i \) do
4: \( \text{if } L^R[a, index] = \text{true} \) then
5: \( \text{ReleaseResource} \left( L^R[a, index] \right) \)
6: \( \text{end if} \)
7: \( index++ \)
8: end while
9: \( index = 0 \)
10: while \( index \leq m_i \) do
11: \( \text{if } L^G[a, index] = \text{true} \) then
12: \( \text{GetResource} \left( L^G[a, index] \right) \)
13: \( \text{end if} \)
14: \( index++ \)
15: end while
16: end function

IV. Functions for Initializing the Lists

In this section we present two algorithms for initializing \( L^R \) and \( L^G \). The proposed algorithms work on each task separately. Therefore, in the description of the initialization algorithms we drop index \( i \) when referring to the parameters of \( \tau_i \).

We propose a two stack model (per task) to prevent improper nesting (Figure 6). The first stack, referred as unlocked stack \((U)\), stores all unlocked resources, while the second stack \((L)\) keeps track of locked resources (locked stack). Initially, all pseudo-resources belonging to the task are in the unlocked stack. The resources are sorted by decreasing priority. Therefore, for all resources with priority lower than \( r \) \( i \), the preemption threshold \( \theta^*_n \) decreases. Since this decrease can not be done ahead, otherwise we would lower the preemption threshold at \( a - 1 \), this is done at preemption point \( a \). We call \( DecreasePriority \) (Line 6) until the head of the locked stack is equal to \( \theta^*_n \) or it is empty (Line 7). Note that when the head of the stack is empty we have reached the base priority of \( \tau_i \). Now we reached the desired preemption threshold for point \( a \). Note that after releasing the scheduler resource, we execute at a higher preemption threshold until we release the respective resources.

Case 1: when the priority of the next preemption point \( \theta^*_{a + 1} \) is larger than the current point \( \theta^*_n \) (Line 11). Note that we first need to check if such a preemption point exists or if we are at the last point (Line 10). The function proceeds by calling \( IncreasePriority \) until the head of the locked stack is equal to \( \theta^*_{a + 1} \). If the next preemption point is equal to the current one we do not need to take any actions.

Theorem 1. Using the \( InitLists \) function described in Algorithm 3 guarantees correct implementation of \( FPDS^* \).

Proof: We must prove the following two properties for the algorithm. (i) The algorithm results in having a proper resource lock for all preemption points, i.e., \( \forall p \in (0, m), r^p_\theta \) has to be
kept locked at preemption point $p$; (ii) the algorithm respects proper nesting. Except the scheduler resource, all other resources are accessed through the $\text{IncreasePriority}$ and $\text{DecreasePriority}$ functions. The scheduler resource is always the last resource to be locked, and it is the first resource to be unlocked. Therefore, property (ii) is proven by Lemma 2. For proving property (i) we show that the function always obtains $r_{\theta^*_a}$ before preemption point $p$, and it releases all resources with priority higher than the current threshold ($\theta^*_a$) at point $p$. Line 11 to 15 guarantee that $r_{\theta^*_a}$ is always locked before reaching point $p$. Line 3 to 8 guarantee that the priority will go down until we reach the current priority level.

B. InitLists’ function

Locking and unlocking resources incur overhead costs. Thus, it is desirable to minimize the number of resource calls in order to reduce the introduced overhead. In order to minimize the number of lockings (and therefore the number of unlocks), we investigate the conditions in the $\text{InitLists}$ function that results in resource lockings. The function only locks a pseudo-resource if the next preemption point has a higher threshold than the current one. In such a case, all resources $r_p \in R_t$ with a resource ceiling between $\theta^*_a$ of the current point $a$ and $\theta^*_{a+1}$ of the next point $a+1$ are locked by $\text{InitLists}$. However, it may be unnecessary to lock all priority levels. Therefore, in the following first we define a preemption point interval which has to be considered when initializing List $L^G$ for the current preemption point $a$. Thereafter, we consider different scenarios in which locking is unnecessary.

We define an interval, referred as $\text{hill}$, for the current point $a$ starting from $a$ and ending in $b$, where $\theta^*_a \leq \theta^*_b$ and all points between $a$ and $b$ have a threshold higher than $\theta^*_b$. We also define function $\text{hill}(a)$ which returns the hill interval of its input preemption point. For instance, Figure 7a illustrates a hill interval starting from $a$ and ending at $a+4$. In order to determine unnecessary lockings it is enough to investigate the hill interval. This is because all obtained locks corresponding to the priority levels between $\theta^*_a$ and $\theta^*_{a+1}$ have to be unlocked at point $b$ ($\theta^*_b \leq \theta^*_a$). Therefore, if a priority level is not used in the hill interval, then we can avoid locking its corresponding resource.

The following four scenarios, depicted in Figure 7, may lead to locking a resource when $\theta^*_{a+1} > \theta^*_a$.

a) If the next preemption point $a+1$ has a threshold level one priority higher than the current preemption point $a$, i.e. $\theta^*_{a+1} = \theta^*_a + 1$. The function locks the resource in order to obtain the correct threshold level at $a+1$.

b) The threshold of the next preemption point $a+1$ is two or more priority levels higher than the current priority threshold $\theta^*_a$, i.e. $\theta^*_{a+1} \geq \theta^*_a + 2$. However, the intermediate priority levels are used by later preemption points, e.g. $a+2$ and $a+3$. Since proper nesting must always be respected, the function locks the intermediate resources at point $a$, before locking $r_{\theta^*_{a+1}}$.

c) This scenario is similar to (b), except that the intermediate priority levels are not used by a preemption point within the following interval: from point $a$ to the first preemption point with threshold smaller or equal to $\theta^*_a$. If the intermediate resource is locked at point $a$, it needs to be unlocked latest at point $a+4$. This is because the ceiling of the intermediate resource is higher than the threshold level of point $a$. Therefore, locking the intermediate resource is unnecessary.

d) In this scenario, all intermediate priority levels are used. But one of the intermediate priority levels is followed by a second intermediate priority level with larger priority, e.g. $a+2$ followed by $a+3$ and $\theta^*_{a+2} < \theta^*_{a+3}$. The function does not need to lock the resource of $a+3$, since the lower priority of $a+2$ would need the resource to be unlocked prior to its intended use.

We define a function, referred as $\text{filter}$, which processes the hill interval of the current point and filters out the unnecessary lockings. Algorithm 4 presents the $\text{filter}$ function. This function returns a set of pseudo-resources required to be locked. We use $F$ to represent the output of $\text{filter}$. Since we always need to lock a resource corresponding to the priority level of the next point, $F$ is initialized with $\{r_{\theta^*_a}\}$. We also keep track of the resource with the minimum priority level in $F$ with the variable minF. The loop starts from point $a+2$, because we already considered point $a+1$. Line 5 makes sure that only the points inside the hill interval are considered. Once we find a preemption point which has a priority lower than minF, then we add the resource corresponding to that priority level to $F$ and update minF. The intuition behind Line 6 is as follows. The points that have higher threshold than minF can be considered at later points. This is because when we reach the point that has a threshold equal to the priority of minF, we have to unlock all resources that have priority higher than minF. In the following we present an example for explaining the $\text{filter}$ function.

Figure 8 shows the preemption point schedule, note that the schedule contains both identified scenarios for unnecessary lockings. Each of the subfigures depicts one iteration of the algorithm starting from point 3 to point 6. Point 1 and 2 are considered during initialization before the loop. In 8a, the first point (3) is checked by the algorithm. Since there is no point with lower priority in $F$ the function adds $r_{\theta^*_3}$ and adjusts minF. For the next iteration (8b), the priority level of the point
under consideration is larger than \( \min F \). Thus the point is not added to \( F \). This corresponds to the case shown in 7d. Figure 8c shows the scenario at the next preemption point 6. This priority is already included in \( F \), thus the algorithm proceeds to the next and last point 6 (Figure 8d). Line 5 of Algorithm 4 detects the end of the hill and the function terminates, returning \( F \).

**Algorithm 4: Filter function**

1: function \( \text{filter}(a) \)
2: \( F = \{ r_{a+1} \} \);
3: \( \min F = \theta_{a+1}^* \);
4: \( p = a + 2 \);
5: while \( p \leq m \) and \( \theta_p^* > \theta_a^* \) do
6: if \( r_{\theta_p^*} < \min F \) then
7: \( \min F = r_{\theta_p^*} \);
8: \( F = F \cup \{ r_{\theta_p^*} \} \);
9: end if
10: \( p++ \);
11: end while
12: return \( F \);
13: end function

We define new abstractions for locking and unlocking resources. We introduce the set \( S = \{ s_1, \ldots, s_k \} \), where \( s_i \) represents the state (locked or unlocked) of \( r_{\theta_i^*} \). Therefore, we overload the \( \text{IncreasePriority} \) and \( \text{DecreasePriority} \) functions. The new function for increasing priority has an additional parameter called flag for specifying whether the resource needs to be locked or not, i.e. \( \text{IncreasePriority}(U, L, a, \text{flag}) \). We can skip unnecessary priority levels without locking them by passing \( \text{false} \) as the flag parameter. This function sets the corresponding index in \( L^U \) to one and \( s_a = \text{true} \) if the flag is true. The pop and push operations are performed as before. Similarly, the new function for decreasing the priority only sets the corresponding index in \( L^R \) to one if \( s_a = \text{true} \). Once \( L^R[a, l] \) is set to one it assigns false to \( s_a \). The push and pop operations are performed as before.

Another approach for reducing the number of locking operations is to use one locking for priority levels that are revisited throughout the set of the preemption points, i.e., points \( p \) and \( p' \) such that \( \theta_{\theta_{p'}} = \theta_{\theta_p^*} \). For instance, in Figure 7c we can use one locking for points \( a + 1 \) till \( a + 3 \).

**Lemma 3.** Assume \( \tau \) revisits its priority level in preemption point \( p \) and \( p' \), i.e., \( \theta_{\theta_{p'}} = \theta_{\theta_p^*} \), where point \( p \) is prior to point \( p' \). In order to be possible to use the same locking for both points the following condition should hold: \( \forall e \in [p, p'] \) \( \theta_e^* \geq \theta_{\theta_p^*} \).

**Proof:** The proof is done by contradiction. Assume there exist a point \( e \in [p, p'] \) such that \( \theta_e^* < \theta_{\theta_p^*} \), and assume that we do not release \( r_{\theta_e^*} \) until point \( p' \). Since, \( r_{\theta_p^*} \) is locked at point \( e \) and the priority level of \( r_{\theta_p^*} \) is higher than threshold of point \( e \), the task will not be able to land to the correct priority level at point \( e \).
Algorithm 5 shows our improved `InitLists` function referred as `InitLists*`. The function is similar to the previous version and it behaves the same for the case where $\theta_{a-1} > \theta_a$ (Line 4 to 8). For the second case, where $\theta_{a+1} > \theta_a$, we call the function `hill` prior to locking all resources in order to reach the preemption threshold of $\theta_{a+1}$. As second step we call the `filter` function. This function will exclude all points of the hill, which not need to be locked at $\theta_a$. If a resource is element of $\mathcal{F}$, the `IncreasePriority` function locks the resource. Otherwise it is not locked.

**Theorem 2.** Function `InitLists*` presented in Algorithm 5 is optimal in the sense that there exists no other algorithm that can correctly implement the FPDS* scheme through resource sharing primitives of an OSEK-compliant OS with fewer number of locking than `InitLists*`.

**Proof:** We need to prove that the function `InitLists*` has the following properties. (i) It reuses the same locking for revisited priority levels for all possible revisited thresholds. (ii) It never performs any unnecessary lockings. The function only releases locks if the current priority level is below the obtained locks ceiling (Line 4 to 8) which is inevitable according to Lemma 3. Therefore, property (i) holds. Showing that for each point the function produces at most one locking implies that no unnecessary locking is done. The function only performs locking, if the next point has a higher priority than the current point. The only condition that more than one point ahead is considered is when there is a jump in priority levels and the jumped level(s) are in set $\mathcal{F}$ returned by function `filter`. Therefore, we must show that if point $j$ is considered at point $i$, it will not be considered again in any other points between $i$ and $j$. We prove by contradiction. Assume that point $j$ is considered at point $i$ as well as point $e \in (i, j)$. Let $\mathcal{F} = \text{filter}(\text{hill}(i))$ and $\mathcal{F}' = \text{filter}(\text{hill}(e))$. We can draw the following conclusions from our assumptions: (a) $j \in \mathcal{F}$ because it is considered at point $i$; (b) $j \in \mathcal{F}'$ because it is considered at point $e$; (c) $e \in \text{hill}(i)$ because $j \in \text{hill}(i)$ and $e \in (i, j)$. From (b) we can derive $\theta_e > \theta_j$, and from (c) given that $e$ and $j (e < j)$ are both in the hill of $i$ we have: $\theta_e > \theta_i$. 

---

**Algorithm 5: Optimal InitLists function.**

1: function `InitLists*`(a)  
2: if $a > 0$ then  
3: ReleaseResource($\text{RES\_SCHEDULER}$);  
4: if $\theta_{a-1} > \theta_a$ then  
5: repeat  
6: DecreasePriority($U, L, a$);  
7: until top($L) == \theta_a$  
8: end if  
9: end if  
10: if $a < m$ then  
11: if $\theta_{a+1} > \theta_a$ then  
12: $\mathcal{F} = \text{filter}(a)$  
13: repeat  
14: if top($U) \in \mathcal{F}$ then  
15: IncreasePriority($U, L, a, \text{true}$);  
16: else  
17: IncreasePriority($U, L, a, \text{false}$);  
18: end if  
19: until top($L) == \theta_{a+1}$  
20: end if  
21: GetResource($\text{RES\_SCHEDULER}$);  
22: end if  
23: end function

---

Fig. 8: Example execution of the `filter` function.
A. Experiment design

Experiments were performed on randomly generated preemption point sets, where each set describes one task. We compare tasks with 1 to 100 preemption points. The threshold value for each preemption point was chosen by a uniformly distributed random variable in the range $[0, m]$, where $m$ represents the number of preemption points. We generated 1000 random tasks for each $m$. Since the schedule of the resource access sequence and the preemption points are both not related to other tasks in the system we evaluate each task separately. We then run the different algorithms on the same random sets.

Both algorithms use the functions GetResource and ReleaseResource in order to change the task priorities. Those functions are provided by any OSEK/AUTOSAR implementation. Changing the priority using those functions is constant and does not depend on the pseudo-resource. Thus we can compare the number of resource accesses produced by both algorithms in order to evaluate the overhead of each solution. Figure 9a shows the average number of resource accesses for both InitLists and InitLists* functions. The standard deviation is also plotted for every $10^3$th datapoint in order to keep the graph readable. The InitLists* function results in a linear number of resource accesses whereas the InitLists function grows faster. For a larger number of preemption points it is more likely to get jumps between consecutive preemption points with unused intermediate resources which is exploited by InitLists*. This explains the smaller standard deviation of the InitLists* function as well as the lower number of resource accesses. In Figure 9b the number of resources used by both approaches is shown as well as the standard deviation. Since both algorithms need to lock the same resources in order to reach the threshold priorities of its preemption points the number of used resources is the same. As described before we generated the threshold for each preemption point randomly in the range $[0, m]$ with uniform distribution.

B. Illustrative example

Figure 10 shows an illustrative example, comparing the two functions. The nominal priority of the illustrated task is assumed to be $p_3$. In the example the task has nine subjobs and ten preemption points with varying thresholds. The upper part of the graph shows the current task priority and the lower part depicts the respective calls to the resource sharing primitives. At the first preemption point the two algorithms differ in their output. The first algorithm always locks all resources and results in 48 calls to the resource access primitives. The optimal algorithm avoids unnecessary lockings and results in 32 calls to the resource access primitives.

C. Overhead due to OSEK Priority Ceiling Protocol routines

In order to identify the applicability of our proposed approach, we measured the time a call to the primitives GetResource and ReleaseResource takes in a real AUTOSAR OS implementation. For our experiments we chose ArcticCore [20], as an open source AUTOSAR-compliant OS. Experiments were performed on the Raspberry Pi, with its Broadcom BCM2835 SoC utilizing a 700MHz ARM1176JZF-S processor [21]. We used the ArcticCore port to the Raspberry Pi described in [22].

To measure only the runtime of the OSEK PCP routines we created a task with highest priority in the system. The scheduler resource is then locked and unlocked. Timestamps are taken before the call to the PCP function and after it returned, for both functions respectively. 50,000 samples were taken. The minimum, maximum, average and standard deviation of the observed overhead values are reported in Table III.

VI. DISCUSSIONS

In this section we first compare the two solutions with respect to four aspects. In addition, we show how our approach
can be generalized to support other fixed-priority scheduling policies.

**Memory complexity.** The amount of extra memory required by our approach is due to (i) declaration of pseudo-resources (ii) implementation of the two lists that store the sequence of the primitives (\(L^B\) and \(L^V\)). Both solutions require as many resources as the number of unique preemption threshold levels of all tasks in \(T\). In the worst case we will have \(N−1\) threshold levels. The lists have size \(m_i \times N\). Hence the complexity is \(O(m_i \times N)\).

**Runtime complexity.** The offline part of our approach consists of the following two functions: \texttt{FindPseudoResources} and \texttt{InitLists/InitLists*}. The \texttt{FindPseudoResources} function has complexity of \(O(N^2)\). The \texttt{InitLists} function has run time complexity of \(O(N)\) for each preemption point. The \texttt{InitLists*} function has run time complexity of \(O(m \times N)\) for each preemption point. This is because \texttt{InitLists*} calls the filter function before increasing priority. Recall that \(M = \max_{\varepsilon_i \in \mathcal{T}}(m_i)\). The collective complexity of the offline part of our approach is \(O(MN^2)\) if \texttt{InitLists} is used and it is \(O(N^3M^2)\) if \texttt{InitLists*} is used. The online part of our approach is the \texttt{Yield} function. This function has complexity of \(O(N)\).

**Runtime overhead.** AUTOSAR provides the possibility to associate one exclusive resource with each task, called internal resource. The kernel locks an internal resource for the duration of one job. The priority change is done inside the kernel. Thus a limited version FPTS can be implemented using internal resources. In order to compare the overhead for changing priority incurred by a method implemented in the kernel with our approach, we conducted experiments based on the setup of Section V-C. Our measurements showed that internal resources may be up to four times more efficient than the external resources. Therefore, if we implement the priority change inside the kernel, the overhead will be reduced. However changing the kernel is not trivial and typically the source code of the kernel is not available.

**Number of resource accesses.** Both solutions require at most \((N − i) \times 2 + 2\) calls to resource sharing primitives per preemption point. This worst case is reached if a preemption point with highest threshold is followed by a preemption point with threshold level equal to the task priority and then again by a second preemption point with the highest threshold. Here we assume that the task has \(N − i\) different thresholds for the preemption points. Two additional calls are needed to release and later get the scheduling resource in order to execute the jobs non-preemptively.

**Generalization.** FPDS* is a specific form of FPVS [12]. In FPVS, equivalent to FPDS*, a task \(t_i\) consists of \(m_i\) subjobs, separated by preemption points, and preemption points have individual thresholds. Unlike FPDS*, FPVS executes the subjobs preemptively, instead each subjob is assigned a preemption threshold \(\theta_{i,a}\). The preemption threshold of a preemption point is restricted not to have a higher threshold than its neighboring subjobs, i.e. the condition \(\max(\theta_{i,a}, \theta_{i,a+1}) \geq \theta_{i,a} \geq \pi_i\) must hold for all preemption points. In [12] Bril et al. showed that all other fixed priority scheduling schemes are specializations of FPVS. Thus by showing the applicability of our algorithms for FPVS we show that our proposed algorithm enables AUTOSAR to support all flavors of fixed priority scheduling algorithms.

Figure 11 shows an example FPVS task \(t_i\) with base priority \(\pi_i\). \(t_i\) has three subjobs \(\tau_{i,1}\) to \(\tau_{i,3}\). Each subjob has an assigned preemption threshold \(\theta_{i,j}\) and preemption points have preemption thresholds \(\theta_{i,j}^*\). The top of the figure illustrates the changing priority value during the execution of the task. Table IV specifies the underlying values for the task parameters. We can see that the priority of the task changes as follows during the execution of one job: \(\pi_i \rightarrow \theta_{i,1} \rightarrow \theta_{i,1}^* \rightarrow \theta_{i,2} \rightarrow \theta_{i,2}^* \rightarrow \theta_{i,3} \rightarrow \pi_i\). We convert this sequence of priority values into a set \(P\) with priority values denoted by \(\Phi\), where \(\Phi_{i,1} = \pi_i, \Phi_{i,2} = \theta_{i,1}\) etc.

Since subjobs of FPVS are not executed non-preemptively we need to remove lines 3 and 21 of \texttt{InitLists*}(a) (Algorithm 5) to prevent locking of the scheduler. Generation of the locking sequence can now be done as described for the FPDS* algorithm with the sequence of priorities described in \(P\) being the input of \texttt{InitLists*}. As a consequence, the \texttt{Yield} function is executed both for preemption points and subjob thresholds at runtime.

**VII. Conclusion**

In this paper we exploited the resource access protocol of AUTOSAR in order to change the task priority at runtime according to the FPDS* scheduling algorithm. We provided two algorithms to call pseudo-resources in such a way that the task priorities change as defined by the schedule. We proved the correctness of the first algorithm. Furthermore, we proved that our second algorithm is optimal in the number of calls to the resources access primitives and we showed how to generalize our solution to support all fixed-priority scheduling schemes.
Fig. 10: Illustrative example, comparing the two proposed algorithms to generate the resource accesses to the pseudo-resources.

GetResource() and ReleaseResource() functions respectively. The base priority of the task is $\pi_1$.

**REFERENCES**


