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. FPDS$^\bullet$, 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 FPDS$^\bullet$ schedules. In our approach we exploit pseudo resources in order to implement FPDS$^\bullet$. 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 preemption thresholds (FPTS) [2], [3], [4], [5], [6], [7]. On the other hand, some task models only allow preemption at specific points in time. This approach is known as fixed-priority scheduling with deferred preemptions (FPDS) [8], [9], [10].

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).

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 FPGS, and later in [12] the authors refined the model to fixed-priority scheduling with varying preemption thresholds (FPVS). In the same work FPDS$^\bullet$ 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$^\bullet$*

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$^\bullet$ 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 τ_1 and τ_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_i^P, WR_i^N and WR_i^D 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 τ_2 is divided into two subjobs, then the task set becomes indeed schedulable with FPDS, since the second subjob of τ_2 cannot be preempted by τ_1. Let us consider scheduling the task set with FPTS. We can either execute the jobs of τ_2 with a threshold level equal to the priority of τ_2 or equal to the priority of τ_1. The resulting schedule is then equivalent to FPPS 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

	$T_i = D_i$	C_i	π_i	WR_i^P	WR_i^N	WR_i^D
τ_1	5	2	2	2	**6**	4
τ_2	7	2+2	1	**8**	6	7

TABLE I: Characteristics of the task set used in Example 1. This task set is only schedulable using FPDS.

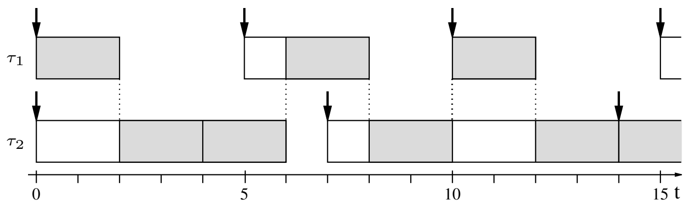


Fig. 1: The execution trace of Example 1 in which τ_2 experiences its worst-case response time. This task set is schedulable using FPDS.

	T_i	D_i	C_i	π_i	WR_i^P	WR_i^N	Θ_i	WR_i^T
τ_1	70	5	5	4	10	**45**	4	5
τ_2	70	50	15	3	20	**55**	3	40
τ_3	80	80	20	2	40	75	3	80
τ_4	200	100	35	1	**115**	75	2	95

TABLE II: Characteristics of the task set used in Example 2. This task set is only schedulable using FPTS.

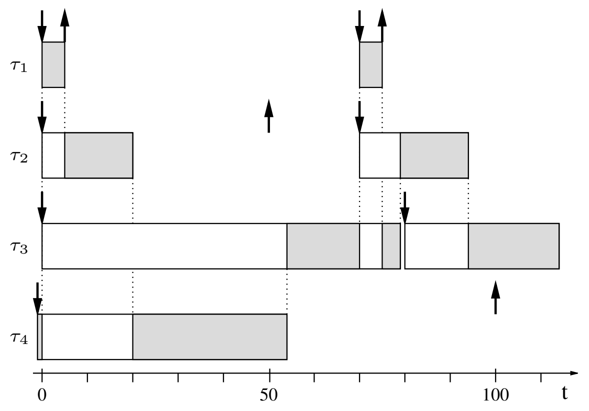


Fig. 2: The execution trace of Example 2 in which τ_3 experiences its worst-case response time. This task set is schedulable using FPTS.

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 τ_i is represented by Θ_i. Figure 2 depicts a possible execution trace, where τ_4 is released just before all other tasks. Since the preemption threshold of τ_4 is equal to the priority of τ_3, τ_3 cannot preempt τ_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 τ_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 τ_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$^\bullet$ and FPVS have a higher success ratio in comparison to all other fixed-priority algorithms considered in the experiment. In order to make

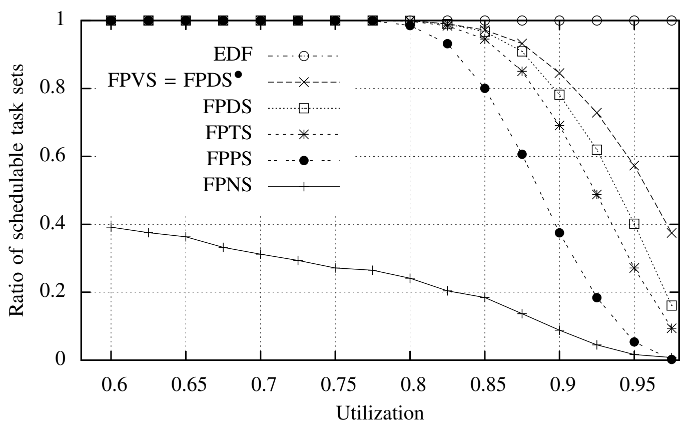


Fig. 3: Ratio of schedulable task sets versus the task-set's utilization (redrawn from [12]).

use of this high schedulability ratio in OSEK/AUTOSAR-compliant operating systems, in this paper, we present an approach to implement the FPDS$^\bullet$ algorithm.

B. Support for FPDS$^\bullet$ 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$^\bullet$, 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$^\bullet$ 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. Backgrounds

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 $\mathcal{T} = \{\tau_1, \ldots, \tau_N\}$, consisting of N independent tasks τ_i. Tasks are scheduled on a single processor. A task τ_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 π_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$^\bullet$, 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. Those 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 $\theta^\bullet_{i,a}$, where $\pi_i \leq \theta^\bullet_{i,a} \leq \pi_N$, and $\theta^\bullet_{i,0} = \theta^\bullet_{i,m_i} = \pi_i$.

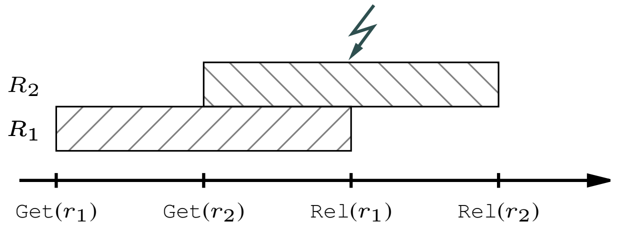
B. OSEK/AUTOSAR

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.

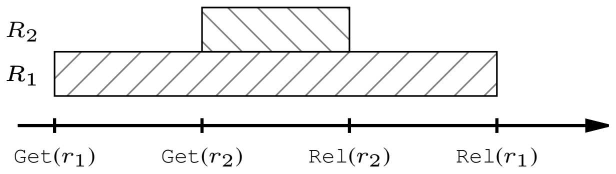
1) Fixed Priority Fully-Preemptive Scheduling;
2) Fixed Priority Non-Preemptive Scheduling;
3) Fixed Priority Mixed Preemptive Scheduling.

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.



(a) Improper nesting of resource access.



(b) Proper nesting of resource access.

Fig. 4: Example of proper and improper nesting of resources. `Get()` and `Rel()` functions represent `GetResource()` and `ReleaseResource()` primitives respectively.

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 exited in the reverse order as they entered the critical section. 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 τ_i to be able to elevate its priority at preemption points and return back to its nominal priority π_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 τ_i locks a resource r_j, first the resource stores

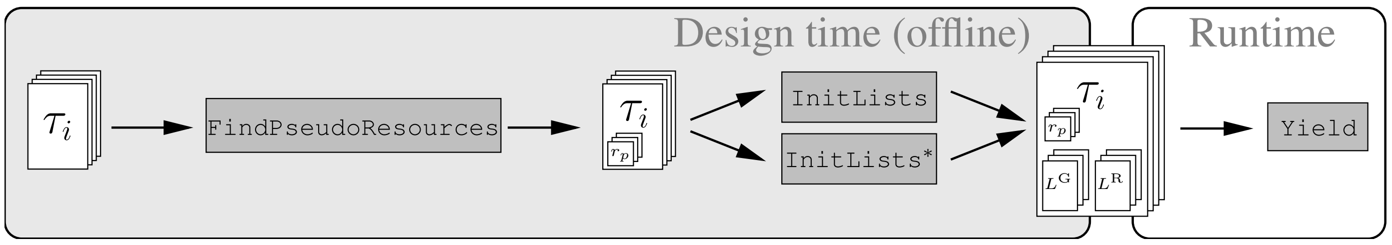


Fig. 5: Information flow of our proposed approach.

the active priority of the task (r_j→`old_task_prio` = τ_i→`activePriority`). Thereafter, the active priority of the task is set to the ceiling of the resource (τ_i→`activePriority` = r_j→`ceiling_priority`). After releasing the resource, the active priority of the task is set back to the old priority stored in the resource (τ_i→`activePriority` = r_j→`old_task_prio`). The proof follows from the implementation of the PCP protocol. Suppose we have improper nesting, i.e, task τ_i first locks resource r_j then r_k, while it first releases r_j and then r_k. After locking we will have: r_j→`old_task_prio` = π_i, r_k→`old_task_prio` = j and τ_i→`activePriority` = k. After unlockings we will have: τ_i→`activePriority` = j. While the task has to return back to its nominal priority π_i after releasing all of its resources. ■

III. Solution

Our approach consists of two parts: (i) the design steps which are performed offline; (ii) a runtime function. The flow of our approach is illustrated in Figure 5. Since we use pseudo-resources for changing the priorities, the first step is to derive the required pseudo-resources. We use the `FindPseudoResources` function presented in Algorithm 1 for deriving the required pseudo-resources. Afterwards we create two lists that store the sequence of locking and unlocking the resources. We provide two different solutions for initializing the two lists. Solution 2 improves upon the first solution by locking only necessary resources and thus resulting in a minimum number of resource lockings. The two functions are explained in detail in Section IV. Finally, the online function (`Yield`) uses the two lists and calls the resource access primitives at proper points in time for implementing the FPDS$^\bullet$ algorithm. In the following we first present the `FindPseudoResources` function. Afterwards, assuming that the lists are initiated correctly, we present the runtime mechanism, i.e. the `Yield` function.

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$^\bullet$ schedule, we define a set $\mathcal{R}$ containing one pseudo resource for each threshold priority used by the tasks in $\mathcal{T}$, where r_p denotes the pseudo-resource with ceiling p. Each task $\tau_i \in \mathcal{T}$ uses a subset of $\mathcal{R}$. Therefore, for elevating the threshold at preemption point a we need to lock $r_{\theta^\bullet_{i,a}}$. 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:

- Every task that has at least one preemption point with a threshold equal to the ceiling of r_p.
- τ_p, because we want the ceiling priority of r_p to be assigned to π_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 $\mathcal{T}$ (Line 3), and it further iterates over all preemption points of that task (Line 4). It may be the case that τ_i is the only task with a preemption point with a preemption threshold equal to π_i. In this case we do not need to add r_i to $\mathcal{R}$, 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 $\mathcal{R}$. It is assumed that each resource can only be added once. If the resource is not already in $\mathcal{R}$, then the algorithm generates a new pseudo-resource. This resource is added to $\mathcal{R}$ 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 $M = \max_{\forall \tau_i \in \mathcal{T}}(m_i)$. The algorithm has a complexity of $\mathcal{O}(N \times M)$ and it is executed offline.

Algorithm 1: FindPseudoResources

```
1:  function FindPseudoResources($\mathcal{T}$)
2:  $\mathcal{R} \leftarrow \emptyset$
3:  for $\forall \tau_i \in \mathcal{T}$ do
4:      for $\theta^\bullet_{i,a} | 1 < a < m_i$ do
5:          if $\theta^\bullet_{i,a} \neq \pi_i$ then
6:              if $\nexists\, r_{\theta^\bullet_{i,a}} \in \mathcal{R}$ then
7:                  New resource $r_{\theta^\bullet_{i,a}}$ and $\mathcal{R} \leftarrow \mathcal{R} \bigcup \{r_{\theta^\bullet_{i,a}}\}$
8:                  Add resource to $\tau_{\theta^\bullet_{i,a}}$
9:              end if
10:             Add resource to $\tau_i$
11:         end if
12:     end for
13: end for
14: end function
```

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$^\bullet$ 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 τ_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 π_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_i^{R} for all calls to ReleaseResource; and L_i^{G} for all calls to GetResource. If $L_i^{\mathrm{G}}[a,p]$ is marked true, then τ_i has to obtain r_p at preemption point a.

Task τ_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 τ_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 $\mathcal{O}(N-i)$ for each task τ_i.

Algorithm 2: Yield function

```
1: function Yield($a, i$)
2: $index = 0$
3: while $index \leq m_i$ do
4:     if $L_i^{\mathrm{R}}[a, index]$ =true then
5:         ReleaseResource($L_i^{\mathrm{R}}[a, index]$)
6:     end if
7:     $index++$
8: end while
9: $index = 0$
10: while $index \leq m_i$ do
11:     if $L_i^{\mathrm{G}}[a, index]$ =true then
12:         GetResource($L_i^{\mathrm{G}}[a, index]$)
13:     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 τ_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. In order to boost the task priority to $\theta_p^\bullet$, resource $r_{\theta_p^\bullet}$ has to be locked. Therefore, for all resources with priority lower than $r_{\theta_p^\bullet}$, we (i) remove the resource from the unlock stack; (ii) lock the resource; (iii) insert it into the locked stack. For reducing the threshold to a lower value, we move resources in the opposite order from the locked stack to the unlocked stack while unlocking them. For our convenience, we define two functions which encapsulate the resource handling operations. Function IncreasePriority(U, L, a) first pops a resource (e.g., r_u) from U. Then, it assigns the corresponding index in the get list to one, i.e. $L^{\mathrm{G}}[a,u] = 1$. Finally, it pushes the resource in L. The DecreasePriority(U, L, a) function, however, first pops a resource (e.g., r_l) from L. Then, it sets the corresponding index in the release list to one, i.e $L^{\mathrm{R}}[a,l] = 1$. Finally, it pushes the resource into U.

Lemma 2. *Any algorithm which allows access to the pseudo-resources only through the* IncreasePriority *and* DecreasePriority *functions results in proper nesting.*

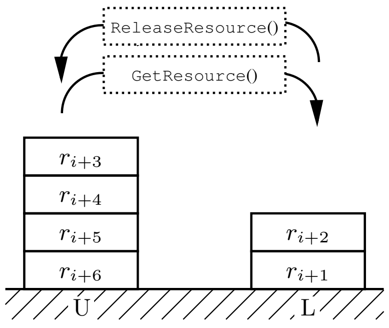


Fig. 6: Stack usage to manage proper nesting of resource accesses by τ_i.

Proof: The proof follows from the fact that locking is done in the order of increasing priority while the unlocking is performed in decreasing priority order. Therefore, $\forall i,j | r_i > r_j$ r_j is locked before r_i, whereas r_j is unlocked after r_i. ∎

A. InitLists *function*

We present our function, referred as InitLists(a) function, which has to be called for each preemption point for initializing L^{G} and L^{R}. Apart from initialization of U with all resource used by the task, no prior steps are needed. Algorithm 3 depicts the InitLists(a) function. The function receives the index of the current preemption point a as an input. As mentioned before, we consider the start and the end of a task as preemption points. Therefore, the function is called at those points as well. The task start and task end indices are equal to 0 and m respectively. The function first checks, if it is the first preemption point or not (Line 2). If not, the task was running non-preemptively and thus the scheduler needs to be released. Before the function returns, the scheduler is locked again, making the next subjob non-preemtable (Line 16). A special case is the end of the task where we have no next subjob and thus do not need to lock the scheduler. In order to set the desired preemption threshold we identified two cases where a change of locked resources is needed.

Case 1: if $\theta_a^\bullet$ is smaller than $\theta_{a-1}^\bullet$, the preemption threshold 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 $\theta_a^\bullet$ or it is empty (Line 7). Note that when the head of the stack is empty we have reached the base priority of π_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 2: when the priority of the next preemption point $\theta_{a+1}^\bullet$ is larger than the current point $\theta_a^\bullet$ (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 $\theta_{a+1}^\bullet$. 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$^\bullet$.*

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, $\forall p \in (0,m)$, $r_{\theta_p^\bullet}$ has to be

Algorithm 3: InitLists function

```
 1: function InitLists($a$)
 2: if $a > 0$ then
 3:   ReleaseResource(RES_SCHEDULER);
 4:   if $\theta^\bullet_{a-1} > \theta^\bullet_a$ then
 5:     repeat
 6:       DecreasePriority(U, L, $a$);
 7:     until top(L) == $\theta^\bullet_a$ || top(L)= $\emptyset$
 8:   end if
 9: end if
10: if $a < m$ then
11:   if $\theta^\bullet_{a+1} > \theta^\bullet_a$ then
12:     repeat
13:       IncreasePriority(U, L, $a$);
14:     until top(L) == $\theta^\bullet_{a+1}$
15:   end if
16:   GetResource(RES_SCHEDULER);
17: end if
18: end function
```

kept locked at preemption point p; (ii) the algorithm respects proper nesting. Except the scheduler resource, all other resources are accessed through the `IncreasePriority` and `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 $r_{\theta^\bullet_p}$ before preemption point p, and it releases all resources with priority higher than the current threshold ($\theta^\bullet_p$) at point p. Line 11 to 15 guarantee that $r_{\theta^\bullet_p}$ 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 unlockings), we investigate the conditions in the `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 $r_p \in \mathcal{R}_i$ with a resource ceiling between $\theta^\bullet_a$ of the current point a and $\theta^\bullet_{a+1}$ of the next point $a+1$ are locked by `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.

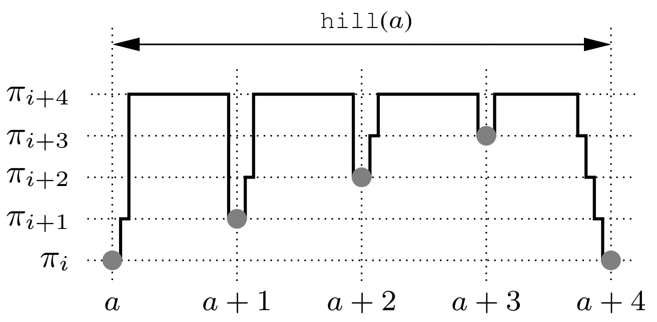
We define an interval, referred as *hill*, for the current point a starting from a and ending in b, where $\theta^\bullet_b \leq \theta^\bullet_a$ and all points between a and b have a threshold higher than $\theta^\bullet_a$. We also define function `hill(`a`)` which returns the hill interval of its 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 $\theta^\bullet_a$ and $\theta^\bullet_{a+1}$ have to be unlocked at point b ($\theta^\bullet_b \leq \theta^\bullet_a$). 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 $\theta^\bullet_{a+1} > \theta^\bullet_a$.

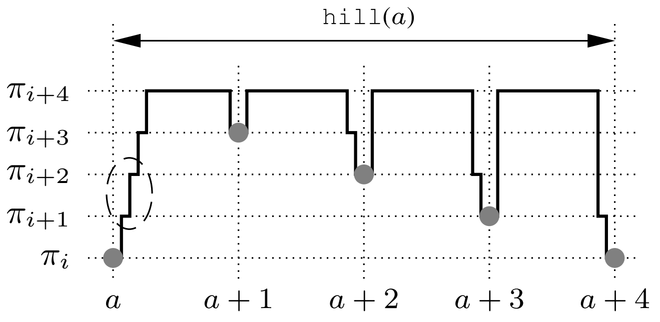
a) If the next preemption point $a+1$ has a threshold level one priority higher than the current preemption point a, i.e. $\theta^\bullet_{a+1} = \theta^\bullet_a + 1$. The function locks the resource in order to obtain the correct threshold level at $a+1$.
b) The threshold of the next preemption point $a+1$ is two or more priority levels higher than the current priority threshold $\theta^\bullet_a$, i.e. $\theta^\bullet_{a+1} \geq \theta^\bullet_a + 2$. 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 $r_{\theta^\bullet_{a+1}}$.
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 $\theta^\bullet_a$. 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 $\theta^\bullet_{a+2} < \theta^\bullet_{a+3}$. 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 `filter`, which processes the hill interval of the current point and filters out the unnecessary lockings. Algorithm 4 presents the `filter` function. This function returns a set of pseudo-resources required to be locked. We use $\mathcal{F}$ to represent the output of `filter`. Since we always need to lock a resource corresponding to the priority level of the next point, $\mathcal{F}$ is initialized with $\{r_{\theta^\bullet_{a+1}}\}$. We also keep track of the resource with the minimum priority level in $\mathcal{F}$ 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 $\mathcal{F}$ 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 `filter` function.

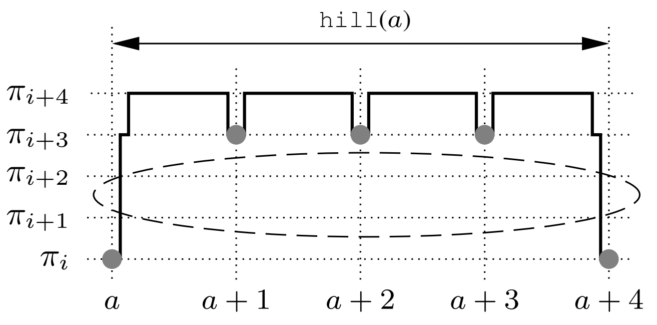
Figure 8 shows the preemption point schedule, note that the schedule contains both identified scenarios for unnecessary lockings. Each of the subfigures depicts one iteration of the algorithm starting from point 3 to point 6. Point 1 and 2 are considered during initialization before the loop. In 8a, the first point (3) is checked by the algorithm. Since there is no point with lower priority in $\mathcal{F}$ the function adds $r_{\tau_{i+2}}$ and adjusts minF. For the next iteration (8b), the priority level of the point



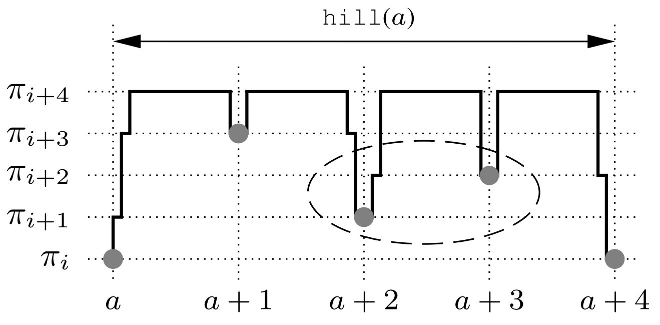
(a) Simple case where the next preemption point has one priority higher than the current one.



(b) The next preemption point is more than one priority higher than the current one but the resource representing the intermediate priority must first be locked to keep proper nesting.



(c) The priority $\theta^\bullet_{a+1}$ of the next preemption point $a+1$ is higher than $\theta^\bullet_a + 1$ and none of the preemption points in the hill interval hill(a) (excluding $a+4$) has a priority between $\theta^\bullet_a$ and $\theta^\bullet_{a+1}$.



(d) There are multiple preemption points with lower priority than the next preemption point and the later one has higher priority than the earlier one. To reduce unnecessary lockings, $r_{\theta^\bullet_{a+3}}$ is not locked at a, since we would need to unlock it at preemption point $a+2$, before using it at $a+3$.

Fig. 7: Different possible scenarios at a preemption point a.

under consideration is larger than minF. Thus the point is not added to $\mathcal{F}$. This corresponds to the case shown in 7d. Figure 8c shows the scenario at the next preemption point 5. This priority is already included in $\mathcal{F}$, 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 $\mathcal{F}$.

Algorithm 4: Filter function

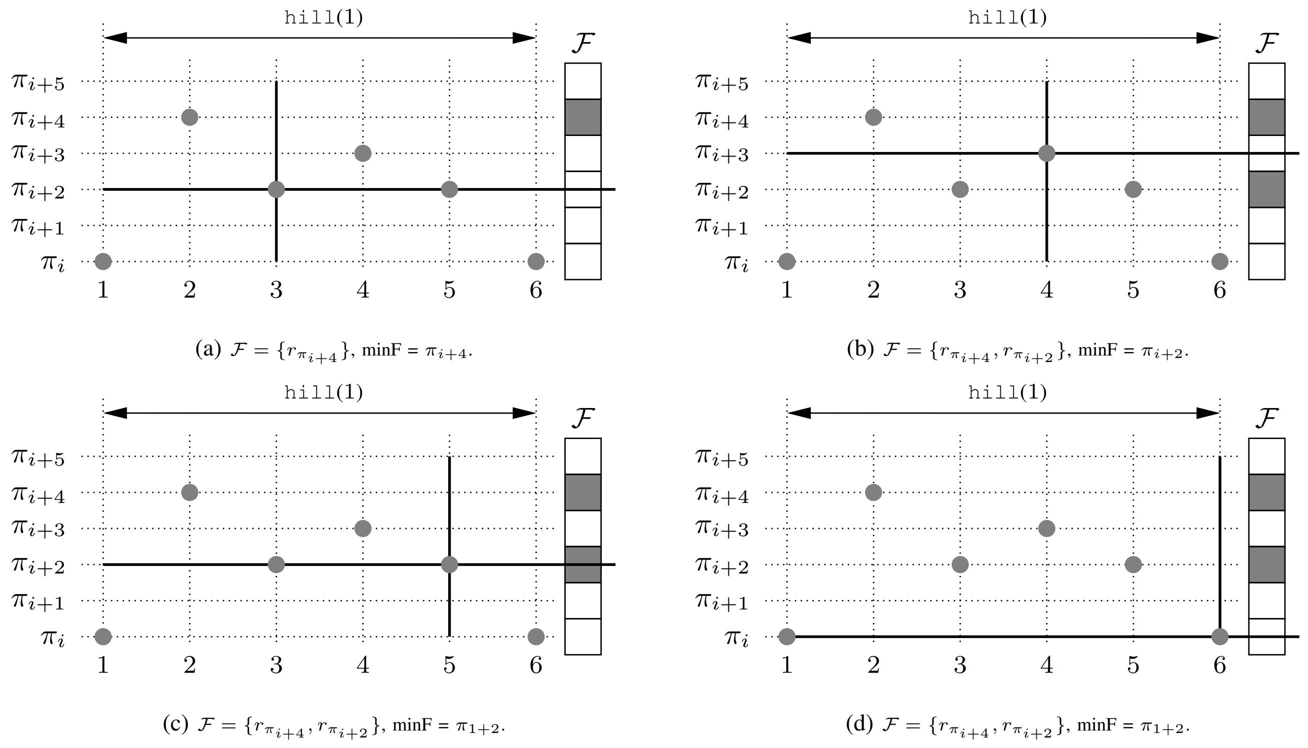
```
1: function = filter(a)
2: $\mathcal{F} = \{r_{\theta^\bullet_{a+1}}\}$;
3: minF = $\theta^\bullet_{a+1}$;
4: $p = a + 2$;
5: while $p \leq m$ and $\theta^\bullet_p > \theta^\bullet_a$ do
6:     if $r_{\theta^\bullet_p} <$ minF then
7:         minF= $r_{\theta^\bullet_p}$;
8:         $\mathcal{F} = \mathcal{F} \bigcup \{r_{\theta^\bullet_p}\}$;
9:     end if
10:    $p$++;
11: end while
12: return $\mathcal{F}$;
13: end function
```

We define new abstractions for locking and unlocking resources. We introduce the set $\mathcal{S} = \{s_1, \cdots, s_k\}$, where s_i represents the state (*locked* or *unlocked*) of $r_{\theta^\bullet_i}$. Therefore, we overload the IncreasePriority and 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 IncreasePriority(U, L, a, flag). We can skip unnecessary priority levels without locking them by passing *false* as the flag parameter. This function sets the corresponding index in L^{G} to one and $s_u = true$ 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 $s_l = true$. Once $L^{\mathrm{R}}[a, l]$ is set to one it assigns false to s_l. 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 $\theta^\bullet_p = \theta^\bullet_{p'}$. For instance, in Figure 7c we can use one locking for points $a+1$ till $a+3$.

Lemma 3. *Assume τ revisits its priority level in preemption point p and p', i.e., $\theta^\bullet_p = \theta^\bullet_{p'}$, 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: $\forall e \in [p, p'] \quad \theta^\bullet_e \geq \theta^\bullet_p$.*

Proof: The proof is done by contradiction. Assume there exist a point $e \in [p, p']$ such that $\theta^\bullet_e < \theta^\bullet_p$, and assume that we do not release $r_{\theta^\bullet_p}$ until point p'. Since, $r_{\theta^\bullet_p}$ is locked at point e and the priority level of $r_{\theta^\bullet_p}$ is higher than threshold of point e, the task will not be able to land to the correct priority level at point e. ■



(a) $\mathcal{F} = \{r_{\pi_{i+4}}\}$, minF = π_{i+4}.

(b) $\mathcal{F} = \{r_{\pi_{i+4}}, r_{\pi_{i+2}}\}$, minF = π_{i+2}.

(c) $\mathcal{F} = \{r_{\pi_{i+4}}, r_{\pi_{i+2}}\}$, minF = π_{1+2}.

(d) $\mathcal{F} = \{r_{\pi_{i+4}}, r_{\pi_{i+2}}\}$, minF = π_{1+2}.

Fig. 8: Example execution of the `filter` function.

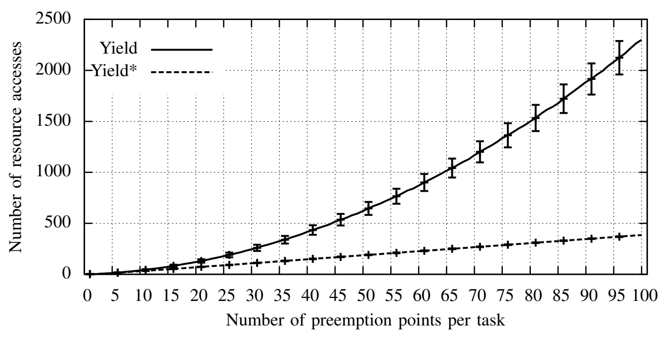
Algorithm 5 shows our improved `InitLists` function referred as `InitLists`*. The function is similar to the previous version and it behaves the same for the case where $\theta^\bullet_{a-1} > \theta^\bullet_a$ (Line 4 to 8). For the second case, where $\theta^\bullet_{a+1} > \theta^\bullet_a$, we call the function `hill` prior to locking all resources in order to reach the preemption threshold of $\theta^\bullet_{a+1}$. As second step we call the `filter` function. This function will exclude all points of the hill, which not need to be locked at $\theta^\bullet_a$. If a resource is element of $\mathcal{F}$, the `IncreasePriority` function locks the resource. Otherwise it is not locked.

Algorithm 5: Optimal InitLists function.

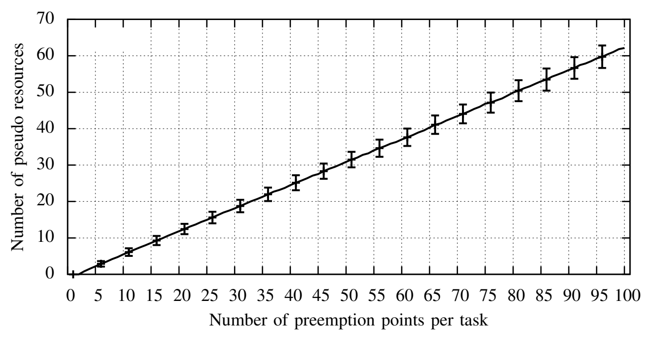
```
 1: function InitLists*($a$)
 2: if $a > 0$ then
 3:   ReleaseResource(RES_SCHEDULER);
 4:   if $\theta^\bullet_{a-1} > \theta^\bullet_a$ then
 5:     repeat
 6:       DecreasePriority(U, L, $a$);
 7:     until top(L) == $\theta^\bullet_a$
 8:   end if
 9: end if
10: if $a < m$ then
11:   if $\theta^\bullet_{a+1}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

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$^\bullet$ 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 $\mathcal{F}$ = `filter(hill(i))` and $\mathcal{F}'$ = `filter(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) $e \in$ `hill`(i) because $j \in$ `hill`(i) and $e \in (i, j)$. From (b) we can derive $\theta^\bullet_j > \theta^\bullet_e$, and from (c) given that e and j ($e < j$) are both in the hill of i we have: $\theta^\bullet_e > \theta^\bullet_i$.



(a) Comparison of the introduced overhead of `InitLists` and `InitLists`* functions.



(b) Number of pseudo-resources needed by both `InitLists` and `InitLists`* functions.

Fig. 9: Evaluation results for the number of resource access calls and the number of pseudo resources.

Therefore, we have $\theta^\bullet_j > \theta^\bullet_e > \theta^\bullet_i$. However, since e is before j, in function `filter`(i) we will have minF= $r_{\theta^\bullet_e}$ before approaching point j. Therefore function `filter`(i) will not add $r_{\theta^\bullet_j}$ to $\mathcal{F}$, and point j will not be considered at point i which contradicts our assumption. ■

V. Evaluations

In this section we investigate the relation between the number of preemption points in a task and (i) the number of resource access primitives generated by the two algorithms (ii) the number of required pseudo-resources. In addition, we report the measurement results regarding the cost of resource access primitives. As part of the evaluation we developed a simulation tool to compare the two algorithms[1].

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^{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.

[1]The source code of our simulation tool is availabe at: https://bitbucket.org/matthiasbecker/fpds_dot_simulation

B. Illustrative example

Figure 10 shows an illustrative example, comparing the two functions. The nominal priority of the illustrated task is assumed to be π_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

	`GetResource`	`ReleaseResource`
Minimum	5.0000 µs	9.0000 µs
Maximum	6.0000 µs	12.0000 µs
Average	5.0063 µs	9.2253 µs
Standard Deviation	0.0796 µs	0.4181 µs

TABLE III: Observed overhead values.

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^{G} and L^{R}). Both solutions require as many resources as the number of unique preemption threshold levels of all tasks in $\mathcal{T}$. In the worst case we will have $N-1$ threshold levels. The lists have size $m_i \times N$. Hence the complexity is $\mathcal{O}(m_i \times N)$.

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

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$^\bullet$ is a specific form of FPVS [12]. In FPVS, equivalent to FPDS$^\bullet$, a task τ_i consists of m_i subjobs, separated by preemption points, and preemption points have individual thresholds. Unlike FPDS$^\bullet$, 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 subjobjs, i.e. the condition $\max(\theta_{i,a}, \theta_{i,a+1}) \geq \theta^\bullet_{i,a} \geq \pi_i$ 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.

Parameter	π_i	$\theta_{i,1}$	$\theta^\bullet_{i,1}$	$\theta_{i,2}$	$\theta^\bullet_{i,2}$	$\theta_{i,3}$
Priority value	π_i	π_{i+3}	π_{i+1}	π_{i+2}	π_{i+2}	π_{i+4}

TABLE IV: Configuration of the example task shown in Figure 11.

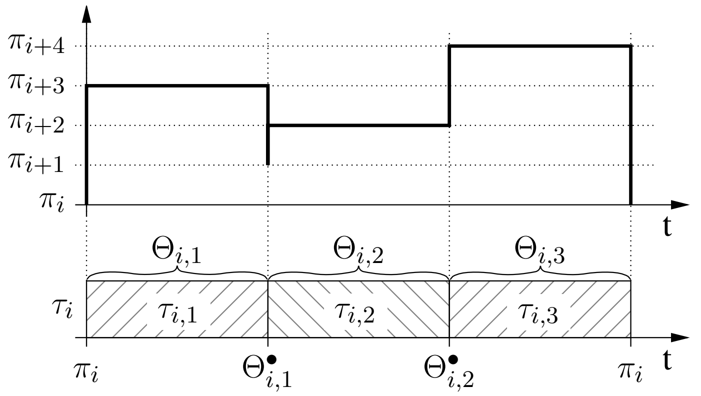


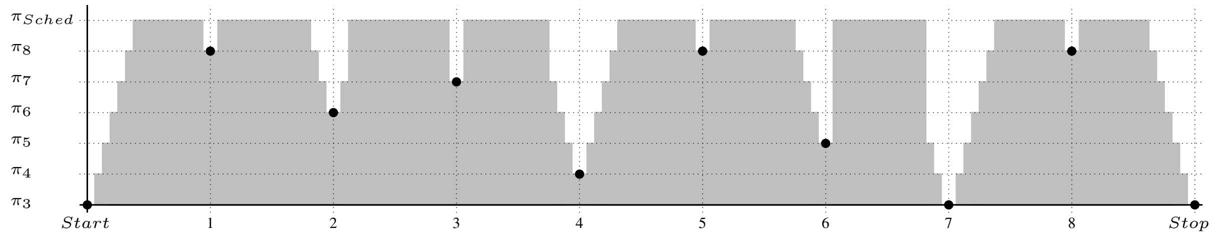
Fig. 11: Examples of the priorities of a FPVS task.

Figure 11 shows an example FPVS task τ_i with base priority π_i. τ_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 $\theta^\bullet_{i,j}$. 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: $\pi_i \rightarrow \theta_{i,1} \rightarrow \theta^\bullet_{i,1} \rightarrow \theta_{i,2} \rightarrow \theta^\bullet_{i,2} \rightarrow \theta_{i,3} \rightarrow \pi_i$. We convert this sequence of priority values into a set $\mathcal{P}$ with priority values denoted by Φ, where $\Phi_{i,1} = \pi_i$, $\Phi_{i,2} = \theta_1$ etc.

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

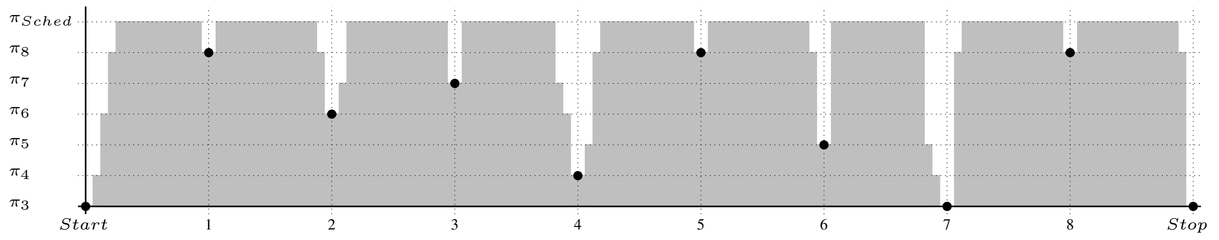
VII. Conclusion

In this paper we exploited the resource access protocol of AUTOSAR in order to change the task priority at runtime according to the FPDS$^\bullet$ 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.



1	Get(4)	Rel(SCHED)	Rel(SCHED)	Rel(SCHED)	Rel(SCHED)	Rel(SCHED)	Rel(SCHED)	Rel(SCHED)	Rel(SCHED)	Rel(SCHED)
2	Get(5)	Get(SCHED)	Rel(8)	Get(SCHED)	Rel(7)	Get(SCHED)	Rel(8)	Rel(5)	Get(SCHED)	Rel(8)
3	Get(6)		Rel(7)		Rel(6)		Rel(7)	Rel(4)		Rel(7)
4	Get(7)		Get(7)		Rel(5)		Rel(6)	Get(4)		Rel(6)
5	Get(8)		Get(SCHED)		Get(5)		Get(SCHED)	Get(5)		Rel(5)
6	Get(SCHED)				Get(6)			Get(6)		Rel(4)
7					Get(7)			Get(7)		
8					Get(8)			Get(8)		
9					Get(SCHED)			Get(SCHED)		

(a) Priority curve of an example task with different preemption points if function `InitLists` is used.



1	Get(4)	Rel(SCHED)	Rel(SCHED)	Rel(SCHED)	Rel(SCHED)	Rel(SCHED)	Rel(SCHED)	Rel(SCHED)	Rel(SCHED)	Rel(SCHED)
2	Get(6)	Get(SCHED)	Rel(8)	Get(SCHED)	Rel(7)	Get(SCHED)	Rel(8)	Rel(5)	Get(SCHED)	Rel(8)
3	Get(8)		Get(7)		Rel(6)		Get(SCHED)	Rel(4)		
4	Get(SCHED)		Get(SCHED)		Get(5)			Get(8)		
5					Get(8)			Get(SCHED)		
6					Get(SCHED)					

(b) Priority curve of an example task with different preemption points if function `InitLists`* is used.

Fig. 10: Illustrative example, comparing the two proposed algorithms to generate the resource accesses to the pseudo-resources. `Get()` and `Rel()` functions represent `GetResource()` and `ReleaseResource()` primitives respectively. The base priority of the task is π_3.

References

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, January 1973.

[2] M. Saksena and Y. Wang, "Scalable real-time system design using preemption thresholds," in *Proc. RTSS*, 2000, pp. 25–34.

[3] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *Proc. RTCSA*, 1999, pp. 328–335.

[4] J. Regehr, "Scheduling tasks with mixed preemption relations for robustness to timing faults," in *Proc. RTSS*, 2002, pp. 315–326.

[5] G. Yao and G. Buttazzo, "Reducing stack with intra-task threshold priorities in real-time systems," in *Proc. EMSOFT*, 2010, pp. 109–118.

[6] U. Keskin, R. Bril, and J. Lukkien, "Exact response-time analysis for fixed-priority preemption-threshold scheduling," in *Proc. ETFA*, 2010, pp. 1–4.

[7] R. Davis, N. Merriam, and N. Tracey, "How embedded applications using an RTOS can stay within on-chip memory limits," in *Proc. ECRTS*, 2000, pp. 71–77.

[8] R. J. Bril, J. J. Lukkien, and W. F. J. Verhaegh, "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited," in *Proc. ECRTS*, 2007, pp. 269–279.

[9] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo, "Optimal selection of preemption points to minimize preemption overhead," in *Proc. ECRTS*, 2011, pp. 217–227.

[10] M. Bertogna, G. Buttazzo, and G. Yao, "Improving feasibility of fixed priority tasks using non-preemptive regions," in *Proc. RTSS*, 2011, pp. 251–260.

[11] R. J. Bril, M. M. H. P. van den Heuvel, U. Keskin, and J. J. Lukkien, "Generalized fixed-priority scheduling with limited preemptions," in *Proc. ECRTS*, 2012, pp. 209–220.

[12] R. J. Bril, M. M. H. P. van den Heuvel, and J. J. Lukkien, "Improved feasibility of fixed-priority scheduling with deferred preemption using preemption thresholds for preemption points," in *Proc. RTNS*, 2013, pp. 255–264.

[13] *AUTOSAR - Specification of Operating System*, AUTOSAR Std. V5.0.0, Rev. R4.0 Rev3, 2011. [Online]. Available: www.autosar.org

[14] "OSEK/VDX operating system," OSEK group, Tech. Rep., 2005.

[15] H. Zeng, M. di Natale, and Q. Zhu, "Minimizing stack and communication memory usage in real-time embedded applications," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 5s, July 2014, article 149.

[16] R. I. Davis and M. Bertogna, "Optimal fixed priority scheduling with deferred pre-emption," in *Proc. RTSS*, 2012, pp. 39–50.

[17] P. Gai, G. Lipari, and M. Di Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip," in *Proc. RTSS*, Dec 2001, pp. 73–83.

[18] T. P. Baker, "Stack-based scheduling for realtime processes," *Real-Time Syst.*, vol. 3, no. 1, pp. 67–99, Apr. 1991.

[19] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, Sep 1990.

[20] "ArcticCore product page." [Online]. Available: http://www.arccore.com/products/arctic-core/

[21] Broadcom Corporation, *BCM 2835 ARM Peripherals*, Broadcom Europe Ltd., 406 Science Park Milton Road Cambridge CB4 0WW, 2012.

[22] S. Zhang, A. Kobetski, E. Johansson, J. Axelsson, and H. Wang, "Porting an AUTOSAR-compliant operating system to a high performance embedded platform," in *Proc. EWiLi*, 2013.