Reducing the Number of Preemptions in Fixed Priority Scheduling

Radu Dobrin* and Gerhard Fohler
Department of Computer Engineering
Mälardalen University, Sweden
{radu.dobrin, gerhard.fohler}@mdh.se

Abstract

Fixed priority scheduling (FPS) has been widely studied and used in a number of applications, mostly due to its flexibility, simple run-time mechanism and small overhead. However, preemption related overhead in FPS may cause undesired high processor utilization, high energy consumption, or, in some cases, even infeasibility.

In this paper, we propose a method to reduce the number of preemptions in legacy FPS systems consisting of tasks with priorities, periods and offsets. Unlike other approaches, our algorithm does not require modification of the basic FPS mechanism. Our method analyzes off-line a set of periodic tasks scheduled by FPS, detects the maximum number of preemptions that can occur at run-time, and reassigns task attributes such that the tasks are schedulable by the same scheduling mechanism, while achieving a significantly lower number of preemptions.

In some cases, there is a cost to pay for a lower number of preemptions in terms of increased amount of tasks and/or reduced task execution flexibility. Our method provides for the ability to choose a user-defined number of preemptions with respect to the cost to pay.

1. Introduction and problem description

Preemptive fixed priority scheduling (FPS) has been widely studied since the work of Liu and Layland [13]. It has gained large acceptance in a number of applications, mostly due to simple run-time scheduling and good flexibility for tasks with incompletely known attributes. However, the impact of preemption related overhead in FPS in the context of real-time systems, is well recognized [2, 14]. In multimedia applications, for example, tasks may introduce a high context switch cost [5]. In fact, preemption related overhead in FPS may cause undesired high processor utilization, high energy consumption, or, in some cases, even infeasibility. In [3], the author showed that the rate monotonic algorithm (RM) introduces a higher number of preemptions than earliest deadline first algorithm (EDF). At the same time, reducing the number of preemptions can also be beneficial from an energy point of view in systems with demands on low power consumption. When a task is preempted there is a great probability that its contents in the cache will be lost. When the execution of the task is again resumed it will cause a lot of energy consuming accesses to off-chip memory. An access to off-chip memory is typically 10-100 times more expensive than an on-chip cache access in terms of energy consumption. Reducing the number of preemptions will reduce these additional expensive memory accesses due to reduced cache pollution.

The direct preemption cost, i.e., costs to perform context switches [7], to handle interrupts [7, 6, 2], or to manipulate task queues [2, 7], has been analyzed. Cache-related preemption, i.e., indirect cost, [10, 15], has been analyzed to incorporate it into schedulability analysis, as well as approaches to bound the cache-related preemption delay have been presented [11]. Approaches to reduce the number of preemptions in FPS have been presented [17, 8, 9], where tasks, besides their priorities, are assigned a threshold value such that they can be preempted only by other tasks with priorities higher than the threshold. This approach results, in essence, in a dual priority system which is not directly suitable for legacy systems, where exchanging the scheduler or modifying it by, e.g., adding mutexes to simulate preemption threshold, is not desirable, or not possible.

In this paper, we propose a method to reduce the number of preemptions in FPS systems in which modifications to the original scheduler are not desirable or not even possible. The method can be directly applied on existing FPS systems with high preemption costs as no additional modifications to the underlying scheduler are required. In particular, we provide users of FPS systems the ability to choose a user defined number of preemptions with respect to an eventual cost to pay.

Our method analyzes a set of periodic tasks with periods, priorities and offsets, scheduled by an FPS algorithm, in

*The work described in this paper has been carried out within the EU IST project FIRST.

order to identify the number of preemptions that can occur at run-time. The basic idea is to reassign attributes to the tasks such that the tasks will execute feasibly at run-time while achieving a lower number of preemptions. To resolve a preemption between two task instances, we either swap the priorities or we force the task instances to be released simultaneously (e.g., by reassigning offsets).

Since reassigning attributes to a particular task instance may lead to inconsistent attributes for the instances of the same task, we create artifact tasks for the task instances to solve the inconsistency. In a recent paper [4] we transformed off-line schedules into FPS schemes. In [1], the authors derived response time bounds for tasks with offsets information and introduced an optimal priority ordering algorithm. In this paper, however, we strictly focus on reducing the number of preemptions in existing FPS systems consisting of tasks with periods, priorities and offsets, without modifying the underlying scheduler.

By choosing to eliminate preemptions one by one, we may lose optimality, since, depending on which preemption is chosen to be eliminated first, the outcome of the method may differ. In this paper we use a global approach to detect preemption dependencies and to selectively choose a user-defined level of preemptions with respect to the trade-off between the number of preemptions and the cost to pay, i.e., the number of new artifacts eventually created and the level of decreased flexibility. To do so, we construct a preemption dependency graph that comprises all possible steps we can perform to eliminate preemptions, and all corresponding states representing the new number of preemptions achieved by the new task attributes, and the cost to pay.

The rest of the paper is organized as follows: in section 2 we give an overview of the proposed method. Section 3 describes the basic approach to solve a single preemption. The algorithm proposed to reduce the number of preemptions is described in section 4 followed by a simple example (section 5) and performance evaluation (section 6). Section 7 concludes the paper.

2. Method overview

In this paper we provide an off-line method to analyze the preemptions which can occur when a set of tasks is scheduled by FPS, and to reduce the number of preemptions by reassigning attributes to the tasks.

Problem statement Given a set of periodic tasks with periods, worst case execution times (wcet), offsets, and priorities, schedulable by a standard FPS algorithm, we want to provide new sets of feasible attributes such that the tasks will achieve a lower number of worst case preemption scenarios if scheduled by the same scheduling algorithm. In particular, we want to provide for the ability to chose a user-defined number of preemptions with respect to the cost to pay.

Task model In our approach, we assume that the tasks are periodic, with attributes suitable for fixed priority scheduling and schedulable by a preemptive fixed priority scheduling algorithm. First we present the terminology we use in the rest of the paper.

- we denote tasks as $T_i,\ i \in \{1, 2, \ldots\}$
- T_i^j is the j^{th} instance of T_i, $j \in \{1, 2, \ldots, \frac{LCM}{P(T_i)}\}$
- $p(T_i)$ is the period of T_i
- $prio(T_i)$ and $offset(T_i)$ are the priority and offset of T_i
- $dl(T_i)$ is the deadline of T_i
- $c(T_i)$ is the worst case execution time (wcet) of T_i
- $rel(T_i^j)$ is the release time of T_i^j
- we define $start(T_i^j)$ and $finish(T_i^j)$ as the actual start and finishing time of T_i^j, derived in the off-line analysis, assuming that all tasks execute for wcet.

Furthermore, we assume that the deadline of each task is less than or equal to its period and tasks do not suspend themselves.

For a given set of periodic tasks scheduled by FPS, we first perform an off-line preemption analysis assuming the task executions for wcet. The preemption analysis takes into account even *potential preemptions* that may occur if tasks execute for less than wcet. By that, we detect the points in time at which a preemption occur. For each preemption we have a *preempting task instance* and a *preempted task instance*. Depending on the processor utilization during the time interval in which the preempting/preempted task instances can feasibly execute, we attempt to eliminate the preemption by forcing the execution of either the preempted or the preempting instance such that they do not preempt each other while still executing feasibly.

In our method, we reassign attributes, i.e., priorities or offsets, to the individual instances. For example, assume that the i^{th} instance of a high priority task A, A^i, is preempting the j^{th} instance of a low priority task B, B^j, at time t. In this case, we can either reassign A^i a lower priority than B^j, or we can attempt to prevent B^j from executing before time t, i.e., to reassign B^j an offset such that B^j will be released at time t. At the same time, we have to make sure that an attribute reassignment, i.e., new priority or new offset, will not prevent any task from completing before its

original deadline. If both options will cause any deadline miss, we can not solve the preemption. We perform a standard response time analysis to investigate the feasibility of the tasks after reassigning attributes, [16, 12].

When we reassign priorities, i.e., when we swap the priorities of two task instances which preempt each other, we consider the priority relations of the rest of the tasks as well. We do so by solving the new priority relation together with the old ones by using integer linear programming (ILP). Particular task priorities can be selected to not be changed (with respect to the other task priorities) by simply reformulating the ILP representation. In the same way, particular task offsets can be specified to remain unchanged. However, that will reduce the possibilities of eliminating preemptions.

Preemption reduction cost Since reassigning attributes to a particular task instance may lead to inconsistent attributes for the instances of the same task, we create artifact tasks for the task instances to solve the inconsistency. The number of artifacts created to achieve a particular number of preemptions is one of the costs we have to pay. However, by using ILP, we ensure that the number of eventually created artifacts is minimal.

Another cost is given by the reduced flexibility in the case we reassign offsets and decrease the time interval in which an instance may execute feasibly. We denote the *target window* of a task instance as the time interval in which the instance has to execute and complete in order to execute feasibly, e.g., the target window of a particular instance T_i^j of a task T_i, is the time interval between its release time, i.e., offset or start of the period, $rel(T_i^j)$, and its deadline, $dl(T_i^j)$:

$$TW(T_i^j) = [rel(T_i^j), dl(T_i^j))$$

We use the number of decreased target windows as a measurement unit for the decreased flexibility introduced by our method.

Since resolving a preemption by reassigning attributes may solve or introduce another one, as the execution pattern of several tasks may change, we construct a *preemption dependency tree* to detect all feasible task sets schedulable by the original fixed priority mechanism, which will execute achieving a lower number of preemptions. By keeping track of the preemption reduction cost, we provide for the ability to chose a user-defined number of preemptions with respect to the cost to pay.

3. Solving a single preemption

In this section we describe our basic approach to solve one particular preemption.

Preemptions: In our off-line preemption analysis we assume that tasks execute for wcet. However, at run-time, tasks will most likely execute for less than wcet, implying a different number of preemptions compared to the ones detected by our off-line method. Our goal is to detect all possible preemptions that may occur at run-time. Hence, we divide the preemptions we attempt to solve in two major categories:

Initial preemptions - are detected in the off-line analysis, i.e., a high priority task instance is *initially preempting* a low priority task instance (figure 1).

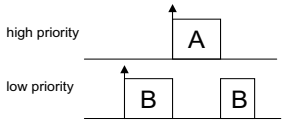

Figure 1. An off-line detected initial preemption

Potential preemptions - can occur at run-time due to task executions less than wcet. In figure 2 a) we can see that if tasks execute for wcet, no preemption will occur. However, in this situation we consider task A *potentially* preempting task B since, if task C, that delays the execution of B, is executing for less then wcet, then B can start executing earlier, i.e., before the release time of A, and will actually be preempted by A (figure 2 b)).

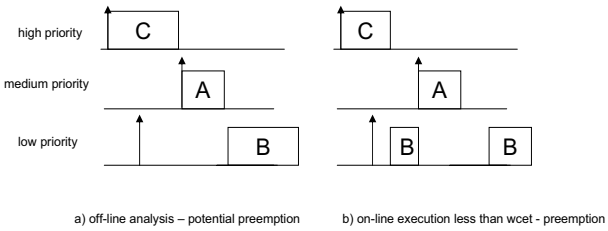

Figure 2. An off-line detected potential preemption

Consequently, in order to cover both types of preemptions, we define the preemptions we attempt to solve as following: T_i^m is preempting T_j^n if the following three conditions hold:

$$prio(T_i^m) > prio(T_j^n) \quad (1)$$

$$rel(T_i^m) > rel(T_j^n) \quad (2)$$

$$finish(T_j^n) > rel(T_i^m) \quad (3)$$

Hence, it is sufficient to eliminate any of the conditions to ensure that the preemption is avoided.

3.1. Solving a preemption by eliminating the first condition

The basic idea is to swap the priorities of the preempting and preempted instances while taking into consideration the priority relations between all task instances as well. We do so by, first, breaking down the original priority relations to the instance level in form of priority inequalities, and then, we solve the inequalities by using *integer linear programming* (ILP).

Preparing step: Before attempting to solve any preemption, we break down the original priority relations, $prio(T_i) > prio(T_j)$, to the instance level. The reason for expressing the original priority relations between the tasks by priority relations between task instances is that our method attempts to solve preemptions by reassigning attributes to individual task instances by using ILP. We first introduce the concept of *interference* between the execution of the task instances. The condition of interference in (5) is used to prevent priority relations between task instances that do not interfere with each other from over constraining the inequality system, as using ILP to solve an over-constrained inequality system, will result in an suboptimal solution. For instance, if there is a priority relation between two tasks A and B, e.g., $prio(A) > prio(B)$, but no actual interference between, e.g., the fifth instance of A and the first instance of B, then there is no need to add the priority inequality $prio(A^5) > prio(B^1)$ to our representation of the priority relations. Hence, we express the original priority relations among the tasks as following:

$$\begin{array}{rcl} \forall i,j,m,n,\ if & \mathbf{interference}(T_i^m, T_j^n) & == 1, \\ replace & prio(T_i)\ \mathbf{op}\ prio(T_j) & by \\ & prio(T_i^m)\ \mathbf{op}\ prio(T_j^n) & \end{array} \quad (4)$$

where

$$\mathbf{op} = \begin{cases} >, \ if\ prio(T_i) > prio(T_j) \\ <, \ if\ prio(T_i) < prio(T_j) \\ =, \ otherwise \end{cases}$$

The interference between two task instances, T_i^m and T_j^n is defined in (5).

$$\mathbf{interference}(T_i^m, T_j^n) = \quad (5)$$

$$\begin{cases} 1, \ if \begin{cases} rel(T_i^m) \leq rel(T_j^n)\ and\ dl(T_i^m) > rel(T_j^n) \\ or \\ rel(T_i^m) \geq rel(T_j^n)\ and\ dl(T_j^n) > rel(T_i^m) \end{cases} \\ 0, \ otherwise \end{cases}$$

ILP formulation Once the priority inequalities between the task instances are derived, we simply search the inequality system for the priority relation between the preempted and preempting task instances, and reverse the inequality. Then, we solve the new inequality system by using integer linear programming (ILP) in order to find a feasible priority assignment. By using ILP, we make sure that the rest of the priority relations between the task instances will not change, even though the individual priority values may change.

The potential cost introduced in this step is an increased number of tasks as a result of priority inconsistencies, i.e., we may have to create new tasks new tasks for the instances with eventual inconsistent priorities. However, the goal function of the ILP solver is formulated to find a solution to the new priority inequalities and to minimize (if any) the number of tasks instances with inconsistent attributes [4]. If the inequality solver yields no solution or, if the schedulability test finds the task unschedulable due to the new attributes, then we can not solve this particular preemption by reassigning priorities.

3.2. Solving a preemption eliminating the second condition

The next alternative we have is to force the task instances to be released simultaneously. That is, to reassign T_j^n a release time (offset) equal to $rel(T_i^m)$. From the definition of the preemptions we aim to solve, (2), we know that $rel(T_j^n) < rel(T_i^m)$, so, in essence, we decrease the time interval in which the instance can feasibly execute, i.e., its target window, from $[rel(T_j^n), dl(T_j^n))$ to $[rel(T_i^m), dl(T_j^n))$. Then, we perform the schedulability analysis to verify the schedulability of all task. However, by reassigning offsets, i.e., by postponing the release time of a task instances, we decrease the flexibility of that particular instance. Thus, the cost introduced in this step is, besides eventual new tasks resulted from offset inconsistencies, a decreased flexibility for particular tasks. If the task set is found unschedulable, we can not solve this particular preemption in this step.

3.3. Solving a preemption eliminating the third condition

The last possibility we have to solve a particular preemption is to reassign attributes such that the third condition no longer holds.

Proposition 1 If a task instance T_j^n is off-line detected to be preempted by at least one task instance T_i^m, we define a *preemption block* of T_j^n, $p\_block(T_j^n)$, as the time interval between $start(T_j^n)$ and $finish(T_j^n)$. The length of $p\_block(T_j^n)$ is constant regardless from the execution order

of the task instances in the block, assuming task executions for wcet.

Hence, a sufficient condition to ensure that the third condition no longer holds is to assign T_i^m an earliest start time equal to $finish(T_j^n) - c(T_i^m)$, i.e., to force the preempted instance to execute at the end of the preemption block.

$$rel(T_i^m) = finish(T_j^n) - c(T_i^m)$$

Proof If T_i^m is forced to execute at the end of the preemption block of T_j^n, it will leave an idle time between $start(T_j^n)$ and $(finish(T_j^n) - c(T_i^m))$ equal to its computation time, $c(T_i^m)$. That time will be used by any of the instances all ready executing in $p\_block(T_j^n)$, so the preemption block will end $c(T_i^m)$ time units earlier. Hence, T_j^n will finish before T_i^m is released and the preemption will be eliminated. □

As in the previous step (3.2), the cost introduced is given by both increased number of tasks as well as decreased flexibility.

However, solving another preemption may turn our 'unsolvable' preemption solvable, or may even solve it automatically as the execution behavior of several tasks changes. In section 4 we show how we take advantage of this phenomena by recursively investigating all possible effects caused by solving a preemption.

3.4. Artifact tasks

By reassigning attributes to eliminate preemptions, we may end up with inconsistent priorities or offsets for different instances of the same task. To solve this side-effect of our proposed preemption reduction method, we split the task with the inconsistent attributes into a number of new periodic, fixed priority tasks and, by that, we create a number of *artifact tasks*, each of them with FPS attributes. The number of new created artifact tasks is the cost we have to pay to reduce the number of preemptions. The instances of the new tasks comprise all instances of the original task.

If at least one of the instances of a task T_i has been reassigned new attributes, i.e., priorities or offsets, and, by that, creating inconsistent task attributes, we transform the instances T_i^j, $j \in [1, \frac{LCM}{period(T_i)}]$, in new tasks τ_{ij} as following:

$$\begin{aligned}
period(\tau_{ij}) &= LCM \\
priority(\tau_{ij}) &= \begin{cases} new\ priority,\ if\ reassigned \\ unchanged,\ otherwise \end{cases} \\
offset(\tau_{ij}) &= \begin{cases} new\ offset,\ if\ reassigned \\ (j-1) * period(T_i),\ otherwise \end{cases} \\
deadline(\tau_{ij}) &= unchanged
\end{aligned}$$

From the implementation point of view, when we create artifacts for the instances of a task T_i, we create copies of the task control block of T_i pointing at the same code and data segment, but with different FPS attributes.

4. Reducing the number of preemptions - the global approach

In this paper, our goal is to reduce the number of preemptions in FPS without modifying its basic mechanism. As described in section 3, we attempt to eliminate (solve) a preemption by task attribute reassignment, i.e., priority or offset reassignment, while guaranteeing the feasibility of the task set.

However, if we use a constructive approach, i.e., if we attempt to eliminate the preemptions in a particular order, we may lose optimality in terms of finding the minimum number of preemptions. Depending on which particular preemption we choose to resolve first, the results, in terms of the number of preemptions achieved and/or the number of artifacts created, may differ. That is due to the fact that resolving a preemption by reassigning task attributes may resolve or introduce additional ones, as the execution pattern of several tasks may change. Therefore, for each feasible task set, we choose to solve all preemptions by all approaches, i.e., we attempt to eliminate each preemption yielded by a particular task set by all three approaches (3.1, 3.2 and 3.3).

4.1. Preemption dependency tree

To find the optimum desired level of preemptions with respect to the artifacts to be created, we construct a *preemption dependency tree* that comprises all possible steps we can perform to eliminate preemptions.

The root of the tree consists of the original task attributes together with the corresponding off-line detected preemptions. Each preemption point is an edge from a node, i.e., solving that particular preemption will give us another node, with another task set and new preemption points. Thus, by eliminating each of the preemptions, we obtain one, two or three new nodes, i.e., one for each successfully eliminated condition. If all approaches yield feasible task sets, we obtain 3 new nodes.

At the same time, we keep track of the cost we have to pay for each preemption we succeed to solve, i.e., the number of artifacts and the number of reduced target windows. Thus, each new created node in the graph will contain:

- The new feasible task attributes and the new preemptions yielded by the tasks in this configuration.
- The cost to pay, i.e., the number of artifacts and the number of reduced target windows.

Since one of the advantages obtained by using FPS is flexibility, the more appropriate way to solve a preemption is to reassign priorities. That is due to the fact that solving a preemption by reassigning offsets implies reducing the target windows of the task instances. However, our goal is to find the set of task attributes that yields the minimum number of preemptions while guaranteeing feasibility. Therefore, we attempt to solve each preemption by eliminating each of the three conditions in order to find all possible preemption sets yielded by corresponding task attributes.

The dependency tree is constructed by recursively using the 3 approaches described in sections 3.1, 3.2,and 3.3, followed by a new preemption analysis each time a preemption is resolved. The feasible task sets contained in the nodes of the tree are schedulable by the same original fixed priority mechanism.

A user-defined state can be selected by performing a basic tree-search, with respect to the trade-off between the number of preemptions, the number of artifacts and the number of reduced target windows.

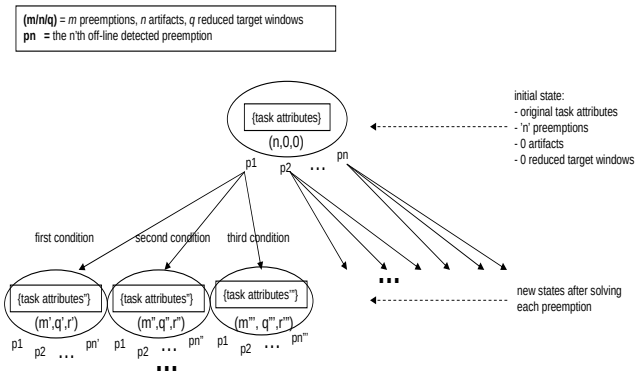

Figure 3. Preemption dependency tree

5. A simple example

We illustrate the proposed preemption reduction method with an example. We assume the set of FPS tasks described in table 1.

$Task$	p	c	prio
A	5	1	3
B	10	3	2
C	20	8	1

Table 1. Original FPS tasks

The task attributes are period (p), worst case execution time (c) and priority (prio). For an easier and more intuitive reading, we assume that the offsets are equal to the start of the period and a higher value represents a higher priority. In this example we assume that the worst case context switch time has been taken into account when calculating the worst case execution times for each task.

From the FPS schedule shown in figure 4, we can see that four preemptions can occur at run-time, if the tasks execute for wcet, when scheduled by FPS. Hence, the root of the preemption dependency tree consist of 3 tasks, A, B and C with the original attributes, 4 preemptions, 0 artifacts, and 0 modified target windows.

The first preemption is detected at time t=5 when A^2 is preempting C^1. We first attempt to solve the preemption by swapping the priorities of A^2 and C^1. To do so, we first derive the target windows of the task instances (table 2) and, then, we break down the original priority relations between the tasks to the instance level. By analyzing the overlapings between the target windows (i.e., the interference between the task instances), we then derive the system of priority inequalities between the individual task instances according to (4). In our example, the original relations $prio(A) > prio(B) > prio(C)$ are transformed as following:

$$\begin{aligned}
prio(A) > prio(B) \quad &\rightarrow \quad prio(A^1) > prio(B^1), \\
&\qquad\quad prio(A^3) > prio(B^2) \\
prio(A) > prio(C) \quad &\rightarrow \quad prio(A^1) > prio(C^1), \\
&\qquad\quad prio(A^2) > prio(C^1), \\
&\qquad\quad prio(A^3) > prio(C^1), \\
&\qquad\quad prio(A^4) > prio(C^1) \\
prio(B) > prio(C) \quad &\rightarrow \quad prio(B^1) > prio(C^1), \\
&\qquad\quad prio(B^2) > prio(C^1)
\end{aligned} \tag{6}$$

Note that the priority inequalities $prio(A^2) > prio(B^1)$, $prio(A^2) > prio(B^2)$, and $prio(A^4) > prio(B^2)$ are not included in the inequality system, since there is no interference (as defined in (5)) between the mentioned instances. Hence, we do not have to take into account the priority relation between them.

$instance\ nr.$	task A	task B	task C
1	[0,5]	[0,10]	[0,20]
2	[5,10]	[10,20]	
3	[10,15]		
4	[15,20]		

Table 2. Target windows for the original task instances

At this point, we just replace the derived priority inequality between A^2 and C^1, i.e., $prio(A^2) > prio(C^1)$,

by $prio(A^2) < prio(C^1)$. By solving the new inequality system using ILP, we obtain new task set consisting of 6 tasks (3 artifacts) with the following priorities: prio(A1)=5, prio(A2)=1, prio(A3)=4, prio(A4)=3, prio(B)=3, prio(C)=2. However, a schedulability test finds the new task set infeasible since the second instance of A will miss its deadline.

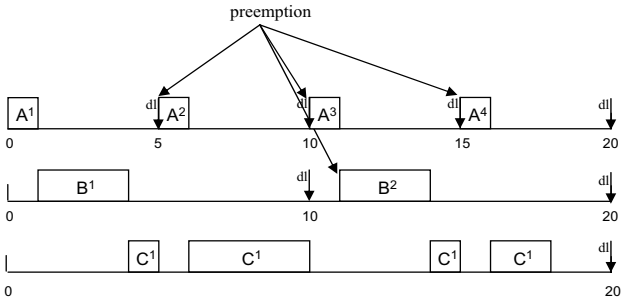

Figure 4. Original FPS schedule: task C is preempted by A and B

The second option we have is to reassign C^1 a release time equal to 5, which is the release time of A^2. This time, the task set is found schedulable. The new node we add to the tree consists of task attributes for 3 tasks, i.e., {*task attributes'*} (Figure 5). The only difference from the original attributes in figure 1 is that C has now an offset equal to 5. The new number of preemptions (detected by performing the off-line preemption analysis) has decreased to 3, we have not introduced artifacts since C has only one instance in LCM and we have reduced one target window ($TW(C^1)$).

The third alternative to solving the preemption is to assign A^2 an offset equal to $finish(C^1) - wcet(A^2) = 17$. However, we can easily see that the new task set will not be feasible since A^2 has a deadline equal to 10. Hence, no new node will be added to the tree.

In the same way we derive the rest of the nodes in the preemption dependency graph, by attempting to solve each preemptions by all three approaches. Due to space limitations, we show only a feasible path in the preemption dependency tree in figure 5. However, when running the example in our simulator, the complete tree consisted of 36 nodes while the solution presented in this section was found after building 8 nodes.

Once the tree is constructed, we can easily traverse it to find the optimum solution. If the choice is to minimize the number of preemptions, one can follow the path shown in figure 5. The final node is obtained after successively

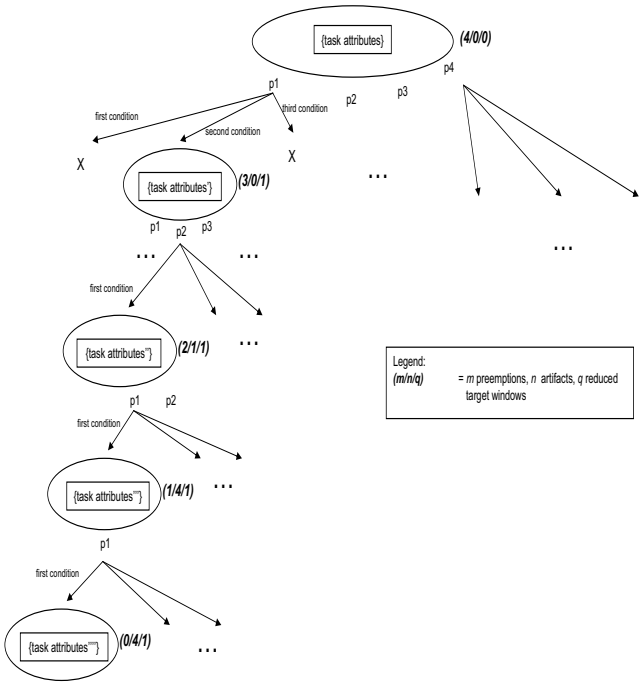

Figure 5. Example: preemption dependency tree

eliminating 4 preemptions: A^2 preempts C^1 (the case we have described above), B^2 preempts C^1, A^3 preempts C^1 and, finally, A^4 preempts B^2. The minimum amount of preemptions, i.e., 0, is yielded by the task attributes shown in table 3, and the cost to pay is 4 artifacts and one reduced target window.

However, the user can choose an intermediate node if the trade-off between the number of preemptions and the number of artifacts is more suitable, e.g., 2 preemptions vs. 1 artifact and 1 reduced target window.

The final set of tasks with new attributes and the FPS schedule are shown in table 3 and figure 6 respectively. In each step, the artifact tasks are created as described in (3.4).

6. Performance evaluation

We have performed a number of experiments to evaluate the efficiency of our method. We have used synthetic tasks with randomly generated attributes, schedulable by FPS.

The experiments were run on each task set until either the tree was complete, a node containing attributes yielding 0 preemptions was found, or for a maximum duration of 2 minutes. The hardware used for the experiments consisted of a P4 PC at 1.9MHz.

$Task$	p	c	offset	prio
A1	20	1	0	2
A2	20	1	5	5
A3	20	1	10	3
A4	20	1	15	1
B1	20	3	0	1
B2	20	3	10	2
C	20	8	5	4

Table 3. The new FPS attributes that yield no preemptions

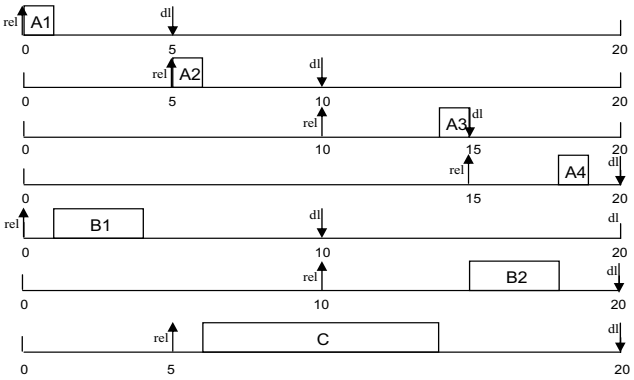

Figure 6. New FPS schedule, zero preemptions

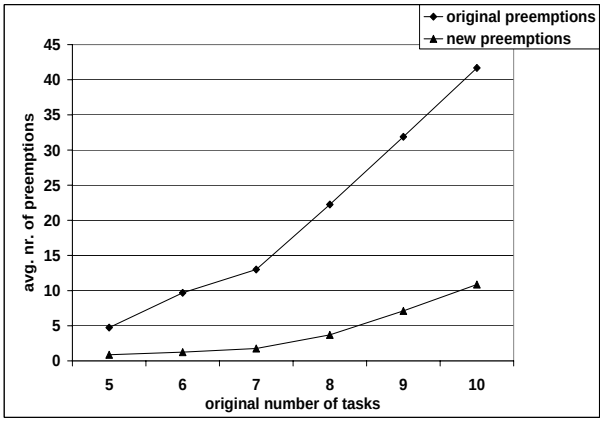

Figure 7. Average preemption reduction

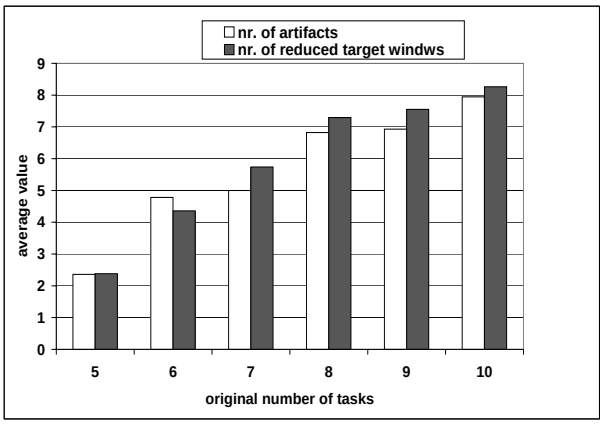

Figure 8. Preemption reduction cost

Each point in the graph was created by performing computations on 50 task sets consisting of 5 to 10 tasks respectively. The LCM for each task set was randomized between 10 to 20 times the number of tasks. The periods were randomized in the interval $[\frac{LCM}{nr.\ of\ tasks}, LCM]$, and the wcet's were chosen to ensure an utilization of at least 0.6. In our experiments, the average utilization was 0.89. The priorities have been assigned according to the RM algorithm and original offsets were set to zero due to the optimality of RM priority assignment. However, our method does not depend on offsets as it relies on release and start times.

When searching the tree for the best solution, the node with the lowest number of preemptions was chosen. If more than one node contained the minimum number of preemptions, the chosen node was the one containing the least number of artifacts.

As we can see in figure 7, the method significantly managed to reduce the worst case number of preemptions for an average cost presented in figure 8. The average increase in the number of tasks introduced by our method is presented in figure 9.

7. Conclusions and future work

In this paper, we have proposed a method to reduce the number of preemptions in FPS without modifying its basic mechanism. Hence, the method is directly applicable to FPS systems in which modifications to the original scheduler are not desirable or not even possible. Assuming the wcet for the tasks, the method analyzes off-line a set of periodic tasks with fixed priorities and offsets, scheduled by FPS, and identifies the number of preemptions that can occur at run-time. Our method reduces the number of preemptions by reassigning task attributes, i.e., priorities and offsets. However, in some cases, the attribute reassignment procedure yields inconsistent attributes for instances of the same task. We solve the phenomena by creating artifacts for the task instances with inconsistent FPS attributes. Hence, no modifications are needed to the original scheduler. We keep the number of artifacts minimal by using ILP. The number of artifacts and the reduced flexibility are the costs

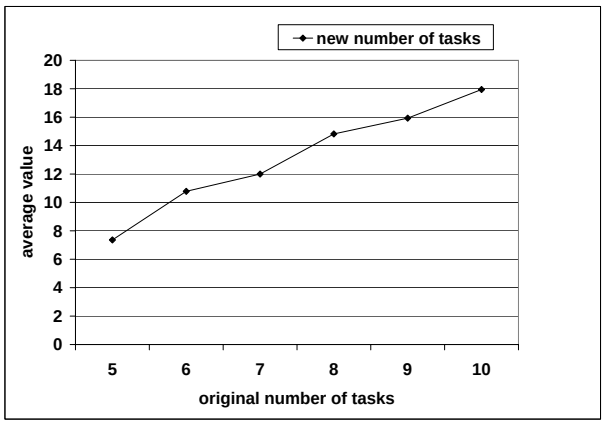

Figure 9. Number of FPS tasks

we have to pay for reducing the number of preemptions.

Since solving a particular preemption changes the execution pattern of several tasks and, thus, may imply introducing or solving other ones, we use a global approach by constructing a preemption dependency tree, to detect all the preemption dependencies and, implicitly, the set of feasible task attributes that yields the minimum number of preemptions. The preemption dependency graph comprises all possible steps we can perform to eliminate preemptions and all corresponding states representing the new task attributes with corresponding new number of preemptions, the number of artifacts to be created and the level of reduced flexibility, i.e., the cost we have to pay. Hence, we provide for the ability to select a user-defined optimum state with respect to the trade-off between the number of preemptions and the cost to pay.

At this point, the model for this work is purely periodic tasks with fixed priorities and offsets. Future work will include investigations of the applicability of the presented technique to systems including tasks with non-periodic behavior.

8. Acknowledgements

The authors wish to express their gratitude to Giorgio Buttazzo, Sanjoy Baruah and Jukka Mäki-Turja for useful discussions, and to the reviewers for their helpful comments on the paper.

References

[1] N. Audsley, K. Tindell, and A. Burns. The end of the line for static cyclic scheduling? In *Proceedings 5th Euromicro Workshop on Real-Time Systems*, pages 36–41, Oulu, Finland, June 1993.

[2] A. Burns, K. Tindell, and A. Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Trans. on Software Engineering*, 21(5):475–80, May 1995.

[3] G. Buttazzo. Rate monotonic vs. EDF: Judgment day. In *Proc. 3rd ACM International Conference on Embedded Software*, Philadephia, USA, Oct 2003.

[4] R. Dobrin, G. Fohler, and P. Puschner. Translating off-line schedules into task attributes for fixed priority scheduling. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, London, UK, Dec. 2001.

[5] J. Echagüe, I. Ripol, and A. Crespo. Hard real-time preemptively scheduling with high context switch cost. In *Proceedings of the 7th Euromicro Workshop on Real-Time Systems*, Odense, Denmark, Mar. 1995.

[6] K. Jeffay and D. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of Real-Time Systems Symposium*, pages 212–221, Dec. 1993.

[7] D. I. Katcher, H. Arakawa, and J. K. Strosnider. Engineering and analysis of fixed priority schedulers. *Software Engineering*, 19(9):920–934, 1993.

[8] S. Kim, S. Hong, and T.-H. Kim. Integrating real-time synchronization schemes into preemption threshold scheduling. In *Proc. 5th IEEE International Symposium on Object-oriented Real-time Distributed Computing*, Crystal City, VA, USA, Apr. 2002.

[9] S. Kim, S. Hong, and T.-H. Kim. Perfecting preemption threshold scheduling for object-oriented real-time system design: From the perspective of real-time synchronization. In *Proc. Languages, Compilers, and Tools for Embedded Systems*, Berlin, Germany, Jun. 2002.

[10] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemtive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, June 1998.

[11] C.-G. Lee, K. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Bounding cache-related preemption delay for real-time systems. *Software Engineering*, 27(9):805–826, Sep. 2001.

[12] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceddings of the Real-Time Symposium*, pages 201–212, Dec. 1990.

[13] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in hard real-time environment. *Journ. of the ACM, 20, 1*, Jan. 1973.

[14] K. Ramamritham and J. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *IEEE*, 82(1):55–67, Jan. 1994.

[15] J. Schneider. Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems, 2000.

[16] K. Tindell. Adding time-offsets to schedulability analysis, internal report, university of york, computer science dept, ycs-94-221.

[17] Y. Wang and M. Saksena. Scheduling fixed priority tasks with preemption threshold. In *In Proceedings, IEEE International Conference on Real-Time Computing Systems and Applications, December 1999*, Dec. 1999.