Online Admission of Non-Preemptive Aperiodic Tasks in Offline Schedules

Stefan Schorr and Gerhard Fohler
Technische Universität Kaiserslautern, Germany
{*stschorr,fohler*}*@eit.uni-kl.de*

***Abstract*—In this paper, we present an approach to extend slot shifting [1] to integrate single non-preemptive aperiodic tasks. The slot shifting algorithm allows to combine the benefits of offline and online scheduling: the offline scheduler handles periodic task sets with complex constraints and the online algorithm integrates aperiodic tasks. For the admission control of the non-preemptive tasks, our algorithm uses precomputed values derived from the offline phase of slot shifting, hence having low memory overhead. Furthermore, our approach allows to minimize the response time of the non-preemptive aperiodic tasks.**

I. Introduction

The scheduling table provided by an offline scheduler allows to intuitively verify the correct system behavior and to give worst case guarantees. The main drawbacks are inflexibility and incapability to handle aperiodic tasks. Online scheduling, on the other hand, offers such run-time flexibility, which is needed to, e.g., provide support for aperiodic tasks. Here, the disadvantage is that these algorithms are tuned to be fast and light-weighted, and hence, do not support the handling of complex constraints. The slot shifting algorithm [1] combines the benefits of both approaches. It is capable of resolving complex constraints using the offline scheduler and to flexibly integrate aperiodic tasks at run-time in a distributed system.

The slot shifting algorithm is designed as a preemptive algorithm. Thus, non-preemptive tasks are not supported or their execution might lead to irregular system behavior. Non-preemptive tasks are required, e.g., to provide secure data access to shared resources of the system, or since many interactions with hardware devices do not allow for delays caused by task preemptions.

The original slot shifting algorithm handles periodic and aperiodic tasks [1] and has been extended later to support sporadic tasks [2]. In a similar approach Pop et al. consider the design of distributed embedded systems implemented with mixed, event-triggered and time-triggered task sets, communicating over bus protocols [9]. Šůcha and Hanzálek describe in [4] an ILP model for offline scheduling of tasks on parallel identical processors. The problem to determine whether a periodic task system is non-preemptively schedulable on a single processor is known to be NP-hard in the strong sense [3]. The non-preemptive scheduling of periodic task systems upon processing platforms comprised of several identical processors is considered in [5]. Baker introduced the notion of *preemption levels* for tasks [7]. Tasks with lower preemption levels cannot preempt a task with higher preemption level. Gai et al. have studied the use of *preemption thresholds* for restricting the number of preemptions in EDF-scheduled systems [8]. Bertogna and Baruah introduced limited-preemption EDF [6], a hybrid limited-preemption scheduling algorithm, that schedules all task systems that can be scheduled by fully preemptive algorithms. The same paper derives an upper bound for the number of times a task is preempted using this algorithm. The problem with all these works is that it either does not feature the slot shifting's capability for online aperiodic task admission, or cannot handle a non-preemptive task at all.

In this paper, we present an extension to the original slot shifting algorithm to support admission of aperiodic non-preemptive tasks. The basic idea is to accommodate a non-preemptive task by shifting the other tasks without harming the already guaranteed tasks.

Our extension uses precomputed values of slot shifting to integrate the non-preemptive tasks, adding only low additional memory demand. The proposed method aims to place the non-preemptive task such that the response time is minimal.

The rest of the paper is structured as follows: the next section describes the basic slot shifting algorithm. Section III analyses the problem and depicts the solution using our algorithm. Section IV gives a detailed example to demonstrate how the algorithm works. The last section discusses the algorithm presented here and concludes the paper.

II. Slot Shifting - Original approach

The original slot shifting algorithm combines offline and online scheduling to provide for the efficient run-time handling of aperiodic tasks on top of a distributed offline schedule. Slot shifting extracts information about unused resources and leeway in an offline schedule and uses this information to add aperiodic tasks feasibly, i.e., without violating constraints of the already scheduled tasks. It consists of an offline scheduling mechanism for complexity reduction and an online mechanism for efficient inclusion of dynamically arriving tasks.

A. Offline complexity reduction

In a first step, an offline scheduling algorithm creates scheduling tables for the periodic tasks and allocates the tasks to nodes. The notion of time used by the algorithm is discrete and the granularity is given in units of slot length. The algorithm resolves the precedence constraints by ordering the task executions accordingly.

In the next step, the algorithm calculates earliest start times and latest finishing times for all tasks, thus the execution sequence constructed by the offline scheduler keeps the precedence order. The deadlines of tasks are then sorted for each node and the schedule is divided into a set of *disjoint execution intervals* for each node. The execution intervals are enumerated in ascending order. Each deadline calculated for a task T_i, $dl(T_i)$, defines the end of an interval I_i, $end(I_i)$. Several tasks with the same deadline constitute one interval. The minimum of the release times of the tasks constituting one interval is called earliest start time of an interval, $est(I_i)$. The start of an interval, $start(I_i)$, is defined as the maximum of its earliest start time and the end of the previous interval. In that way, the resulting scheduling tables list fixed start and end times of task executions and tasks can be shifted arbitrarily within the interval boundaries. However, a task can execute outside its execution interval ("borrowing" mechanism), but not before its earliest start time.

The following formula defines *spare capacities* for all intervals to represent the amount of available resources for aperiodic tasks:

$$sc(I_i) = |I_i| - \sum_{T_i \in I_i} WCET(T_i) + min(sc(I_{i+1}), 0) \quad (1)$$

The spare capacity of an interval is given by its length decreased by the sum of the worst case execution times (WCET) of tasks belonging to the corresponding interval and the amount of slots "lent" to the subsequent interval.

B. Online mechanism

The online scheduling is performed on each node independently and the online scheduler is invoked after each slot at run-time. First, it checks whether aperiodic tasks have arrived during the last slot. Soft aperiodic tasks can be executed when the current spare capacity is greater than zero. If hard aperiodic tasks have arrived, the guarantee algorithm will be invoked: the acceptance test sums up the spare capacities of the current and all succeeding intervals up to the deadline of the task to be guaranteed and checks whether it can be accommodated without conflicts with already guaranteed tasks. If there is sufficient spare capacity to fit in the task, but the deadline of the task does not coincide with the end of an interval, then the interval is split into two intervals and the spare capacities are updated to prevent other tasks from interfering. Spare capacities are maintained in such a way that the current interval never has a negative spare capacity at run-time.

Finally, depending on the status of the ready queue and the spare capacities, either a soft aperiodic or a guaranteed task is executed and the spare capacities are updated. If there are no tasks to execute, then the processor will be left idle.

III. Method

This section is focused on our approach to extend the original slot shifting algorithm to integrate a single non-preemptive task into an arbitrary and feasible offline schedule. It starts with a brief analysis of the given problem and then presents our solution.

A. Analysis

The main difference between adding a preemptive aperiodic task and adding a non-preemptive aperiodic task lies in the procedure to look for a suitable time interval for the execution of the task. The acceptance test has to find not only enough empty slots, but it has to find them in a consecutive order. Once a possible interval has been identified, the algorithm has to ensure that the non-preemptive task is guaranteed and cannot be interrupted by other tasks.

Periodic tasks have earliest start times and deadlines, which specify their largest possible execution windows. In principle, each task can start at the beginning of this execution window or start as late as possible. This creates a number of permutations how all periodic tasks can execute. To accommodate a non-preemptive aperiodic task T_{NP}, among all these permutations, there has to be one permutation that provides an empty interval with the minimum length $WCET(T_{NP})$.

The offline scheduler of the original slot shifting algorithm delivers scheduling tables for all nodes and all their periodic tasks in the distributed system, based on the earliest start times and deadlines of the tasks. Each schedule is divided into disjoint execution intervals as described in section II. Each of these intervals features a spare capacity value which expresses the amount of capacity available in the corresponding interval for aperiodic executions. Periodic tasks may be shifted inside their execution intervals to allow for admission of aperiodic tasks. Tasks are not bound to the execution intervals defined by the offline scheduler but by their earliest start time and their deadline.

B. Algorithm

As the original slot shifting, our approach is split into an offline and an online part. In the offline part, periodic tasks are handled exactly as described in section II: the offline scheduler constructs a feasible schedule, defines execution intervals and calculates the corresponding spare capacities.

In the online part, we distinguish between preemptive and non-preemptive aperiodic tasks. For the former, the same rules as in the original version apply for the acceptance test, for the maintenance of the spare capacities and the execution. For the non-preemptive aperiodic task we propose a new acceptance test, described below.

Our algorithm can only guarantee one non-preemptive tasks after another. For each of them, the acceptance test iterates through the predefined intervals looking for a suitable interval for the non-preemptive task.

First, the algorithm selects the current interval as the considered interval I_c and tries to accommodate the non-preemptive task T_{NP}. When there is a sufficient time interval, the non-preemptive task is accepted, otherwise the succeeding interval is defined as the next I_c. The acceptance test aims at determining two points in time t_{start} and t_{end}, which represent the start and the end of a possible interval for the non-preemptive task. The algorithm separates the tasks in I_c as much as possible from those of the preceding intervals. All tasks executing prior to the currently considered interval I_c will be started as early as possible and all tasks in I_c will be delayed in order to create time in between for the execution of the non-preemptive task T_{NP}. Additionally, the algorithm shifts tasks from I_c to the left, if they can be shifted before a preceding task which determines t_{start}. If I_c reaches the interval containing the deadline of the non-preemptive task, the successor interval has a positive spare capacity and the test still fails to create sufficient space, then the acceptance test will reject the non-preemptive task T_{NP}. In case I_c is the interval containing the deadline of the non-preemptive task and the successor interval is borrowing slots, the algorithm has to iterate up to the last interval with non-positive spare capacity, before it can refuse the non-preemptive task.

In order to derive a final equation for t_{start} and t_{end}, we first define a set B of tasks in equation 2. This set contains all tasks whose earliest start times and deadlines are before the start of the currently considered interval and whose executions start at their earliest start times. The start of the execution is calculated iteratively by equation 3: the execution of a task starts either at its earliest start time or after the previous task finished, whichever is later.

$$B := \{T_i | start(T_i) = est(T_i) \wedge dl(T_i) \leq start(I_c), \forall i\} \tag{2}$$

with

$$start(T_i) := max\Big\{est(T_i), \big(start(T_{i-1}) + WCET(T_{i-1})\big)\Big\} \tag{3}$$

In equation 4, we identify the last task to start among the tasks from the set B: T_{ls}.

$$T_{ls} \in B : est(T_{ls}) > est(T_i), \forall T_i \in B \setminus \{T_{ls}\} \tag{4}$$

So far, only the tasks prior to I_c have been considered. However, there are scenarios where also tasks from inside I_c have to be shifted. Figure 1 shows such an example. All tasks up to those inside I_c are assumed to start as early as possible, i.e. at their earliest start time. Hence, if multiple tasks can start at the same time a queue of tasks will be formed as shown in the figure. t_{end1} defines the point in time where the interval for aperiodic execution ends. In this example, the length of this interval can be increased even more, since there is a task in I_c whose execution can be shifted prior to the start of task T_{ls}, resulting in a new end of the interval, t_{end2}. The possible execution interval for non-preemptive tasks hence expands from $[t_{start}, t_{end1}]$ to $[t_{start}, t_{end2}]$. We define a set M of tasks which belong to I_c and whose shifting to their earliest start times increases the length of the interval

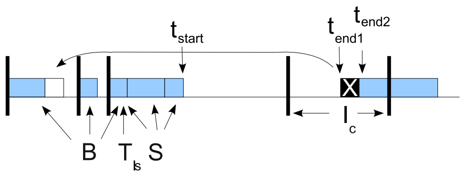


Figure 1. Example clarifying t_{start} and t_{end}.

$[t_{start}, t_{end}]$:

$$M := \{T_i \big| dl(T_i) = end(I_c) \wedge est(T_i) < est(T_{ls})\} \tag{5}$$

If M is non-empty, the following approach is used: the task of M with the smallest earliest start time is selected to start as early as possible. The sets B and M as well as T_{ls} are updated and these steps are repeated until M is empty. As shown in Figure 1, task T_{ls} can be succeeded by several other tasks. The set S is defined as the "succeeding tasks" united with task T_{ls}:

$$S := \{T_i \big| est(T_{ls}) \leq est(T_i) \wedge dl(T_i) \leq start(I_c), \forall i\} \tag{6}$$

Now, t_{start} calculates as the earliest start time of task T_{ls} plus the worst case execution times of all tasks in S:

$$t_{start} := est(T_{ls}) + \sum_{T \in S} WCET(T) \tag{7}$$

and t_{end} is given as:

$$t_{end} := start(I_c) + min\{sc(I_c), \big(dl(T_{NP}) - start(I_c)\big)\} \tag{8}$$

Once these delimiters are known, the acceptance test only has to verify whether the worst case execution time of the non-preemptive task T_{NP} is less or equal than the length of the interval $[t_{start}, t_{end}]$:

$$\begin{aligned} WCET(T_{NP}) \leq & t_{end} - t_{start} \\ = & \Big(start(I_c) + min\{sc(I_c), \\ & dl(T_{NP}) - start(I_c)\}\Big) - \\ & \Big(est(T_{ls}) + \sum_{T \in S} WCET(T)\Big) \end{aligned} \tag{9}$$

IV. EXAMPLE

Figure 2 shows an offline schedule consisting of four tasks T_W, T_X, T_Y and T_Z with their earliest start times, worst case execution times, deadlines and the corresponding intervals. Part A of the figure presents the original properties of the offline schedule, which are the basis for our algorithm as described in section III: The first interval I_1 has a length of 5 and contains T_W with a worst case execution time of 1. Borrowing takes place in I_1, which results in a spare capacity of 3. The next interval contains T_X and makes use of the borrowing mechanism, hence the spare capacity of I_2 is -1. The third interval holds T_Y with worst case execution time 2, which leads to a spare capacity of 2. Interval I_4 is empty and the last interval, I_5, accommodates T_Z with WCET 1 and hence has a spare capacity of 2.

We assume the arrival of an aperiodic non-preemptive task T_{NP} with deadline 16 and WCET 9, when $t_{current}$ equals 1. In the first step, the algorithm defines $I_c = I_1$, $B = \{T_W\}$, $T_{ls} = T_W$, $M = \{\}$, $S = T_W$, $t_{start} = 1$ and $t_{end} = 4$. Since the timespan between t_{start} and t_{end} is not enough for task T_{NP}, the algorithm defines $I_c = I_2$ in the next step. The algorithm recalculates B, T_{ls}, M and S, which all do not change. The task T_{NP} cannot be scheduled starting immediately at t = 4 without T_X missing its deadline, thus I_c is reassigned to interval I_3. Now, tasks T_W and T_X start as early as possible, the spare capacities are updated and $B = \{T_W\}$, $T_{ls} = T_W$, $S = \{T_W, T_X\}$, $M = \{\}$, $t_{start} = 4$ and $t_{end} = 9$, as it is shown in part B of Figure 2.

Due to the lack of successive space for T_{NP}, the algorithm repeates these steps and hence prepones the start of the execution of T_Y in the same manner as T_X when it shifts I_c to the 4th interval. The updated spare capacities and the values for t_{start} and t_{end} are depicted in part C of the Figure.

In the final step, the algorithm assigns I_c to the last interval and sets t_{end} to 15, adds T_{NP} to the schedule and updates all values as shown in part D of Figure 2.

V. Discussion and Conclusion

In this paper, we presented an approach to extend slot shifting, a method which combines table driven and online scheduling, to integrate an aperiodic non-preemptive task into an arbitrary and feasible schedule. Our algorithm does not create a new scheduling table but only modifies the existing one and hence introduces low memory overhead. Moreover, the proposed algorithm allows to minimize the response time of the non-preemptive task.

We considered two different approaches to guarantee the non-preemptive task: one way is to reprogram the interrupt timer. The second is to set the spare capacity to zero while the non-preemptive task executes. Using a temporary variable, a meaningful value for the spare capacity can be restored, once the non-preemptive task finishes its execution.

Let K be the number of jobs whose start times lie between $t_{current}$ and the deadline of the non-preemptive task. In the worst case, our algorithm will visit all those m jobs and shift them which triggers a recalculation of the sets B, S, M and of T_{ls}. Since recalculating those sets has a complexity of $O(K)$ the overall complexity is $O(K^2)$.

In future work we will extend our approach to offer support for multiple non-preemptive tasks simultaneously.

References

[1] G. Fohler, *Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems*, Proc. of the 16^{th} Real-Time Systems Symposium, Pisa, 1995

[2] D. Isovic and G. Fohler, *Efficient Scheduling of Sporadic, Aperiodic, and Periodic Tasks with Complex Constraints*, Orlando, Florida, USA, 2000

[3] K. Jeffay, D. Stanat and C. Martel, *On non-preemptive scheduling of periodic and sporadic tasks*, Proc. of the 12^{th} Real-Time Systems Symposium, San Antonio, Texas, 1991

[4] P. Šůcha and Z. Hanzálek, *Cyclic Scheduling of Tasks with Unit Processing Time on Dedicated Sets of Parallel Identical Processors*, Proc. of the 3rd Multidisciplinary International Conference on Scheduling: Theory and Application, Paris, 2007

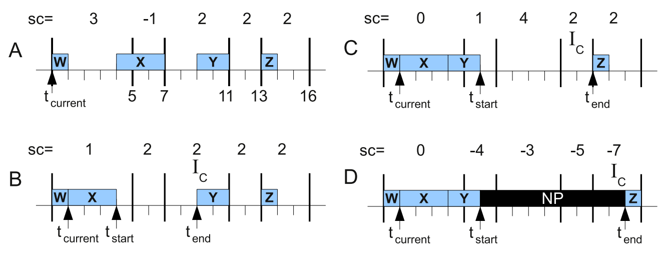
[5] S. Baruah, *The Non-preemptive Scheduling of Periodic Tasks upon Multiprocessors*, Real-Time Systems Vol.32, 2006

[6] M. Bertogna and S. Baruah, *Uniprocessor Scheduling of Sporadic Task Systems under Preemption Constraints*, preprint

[7] T.P. Baker, *Stack-Based Scheduling of Real-Time Processes*, Real Time Systems: The International Journal of Time Critical Computing 3, 1991

[8] 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*, Proc. of the IEEE Real-Time Systems Symposium, 2001

[9] T. Pop, P. Eles and Z. Peng, *Schedulability Analysis for Distributed Heterogeneous Time/Event Triggered Real-Time Systems*, 15^{th} Euromicro Conference on Real-Time Systems, Porto, Portugal, 2003, pp. 257-266



Task	Earliest Start Time	Deadline	WCET
W	0	5	1
X	0	7	3
Y	0	11	2
Z	13	16	1

Figure 2. Example schedule with four tasks.