Handling Aperiodic Tasks in Diverse Real-Time Systems via Plug-Ins

Tomas Lennvall, Gerhard Fohler, and Björn Lindberg[*]
Department of Computer Engineering
Mälardalen University, Sweden
{tomas.lennvall,gerhard.fohler}@mdh.se

Abstract

Functionality for various services of scheduling algorithms is typically provided as extensions to a basic algorithm. Aperiodic task handling, guarantees, etc., are integrated with a specific basic scheme, such as earliest deadline first, rate monotonic, or off-line scheduling. Thus, scheduling services come in packages of scheduling schemes, fixed to a certain methodology.

A similar approach dominates operating system functionality: implementation of the actual real-time scheduling algorithm, i.e., take the decisions which task to execute at which times to ensure deadlines are met, are intertwined with kernel routines such as task switching, dispatching, and bookkeeping to form a scheduling/dispatching module.

Consequently, designers have to choose a single scheduling package, although the desired functionality may be spread over several ones. Instead, there is a need to seamlessly integrate new functionality with a developed system, enabling designers to choose the best of various packages.

In this paper, we propose the use of a plug-in approach to add functionality to existing scheduling schemes and provide for easy replacement on the operating system level. In particular, we present an architecture to disentangle actual real-time scheduling from dispatching and other kernel routines with a small API, suited for a variety of scheduling schemes as plug-ins. We detail two plug-ins for aperiodic task handling and how they can extend two target systems, table-driven and earliest deadline first scheduling using the presented approach.

1. Introduction

Scheduling algorithms have been typically developed around central paradigms, such as earliest deadline first (EDF) [6], rate monotonic (RM)[6], or off-line scheduling. Additional functionality, such as aperiodic task handling, guarantees, etc., is typically provided as extensions to a basic algorithm. Over time, scheduling packages evolved, providing a sets of functionality centered around a certain scheduling methodology.

EDF or fixed priority scheduling (FPS), for example, are chosen for simple dispatching and flexibility. Adding constraints, however, increases scheduling overhead [12] or requires new, specific schedulability tests which may have to be developed yet. Off-line scheduling methods can accommodate many specific constraints and include new ones by adding functions, but at the expense of runtime flexibility, in particular inability to handle aperiodic and sporadic tasks.

A similar approach dominates operating system functionality: implementation of the actual real-time scheduling algorithm, i.e., take the decisions which task to execute at which times to ensure deadlines are met, are intertwined with kernel routines such as task switching, dispatching, and bookkeeping to form a scheduling/dispatching module. Additional real-time scheduling functionality is added by including or "patching" this module. Replacement or addition of only parts is a tedious, error prone process.

Consequently, a designer given an application composed of mixed tasks and constraints has to choose which constraints to focus on in the selection of scheduling algorithm; others have to be accommodated as good as possible. Along with the choice of algorithm, operating system modules are chosen early on in the design process.

This contrasts actual industrial demands: designers want to select various types of functionality without consideration of which package they come from. They are reluctant to abandon trusted methods and to switch packages for the sake of an additional functional module only. Instead, there is a need to seamlessly integrate new functionality with a developed system, enabling designers to choose the best of various packages.

In this paper, we propose the use of a plug-in approach to add functionality to existing scheduling schemes and provide for easy replacement at the operating system level. In particular, we present an architecture to disentangle actual

[*]Björn Lindberg is now with Ericsson Radio Systems, Gothenburg, Sweden

real-time scheduling from dispatching and other kernel routines with a small API, suited for a variety of scheduling schemes as plug-ins. We detail two plug-ins for aperiodic task handling and how they can extend two target systems, table-driven and EDF scheduling using the presented approach.

A number of aperiodic task handling methods have been presented [10, 11, 9], but within their respective packages only. Instead of extending an existing scheduling package, we concentrate the functionality into a module, define the interface and discuss its application to off-line and on-line scheduling methods as examples. S.Ha.R.K [7] is an operating system where scheduling algorithms including aperiodic servers are created in a modular fashion. The interface between the system and the scheduler in S.Ha.R.K is more complex than the interface we propose in this paper.

The rest of the paper is organized as follows: in section 2, we describe our notion of plug-in and target system, its diversity is described in section 3, followed by a description of the aperiodic task handling functionality in section 4. In section 5 we show an example and section 6 concludes the paper.

2. System and Plug-In Architecture

A *plug-in* can be thought of as a hardware or software module that adds a specific feature or service to an existing system. The purpose of a plug-in is to add functionality without calling for redesign or extensive modifications. To accomplish this it must be clear what services the plug-in provides and an interface between the plug-in and the target system must be defined.

2.1. Target System Architecture and Interface

Before we go into the details of the plug-in, we define the target system model that the plug-in will interact with. The model is presented in figure 1 and it consists of three separate modules as parts of the system:

Execution Sequence Table This is the table where the tasks are kept sorted in a certain order, depending on the plug-in module's scheduling algorithm. The plug-in module has exclusive modification rights on this table. To manipulate the table, the plug-in module uses the two methods *insert(task, pos)* and *remove(task)*.

Dispatcher It is responsible for taking the first task in the execution sequence table and execute it. The dispatcher has access to view the contents of the whole execution sequence table, but it cannot modify it. The plug-in module also has exclusive control over the dispatcher and it is activated by the *dispatch()* call. When the dispatcher is activated, it will check if there is an executing task and either preempt the task, if it exists in the execution sequence table, or else abort it.

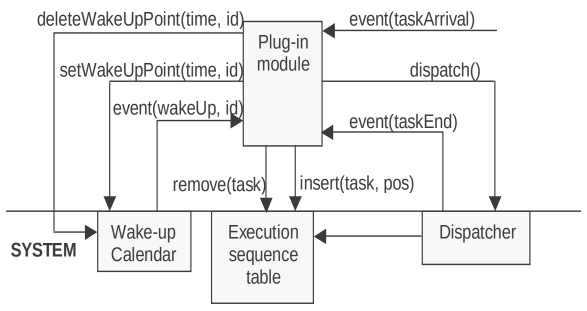

Figure 1. Plug-in and system architecture

Wake-up Calendar This calendar controls a set of watchdog timers, all tasks will get entries set in the calendar corresponding to their deadlines (to catch deadline misses). The calendar will also hold other time critical points, such as the critical slots from [3]. To set or remove these wake-up points the plug-in module uses the *setWakeUpPoint(time, id)* and the *deleteWakeUpPoint(time)* methods. All the wake-up points are associated with an id. The id's represents deadlines, critical slots, and so on.

2.2. Plug-In Interface

The plug-in module encapsulates a scheduling algorithm for scheduling of user level tasks (not system level tasks), such that the rest of the system becomes completely decoupled from the scheduling. This means that the plug-in module is the only part of the system that knows about scheduling, and it is also the only part that needs to be changed, if the scheduling algorithm is being changed.

Therefore the interface to the plug-in module is kept small and simple such that it is clear how to write a new plug-in. This makes it easier for designers to create the scheduling package they want. The plug-in interface is used by the system, specifically the wake-up calendar and dispatcher, to activate the plug-in module at certain events or times. Thus each plug-in module that is implemented, is responsible for reacting correctly to the events that activates it.

The details of the plug-in module interface and the events it must react to follow below:

event(taskArrival) This event activates the plug-in when a new user level task has been activated. The plug-in is responsible for executing the appropriate acceptance test to either accept or reject the new task. If the task is accepted, the plug-in must insert it at the correct position in the execution sequence table and activate the dispatcher.

event(wakeUp, id) This event is sent by the wake-up calendar and it activates the plug-in at a certain point of time earlier set by the plug-in itself. Here, the plug-in must check what the wake-up activation corresponds to, by looking at the id, and take the appropriate actions.

event(taskEnd) The dispatcher sends this event to the plug-in when a task has finished its execution. The dispatcher does not care if the task is periodic (and should be reactivated later) or aperiodic, it's the job of plug-in module to make the correct decision based on this. Here, the plug-in should remove the task from the execution sequence table and activate the dispatcher.

2.3. System and Plug-In Interaction

In figure 1, we can see the interface the system and the plug-in uses to interact with with each other. In this section we will describe in more detail how this interaction works for some of the events that can happen during system execution.

Task arrival when a new user task is activated, *event(taskArrival)* is called to activate the plug-in module. The module executes its acceptance test to either accept or reject the task. If the task is accepted, the plug-in calls *setWakeUpPoint(dl, id)* to set a watchdog on the deadline of the task. Then, the task is inserted into the execution sequence table, using *insert(task, pos)* to set it at the correct position according to the scheduling algorithm. Finally the plug-in activates the dispatcher, by calling *dispatch(),* and then it suspends itself. The dispatcher is activated, looks at the front of the execution sequence table, picks that task for execution and then it suspends.

Task finishing execution when a task has finished its execution in a timely manner, the dispatcher gets activated and activates the plug-in module by calling *event(taskEnd)*, then the dispatcher suspends. The plug-in removes the wake-up time for the task deadline with *removeWakeUpPoint(dl, id),* then it removes the task from the execution sequence table by *remove(task)*. The plug-in also calls *dispatch()* again to activate the dispatcher. The dispatcher looks at the front of the execution sequence table and picks that task for execution, then it suspends.

Task deadline miss if a task has not finished execution before its deadline, the wake-up calendar will be activated by a timer interrupt. It will then use *event(wake-up, id)* to activate the plug-in module. The plug-in module sees that the *id* indicates a deadline miss and removes the task from the execution sequence table, and, if necessary takes other actions to handle a deadline miss. Then the plug-in calls the dispatcher, using *dispatch()*, to activate it. The dispatcher checks if the executing task exist in the execution sequence table. When it discovers that the task has been removed by the plug-in it will abort the task. The dispatcher also checks for the first task in the execution sequence table, picks it for execution, and suspends itself.

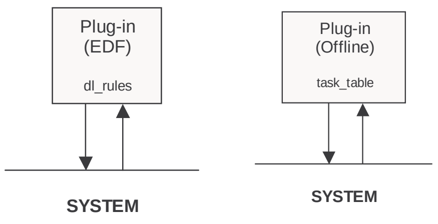

Figure 2. Example plug-ins

3. Target System Diversity and Plug-In Applicability

The plug-in module design makes it possible to hide the differences between scheduling algorithms behind a common interface. We will discuss how this architecture would be applied to the different scheduling paradigms that exist, and detail what the functions in the interface would do. Figure 2 shows the plug-ins.

3.1. Earliest deadline scheduled system

In an event-triggered system using the EDF scheduling algorithm, the tasks are characterized by start times, worst case execution time (WCET), and deadlines. The tasks can also be either periodic, and have the period as an additional attribute, or aperiodic. Before the start of the system, the plug-in sorts any existing tasks in the execution sequence table in EDF order. It also sets the wake-up events for the deadlines of the tasks in the wake-up calendar.

When the system is started, the plug-in activates the dispatcher and suspends itself. The dispatcher does it's job and suspends. If no new task arrives, the executing task will continue until it finishes its execution and then the dispatcher will activate the plug-in module again. The plug-in will see that it has been activated by a task-end event and remove that task from the execution sequence table. Then it activates the dispatcher again. This is how the plug-in and the system would interact if no new tasks would arrive or no deadline misses would occur.

If a new task arrives, the plug-in is activated and executes the acceptance test. If the task is accepted, it will be inserted into the execution sequence table at the correct position. The plug-in then activates the dispatcher and suspends, and the interaction continues as normal.

If a deadline miss occurs and activates the plug-in, the task will be removed from the execution sequence table. The plug-in then activates the dispatcher and the execution continues.

3.2. Off-line scheduled system

A target system using an off-line generated [8] schedule usually has more stringent task requirements, such as precedence constraints, than an on-line scheduled, event-triggered counterpart. In an off-line generated schedule, tasks have fixed starting and finishing times. In off-line scheduled systems there are only off-line scheduled task and no new task will dynamically arrive during the runtime of the system.

Before the execution of the system, the plug-in prepares the execution sequence table to correspond to the task table internally stored in the plug-in. The on-line execution of this plug-in will therefore be simpler with an EDF plug-in module. As with the EDF plug-in, wake-up points will also be set for the deadline of the tasks in the off-line schedule. The plug-in also sets wake-up points for every time slot, like the MARS system described in [5].

When the system is activated, the plug-in immediately sets a wake-up point at the next time-slot. If no task has a start time equal to the current time, it suspends. The plug-in will be activated at the start of the next time-slot and repeat what it did in the previous time-slot.

If there is a task with the start time equal to the current time, the plug-in activates the dispatcher, then it suspend. The dispatcher activates the execution of the next task and suspends.

The plug-in will be activated every slot, and it will also get events when tasks end or if tasks miss their deadline. If a task finishes execution in a timely manner, the dispatcher activates the plug-in, which removes the task from the execution sequence table and then checks if there is a task ready.

4. Plug-Ins for Aperiodic Task Handling

Below we present two plug-ins that handles aperiodic tasks. These plug-ins are meant to be “plugged into” a scheduling module that makes scheduling decisions based on earliest start times and deadlines. The plug-ins work independently of the scheduling module and can be seen as a layer on top of it.

At all times, the scheduling module schedules task that are ready to execute, that is, tasks that are present in the ready-queue. The plug-in deals with the aperiodic tasks and places them in the ready-queue of the scheduling module, which then processes the aperiodic tasks as it would any other tasks in the system.

The mechanism for the two plug-ins for aperiodic task handling is based on the slot shifting [2], taking advantage of resources not needed by non-aperiodic tasks and using them to schedule aperiodic tasks.

We have named the different plug-ins, plug-in A and plug-in B to distinguish between the two different algorithms. Plug-in A focuses on guarantees and handling of single aperiodic tasks with fixed demands, e.g., execution time, while plug-in B is geared towards large number of aperiodic tasks with changing requirements.

Aperiodic tasks have *unknown* arrival times. The earliest start time of an aperiodic task is equal to its arrival time. Aperiodic tasks are considered independent. We assume that task dependencies are resolved in the off-line phase.

Known WCET Aperiodic tasks with known worst case times and deadlines are termed *firm* aperiodic. If accepted, which is determined by a guarantee test, these tasks must be completed before their deadlines.

Unknown WCET Aperiodic tasks without deadlines and possibly without known maximum execution times are termed *soft* aperiodic. These are executed in a best effort fashion at lower priority than guaranteed tasks such that the timely execution of guaranteed tasks is not impaired.

4.1. Off-line Preparations - Slot Shifting

We propose to use the off-line transformation and on-line management of the slot shifting method [2]. Due to space limitations, we cannot give a full description here, but confine to salient features relevant to our new algorithms. More detailed descriptions can be found in [1], [2], [3]. It uses standard off-line schedulers, e.g., [8], [1] to create schedules which are then analyzed to define start-times and deadlines of tasks.

After off-line scheduling, and calculation of start-times and deadlines, the deadlines of tasks are sorted for each node. The schedule is divided into a set of *disjoint execution intervals* for each node. *Spare capacities (sc)* to represent the amount of available resources are defined for these intervals.

Each deadline calculated for a task defines the end of an interval I_i, $end(I_i)$. Several tasks with the same deadline constitute one interval. Note that these intervals differ from execution windows, i.e. start times and deadline: execution windows can overlap, intervals with *sc* are disjoint. The deadline of an interval is identical to that of the task. The start, however, is defined as the maximum of the end of the previous interval or the earliest start time of the task. The end of the previous interval may be later than the earliest start time. Thus it is possible that a task executes outside its interval, i.e., earlier than the interval start, but not before its earliest start-time.

The *sc* of an interval I_i are calculated as given in formula 1:

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

The length of I_i, minus the sum of the activities assigned to it, is the amount of idle time in that interval. These have to be decreased by the amount "lent" to subsequent intervals: Tasks may execute in intervals prior to the one they are assigned to. Then they "borrow" spare capacity from the "earlier" interval.

Obviously, the amount of unused resources in an interval cannot be less than zero, and for most computational purposes, e.g., summing available resources up to a deadline are they considered zero, as detailed in later sections. We use negative values in the spare capacity variables to increase runtime efficiency and flexibility. In order to reclaim resources of a task which executes less than planned, or not at all, we only need to update the affected intervals with increments and decrements, instead of a full recalculation. Which intervals to update is derived from the negative spare capacities. The reader is referred to [1] for details.

Thus, we can represent the information about amount and distribution of free resources in the system, plus online constraints of the off-line tasks with an array of four numbers per task. The runtime mechanisms of the first version of slot shifting added tasks by modifying this data structure, creating new intervals, which is not suitable for frequent changes as required by sporadic tasks. The method described in this paper only modifies spare capacity.

4.2. Online Activities

Runtime scheduling is performed locally for each node. If the spare capacities of the current interval $sc(I_c) > 0$, EDF is applied on the set of ready tasks. $sc(I_c) = 0$ indicates that a guaranteed task has to be executed or else a deadline violation in the task set will occur. It will execute immediately. Since the amount of time spent is known and represented in *sc*, guarantee algorithms include this information.

After each scheduling decision, the spare capacities of the affected intervals are updated. If, in the current interval I_c, an aperiodic task executes, or the CPU remains idle for one slot, current spare capacity in I_c is decreased. If an off-line task assigned to I_c executes, spare capacity does not change. If an off-line task T assigned to a later interval $I_j, j > c$ executes, the spare capacity of I_j is increased - T was supposed to execute there but does not, and that of I_c decreased. If I_j "borrowed" spare capacity, the "lending" interval(s) will be updated. This mechanism ensure that negative spare capacity turns zero or positive at runtime. Current spare capacity is reduced either by aperiodic tasks or idle execution and will eventually become 0, indicating a guaranteed task has to be executed. See [2] for more details.

4.2.1 Guarantee Algorithm A

Assume that an aperiodic task T_a is tested for guarantee. We identify three parts of the total spare capacities available:

- $sc(I_c)_t$, the remaining sc of the current interval
- $\sum sc(I_i), \ c < i \leq l, \ end(I_l) \leq dl(T_A) \wedge end(I_{l+1}) > dl(T_A), \ sc(I_i) > 0$, the positive spare capacities of all *full* intervals between t and $dl(T_A)$
- $min(sc(I_{l+1}), dl(T_A) - start(I_{l+1}))$, the spare capacity of the last interval, or the execution need of T_A before its deadline in this interval, whichever is smaller

If the sum of all three is larger than $wcet(T_A)$, T_A can be accommodated, and therefore guaranteed. Upon guarantee of a task, the spare capacities are updated to reflect the decrease in available resources. Taking into account that the resources for T_A are not available for other tasks. This guarantee algorithm is $O(N)$, N being the number of intervals.

4.2.2 Guarantee Algorithm B

This plug-in uses a newer version of slot shifting as guarantee test and the basic idea behind it is based on the standard EDF guarantee. EDF is based on having full availability of the CPU, so we have to consider interference from the non-aperiodic tasks in S and pertain their feasibility.

Assume that at time t_1 we have a set of guaranteed aperiodic tasks G_{t_1} and a set of non-aperiodic tasks S. At time t_2 where $t_1 < t_2$, a new aperiodic A arrives to the plug-in module. Meanwhile, a number of tasks of G_{t_1} may have executed; the remaining task set at t_2 is denoted G_{t_2}. We test if $A \cup G_{t_2}$ can be accepted, considering tasks in S. If so, we add A to the set of guaranteed aperiodic tasks, G.

The finishing time of a firm aperiodic task A_i, with an execution demand of $c(A_i)$, is calculated with respect to the finishing time of the previous task, A_{i-1}. Without any off-line tasks, it is calculated the same way as in the EDF algorithm:

$$ft(A_i) = ft(A - i - 1) + c(A_i) \quad (2)$$

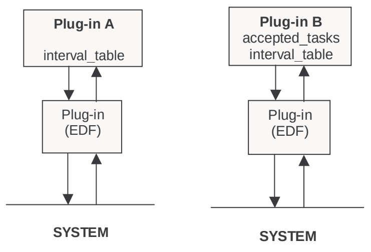

Figure 3. Plug-in A and Plug-in B

Since we guarantee firm aperiodic tasks together with tasks in S, we extend the formula above with a new term that reflects the amount of resources reserved for these tasks:

$$ft(A_i)=c(A_i)+\begin{cases} t+\mathrm{R}[t,ft(A_1)] & ,i=1 \\ ft(A_{i-1})+\mathrm{R}[ft(A_{i-1}),ft(A_i)], & i>1 \end{cases} \quad (3)$$

where $R[t1,t2]$ stands for the amount of resources (in slots) reserved for the execution of the tasks in S between time t_1 and time t_2 . We can access $R[t1,t2]$ via spare capacities and intervals at runtime:

$$R[t_1,t_2]=[t_2-t_1]-\sum_{I_c\in(t_1,t_2)} max(sc(I_c),0) \quad (4)$$

As $ft(A_i)$ appears on both sides of the equation, a simple solution is not possible. But in [4] an algorithm, with a complexity of $O(N)$, for computing the finishing times of hard aperiodic tasks is presented.

In this plug-in module no explicit reservation of resources is done, which would require changes in the intervals and spare capacities, as done in the plug-in A module. Rather, resources are guaranteed by accepting the task only if it can be accepted together with the previous tasks in G and S. This enables the efficient use of rejection strategies, and simplifies the handling of the intervals and *sc*.

4.3. Guarantee Plug-Ins

When a plug-in is activated, it updates the intervals in conformity with the last task execution and checks if there are any pending aperiodic tasks. If so, it processes them and puts one or more of them into the ready-queue of the scheduler. Figure 3 show the two plug-ins and the data structures they contain.

4.3.1 Plug-In A

The plug-in keeps a table consisting of the intervals and their attributes (start, end, sc, and so on) that was created in the off-line phase. It must also keep track of which task executed last, when it started its latest execution, and how much time it consumed, to be able to update the intervals table. Using this information, the plug-in updates interval spare capacities and possibly also wake-up points.

4.3.2 Plug-in B

Plug-in B also needs information about the last task execution to be able to update spare capacities and wake-up points in the intervals table it keeps locally. It focuses on handling large numbers of aperiodic tasks with changing requirements, therefore accepting tasks is done with explicit guarantees via modifying intervals and spare capacities. Rather, guarantees are including implicitly, by keeping a list of the so far accepted task. Should a task finish early, it is removed from the list and the resources reserved for it are freed without further provisions. It is well suited for efficient overload handling, since task removals do not require changes in intervals and spare capacities as in plug-in A.

After each scheduling decision, the spare capacities of the affected intervals are updated as for plug-in A.

5. Example

In this section we will use an example to illustrate how the two plug-in modules we defined earlier, plug-in A and plug-in B, work and interact with the rest of the system. We assume that there are three periodic tasks scheduled by the EDF algorithm, and the task-set is the following: $A=(1,4)$, $B=(1,6)$, $C=(2,12)$, where (C,T) represents WCET and period. Deadline is assumed to be equal to the end of the period ($D=T$). The tasks have harmonic periods to make the example simple. Firm aperiodic tasks have the format: $Ta_f=(C,D)$, and soft aperiodic tasks have the following format: $Ta_s=(C)$.

Off-line In the off-line phase the plug-ins create a table that contains all the interval start and end points, the length of the interval, the *sc* and total execution time in an interval, and lastly the wake-up (wu) point of the interval. This table is stored within the plug-in and it will be updated during runtime to reflect the correct state. Both plug-ins create identical tables as shown in table 1. The table is created with a length equal to the least common multiple (LCM) of the periods of the tasks. This table will be restored and repeated when time t is equal to a multiple of the LCM.

The execution sequence table (ES-table) contains the following periodic tasks from the start: ES-table$=\{A_0,B_0,C_0\}$.

On-line The on-line behavior of the two models differs so we will show step by step how each of them behave, and what happens with the interval table at different times. Below we will see the actions taken during each step by the system and the plug-ins.

Interval	*start(I)*	*end(I)*	$\mid I \mid$	*sc(I)*	*wu(I)*
I_0	0	4	4	3	3
I_1	4	6	2	1	5
I_2	6	8	2	1	7
I_3	8	12	4	0	8

Table 1. The original interval table.

Interval	*start(I)*	*end(I)*	$\mid I \mid$	*sc(I)*	*wu(I)*
I_0	0	4	4	3	3
I_{1a}	4	5	1	0	4
I_{1b}	5	6	1	0	5
I_2	6	8	2	1	7
I_3	8	12	4	0	8

Table 2. Updated interval table for plug-in A.

Time	System actions	Plug-in actions

This shows how the actions by the different parts will be represented. At each point time we can see the system's dispatcher, wake-up calendar actions, and the plug-in's actions.

$t = 0$	dispatch A_0	*setWakeUpPoint(3), dispatch()*

No new aperiodic tasks have arrived so the plug-in sets a wake-up point and suspends.

$t = 1$	dispatch Ta_f	*remove(A_0),* Guarantee-test, *deleteWakeUpPoint(3,critical-slot), setWakeUpPoint(4), insert(Ta_f,dl-pos), dispatch()*

ES-table= $\{B_0, C_0\}$ and a firm aperiodic task has arrived, $Ta_f = (1, 4)$.

Plug-in A The absolute deadline of Ta_f is 5, so $I_c = I_0$ and $I_f = I_1$ and the available sc in this interval is 4 ($sc(I_c) + sc(I_f)$), which is larger than Ta_f execution requirement, so Ta_f will be guaranteed. Since Ta_f 's deadline, 5, is not equal to $end(I_f)$, I_1 will have to be split. The *sc* is also updated after the split and the interval table for plug-in A is shown in table 2.

Plug-in B In this plug-in the set of guaranteed aperiodic tasks (G) is empty. The plug-in tests if Ta_f can be accepted together with the periodic tasks. This is done by calculating the finishing time of Ta_f, which is 2 in this case (according to formula 3). No interval split will occur in this plug-in, nor any change to the *sc* of the intervals table because an aperiodic task was accepted.

Both plug-ins will set an updated wake-up point. The wake-up point has been changed because task A_0 has executed one slot, and then suspend.

$t = 2$	*event(taskEnd),* dispatch B_0	*remove(Ta_f),* Internal-work, *dispatch()*

ES-table= $\{B_0, C_0\}$. No new aperiodic tasks has arrived, Ta_f has finished. The plug-ins will be activated by this task-end event, *plug-in A* will modify the wake up point of the interval Ta_f belonged to in the intervals table, $wu(I_1a = 5)$, and then suspend again. *Plug-in B* takes no action and suspends.

$t = 3$	*event(taskEnd),* dispatch C_0	*remove(B_0),* Internal-work, *dispatch()*

ES-table= $\{C_0\}$. No new aperiodic tasks have arrived. C_0 will execute. B_0 has finished, the wake up point is not modified because B_0 belongs to a later interval (but the wu in that interval is modified, so $wu(I_1) = 6$).

$t = 4$	*event(wakeUp),* dispatch Ta_s	*insert(Ta_s,first-pos), insert(A_1,pos), setWakeUpPoint(5), setWakeUpPoint(6), dispatch()*

ES-table= $\{A_1, C_0\}$. Next instance of task A is ready. C_0 has finished executing and it belongs to a later interval, so the wu of that interval is modified ($wu(I_3) = 9$).

A soft aperiodic task $Ta_s = (4)$ has arrived. Both plug-ins will behave in the same manner: since $sc(I_c) > 0$, task Ta_s will be inserted first in the ready-queue. *Plug-in B* will set the next wake up point and suspend. *Plug-in A* will set the wake up to 5 even though the original $wu(I_c) = 4$, this has changed because Ta_f executed in an earlier interval and thus the $sc(I_c)$ increased to 1.

$t = 5$	*event(wakeUp),* dispatch Ta_s	*setWakeUpPoint(6), dispatch()*

ES-table= $\{A_1, C_0\}$. Plug-in A is activated by the wake-up point event. Normally this means that the execution of the soft task must be stopped in favor of a periodic task. But in this case we have only an interval change, and the $sc(I_c) > 0$, so the soft task can continue to execute ($sc(I_c) > 0$ because B_0 executed in an earlier interval). *Plug-in A* resets the wake up point and suspends itself. *Plug-in B* is not activated.

$t = 6$	*event(wakeUp),* dispatch Ta_s	*insert(B_1,EDF-pos), setWakeUpPoint(7), dispatch()*

ES-table= $\{A_1, B_1, C_0\}$. The second instance of task B is activated. Both plug-ins are activated by wake up points, this means that the execution of the soft task must be stopped in favor of a periodic task. Once again, there is only an interval change and a new wake up point can be set, and since the $sc(I_c) > 0$, Ta_s can continue to execute. Both the plug-ins suspend.

$t = 7$	*event(wakeUp),* dispatch A_1	*remove(Ta_s), setWakeUpPoint(8), dispatch()*

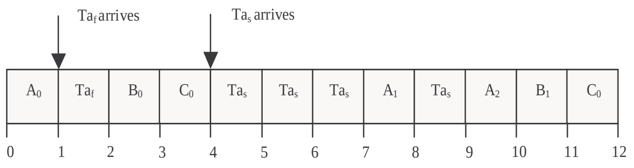

Table 3. Example execution trace

ES-table= $\{A_1, B_1, C_0\}$. The plug-ins are activated due to the wake up point. Ta_s must be interrupted so A_1 won't miss it's deadline. The plug-ins set the next wake up point and suspend.

$t = 8$	*event(wakeUp)*, *event(taskEnd)*, dispatch Ta_s	*remove(A_1), insert(A_2,pos), insert(Ta_s,first-pos), setWakeUpPoint(9), dispatch()*

ES-table= $\{A_2, B_1, C_0\}$. The next instance of task A is activated. Since the $sc(I_c) > 0$, Ta_s will be put first in the ready queue and executed. The plug-ins set the next wake up point and suspend.

$t = 9$	*event(wakeUp)*, *event(taskEnd)*, dispatch A_2	*remove(Ta_s), setWakeUpPoint(10), dispatch()*

ES-table= $\{A_2, B_1, C_0\}$. Ta_s has finished executing, the plug-ins set the next wake up point and suspend.

$t = 10$	*event(wakeUp)*, *event(taskEnd)*, dispatch B_1	*remove(A_2), setWakeUpPoint(11), dispatch()*

ES-table= $\{B_1, C_0\}$. A_2 has finished it's execution, B_1 is executed. The plug-ins set the next wake-up point and suspend.

$t = 11$	*event(wakeUp)*, *event(taskEnd)*, dispatch C_0	*remove(B_1), setWakeUpPoint(12), dispatch()*

ES-table= $\{C_0\}$. B_1 has finished executing, the plug-ins set the next wake up point and suspend.

After this, because $t =$ total length of the interval tables, the plug-ins recreate the original intervals table by restoring the *sc* and *wu* of the intervals. If an aperiodic task arrives and has a deadline longer than the end of the interval table, the table will be extended by repeatedly adding the original table to the end of the extended table, until it is longer than the deadline. All the interval information (start, end, sc, and so on) of the extended table is adjusted to represent a larger table, and thus later time points.

6. Conclusion

In this paper we addressed the need for adding functionality to systems, in particular scheduling algorithms, without need for abandoning trusted methods or major revisions.

We proposed a plug-in approach for aperiodic task handling, presented two different plug-in modules, and showed their applicability to two scheduling schemes, EDF, and off-line scheduling. Our method concentrates the aperiodic task functionality into a software module with a defined interface.

We presented an architecture to disentangle actual real-time scheduling from dispatching and other kernel routines with a small API, suited for a variety of scheduling schemes as plug-ins. As the functionality of the plug-in is independent of the basic scheduling scheme and the interface is very small, we can insert and apply the aperiodic-plug-ins to both off-line and on-line scheduling methods.

Further research will go into extending the applicability to a wider range of systems and algorithms.

References

[1] G. Fohler. *Flexibility in Statically Scheduled Real-Time Systems*. PhD thesis, Wien, Österreich, April 1994.

[2] G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proceedings of the 16th Real-Time Systems Symposium*, Pisa, Italy, Dec. 1995.

[3] D. Isovic and G. Fohler. Handling sporadic tasks in statically scheduled distributed real-time systems. In *Proceedings of the 10th Euromicro Real-Time Systems Conference*, June 1999.

[4] D. Isovic and G. Fohler. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, Orlando, Florida, USA, Nov. 2000.

[5] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The MARS Approach. 9(1):25–40, Feb. 1989.

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

[7] M. G. P. Gai, L. Abeni and G. Buttazzo. A new kernel approach for modular real-time systems development. In *Proceedings of the 13th Euromicro Real-Time Systems Conference*, June 2001.

[8] K. Ramamritham. Allocation and scheduling of complex periodic tasks. In *International Conference on Distributed Computing Systems*, pages 108–115, 1990.

[9] M. Spuri and G. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *The Journal of Real-Time Systems*, pages 179–210, Mar. 1996.

[10] S. R. Thuel and J. Lehoczky. On-line scheduling of hard deadline aperiodic tasks in fixed-priority systems. pages 160–171, Dec. 1993.

[11] S. R. Thuel and J. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *Proceedings of the Real-Time Symposium*, pages 22–33, San Juan, Puerto Rico, Dec. 1994.

[12] V. Yodaiken. Rough notes on priority inheritance. Technical report, New Mexico Institut of Mining, 1998.