Efficient Scheduling of Sporadic, Aperiodic, and Periodic Tasks with Complex Constraints

Damir Isović and Gerhard Fohler
Department of Computer Engineering
Mälardalen University, Sweden
{dic,gfr}@mdh.se

Abstract

Many industrial applications with real-time demands are composed of mixed sets of tasks with a variety of requirements. These can be in the form of standard timing constraints, such as period and deadline, or complex, e.g., to express application specific or non temporal constraints, reliability, performance, etc. Arrival patterns determine whether tasks will be treated as periodic, sporadic, or aperiodic. As many algorithms focus on specific sets of task types and constraints only, system design has to focus on those supported by a particular algorithm, at the expense of the rest.

In this paper, we present an algorithm to deal with a combination of mixed sets of tasks and constraints: periodic tasks with complex and simple constraints, soft and firm aperiodic, and sporadic tasks. Instead of providing an algorithm tailored for a specific set of constraints, we propose an EDF based runtime algorithm, and the use of an offline scheduler for complexity reduction to transform complex constraints into the EDF model. At runtime, an extension to EDF, two level EDF, ensures feasible execution of tasks with complex constraints in the presence of additional tasks or overloads. We present an algorithm for handling offline guaranteed sporadic tasks, with minimum interarrival times, in this context which keeps track of arrivals of instances of sporadic tasks to reduce pessimism about future sporadic arrivals and improve response times and acceptance of firm aperiodic tasks.

A simulation study underlines the effectiveness of the proposed approach.

1 Introduction

Many industrial applications with real-time demands are composed of tasks of various types and constraints. Arrival patterns and importance, for example, determine whether tasks are periodic, aperiodic, sporadic, and soft, firm, or hard. The controlling real-time system has to provide for a *combined* set of such task types. The same holds for the various constraints of tasks. In addition to basic temporal constraints, such as periods, start-times, deadlines, and synchronization demands, e.g., precedence, jitter, or mutual exclusion, a system has to fulfill complex application demands which cannot be expressed as generally: Control applications may require constraints on individual instances [27], rather than periods, reliability demands can enforce allocation and separation patterns, or engineering practice may require relations between system activities, all of which cannot be expressed directly with basic constraints.

The choice of scheduling algorithm or paradigm determines the set of types and constraints on tasks during system design. Earliest deadline first or fixed priority scheduling, for example, are chosen for simple dispatching and flexibility. Adding constraints, however, increases scheduling overhead [29] or requires new, specific schedulability tests which may have to be developed yet. Offline 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. 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 well as possible.

In this paper we present an algorithm to deal with mixed sets of tasks and constraints: periodic tasks with complex and simple constraints, soft and firm aperiodic, and sporadic tasks. Instead of providing an algorithm tailored for a specific set of constraints only, we propose an EDF based runtime algorithm, and the use of an offline scheduler for complexity reduction to transform complex constraints into the simple EDF model. So, at runtime, all tasks are constrained by start-time and deadline only, which serves as an "interface" between tasks of mixed constraints. An extension of EDF, *two level EDF* ensures the feasible execution

of those tasks, whose complex constraints have been transformed offline, even in the presence of additional runtime activities and overloads. It serves as a basis for the sporadic task handling presented.

The offline transformation determines resource usage and distribution as well, which we use to handle sporadic tasks. An offline test determines and allocates resources for sporadic tasks such that worst case arrivals can be accommodated at any time. At runtime, however, when a sporadic task arrives, we do not need to account for its arrival at least for its minimum inter-arrival time and reuse its allocated resources for aperiodic tasks. Our algorithm keeps track of sporadic task arrivals to update the "current worst case" and applies it for an online $O(N)$ acceptance test for aperiodic tasks.

Our methods also provides for the integration of offline and online scheduling: A complete offline schedule can be constructed, transformed into EDF tasks, and scheduled at runtime together with other EDF tasks. Thus, our method combines handling of complex constraints, efficient and flexible runtime scheduling, as well as offline and online scheduling.

A variety of algorithms have been presented to handle periodic and aperiodic tasks, e.g., [22], [21], [24]. Most concentrate on particular types of constraints. An on-line algorithm for scheduling sporadic tasks with shared resources in hard real-time systems has been presented in [14]. Scheduling of sporadic requests with periodic tasks on an *earliest-deadline-first (EDF)* basis [17] has been presented in [25]. The slot shifting algorithm to combine offline and online scheduling was presented in [8]. It focuses on inserting aperiodic tasks into offline schedules by modifying the runtime representation of available resources. While appropriate for including sequences of aperiodic tasks, the overhead for sporadic task handling becomes too high. An offline test for sporadic tasks was given in [11]. It does not, however, provide for firm aperiodic tasks, only soft ones. The method presented here is based on the offline transformation of slot shifting but provides a new runtime algorithm, in particular for efficient sporadic task handling and resource reclaiming at runtime. It handles firm and soft aperiodic tasks as well.

The use of information about amount and distribution of unused resources for non periodic activities is similar to the basic idea of slack stealing [24], [5] which applies to fixed priority scheduling. Our method applies this basic idea in the context of offline and EDF scheduling. It requires a small runtime data structure, simple runtime mechanisms, going through a list with increments and decrements, provides $O(N)$ acceptance tests and facilitates changes in the set of tasks, for example to handle overloads. Furthermore, our method provides for handling of slack of non periodic tasks, as well, e.g., instances of tasks can be separated by intervals other than periods.

The rest of this paper is organized as follows: The task model is described in section 2. The subsequent sections describe the different types of tasks: section 3 describes handling of periodic tasks, the offline complexity reduction of constraints, and a description of the extended EDF runtime scheduling method. Soft and firm aperiodic task handling is discussed in section 4. The algorithm for handling sporadic task together with the other task types is presented in section 5. Simulation results in section 6 illustrate the effectiveness of the algorithm. Finally, section 7 concludes the paper.

2 Task and System Assumptions

In this paper, we distinguish between *simple* constraints, i.e., period, start-time, and deadline, for the earliest deadline first scheduling model, and *complex* constraints.

2.1 Complex constraints

We refer to such relations or attributes of tasks as *complex constraints*, which cannot be expressed directly in the earliest deadline first scheduling model using period, start-time, and deadline. Offline transformations are needed to schedule these at runtime with EDF. For some specific constraints such transformations have been presented, e.g., [4], [10]. Our method uses a general technique, capable of incorporating various constraints and their combinations. In the following, we list examples to illustrate and motivate the general approach.

Synchronization: Execution sequences, such as sampling - computing - actuating require a *precedence* order of task execution. An algorithm for the transformation of precedence constraints on single processors to suit the EDF model has been presented in [4]. Many industrial applications, however, demand the allocation of tasks, in particular for sensing and actuating to different processors, necessitating a distributed system with internode communication. The transformation of precedence constraints with an end-to-end deadline in this case requires subtask deadline assignment to create execution windows on the individual nodes so that precedence is fulfilled, e.g., [6]. The analysis presented in [26] focuses on schedulability analysis for pairs of tasks communicating via a network rather than the decomposition of the precedence graph.

Jitter: The execution start or end of certain tasks, e.g., sampling or actuating in control systems, is constrained by maximum variations. Strictly periodic execution can solve some instances of this problem, but over-constrains

the specification. Algorithms are computationally expensive [2].

Non periodic execution: The commonly used model is periodic, i.e., instances of tasks are released at constant, periodic intervals of time. Non periodic constraints, such as some forms of jitter, e.g., for feedback loop delay in control systems [27], which require instances of tasks to be separated by *non constant* length intervals cannot be handled in this model, or have to be fitted into the periodic model at the cost of over constrained specification. Similar reasoning applies to constraints over more than one instance of a task, e.g., for iterations, data history or ages. A constraint can be of the type "separate the execution of instance i and $i + 4$ by no more than max and no less than min".

Non temporal constraints: Demands for reliability, performance, or other system parameters impose demands on tasks from a system perspective, e.g., to not allocate two tasks to the same node, or to have minimum separation times, etc.

Application specific constraints - engineering practice: Applications may have demands specific to their nature. Duplicated messages on a bus in an automotive environment, for example, may need to follow a certain pattern due to interferences such as EMI. Wiring can have length limitations, imposing allocation of certain tasks to nodes according to their geographical positions. An engineer may want to improve schedules, creating constraints reflecting his practical experience.

2.2 Task types

We assume all tasks in the system to be fully preemptive and to communicate with the rest of the system via data read at the beginning and written at the end of their executions.

Periodic tasks execute their invocations within regular time intervals. A periodic task T_P is characterized by its worst case execution time ($wcet$) [19], period (p) and relative deadline (dl).
The k^{th} invocation of T_P is denoted T_P^k and is characterized by its earliest start time (est) and relative deadline (dl).

We refer to a periodic task with complex constraints which have been transformed offline as an *offline task*.

Aperiodic tasks are invoked only once. Their arrival times are unknown at design time. A *firm* aperiodic task T_A has the following set of parameters: the arrival time (ar), worst case execution time and relative deadline. *Soft* aperiodic tasks have no deadline constraints.

Sporadic tasks can arrive at the system at arbitrary points in time, but with defined minimum inter-arrival times between two consecutive invocations. A sporadic task T_S is characterized by its relative deadline, minimum inter-arrival time (λ) and worst case execution time.
These attributes are known before the run-time of the system. Additional information available on-line, is its arrival time and its absolute deadline.

2.3 System assumptions

Distributed system: We consider a *distributed* system, i.e., one that consists of several processing and communication nodes [23]. While we allow for distributed system and distribution in the complex constraints, we handle those issues in the offline phase, i.e., at runtime, no task migration takes place.

Time model: We assume a discrete time model [16]. Time ticks are counted globally, by a synchronized clock with granularity of *slotlength*, and assigned numbers from 0 to ∞. The time between the start and the end of a slot *i* is defined by the interval $[slotlength * i, slotlength * (i+1)]$. Slots have uniform length and start and end at the same time for all nodes in the system. Task periods and deadlines must be multiples of the slot length.

2.4 Task handling - overview

The following table gives an overview of when types of tasks are handled by our method.

		soft aper.	firm aper.	spo-radic	simp. per.	comp. per.
offline	sched.				x	x
	test			x		
online	sched.	x	x	x	x	x
	test	x	x			

3 Periodic Tasks - Offline Complexity Reduction

In this section we start describing the handling of various types of tasks, with periodic tasks with complex constraints. We will present a method for complexity reduction and online scheduling to ensure these transformed constraints.

3.1 Offline complexity reduction

Finding optimal solutions to most sets of complex constraints is an NP hard problem [9]. Consequently algorithms will be heuristic and produce suboptimal solutions

only. Performing the complexity reduction offline, however, allows for elaborate methods, improvement of results and modifications in the non-successfull case. The transformation method should be flexible to include new types of constraints, to accommodate application specific demands and engineering requirements. A number of general methods for the specification and satisfaction of constraints can be applied for real-time tasks, e.g., [13] or [28]. Runtime scheduling has to ensure that tasks execute according to their constraints, even in the presence of additional tasks or overload situations.

We propose to use the offline transformation and online guarantee of complex constraints of the slot shifting method [8][1]. 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 [7], [8], [12]. It uses standard offline schedulers, e.g., [20], [7] to create schedules which are then analyzed to define start-times and deadlines of tasks.

First, the offline scheduler creates scheduling tables for the selected periodic tasks with complex constraints. It allocates tasks to nodes and resolves complex constraints by constructing sequences of task executions. The resulting offline schedule is a single feasible, likely suboptimal solution. These sequences consist of subsets of the original task set separated by allocation. Each task in a sequence is limited by either sending or receiving of internode messages, predecessor or successor within the sequence, or limits set by the offline scheduler. Start times and deadline are set directly to the times of internode messages or offline scheduler limits, or calculated recursively for tasks constrained only within sequences. A more detailed description can be found in [8]. The final result is a set of independent tasks on single nodes, with start-times and deadlines.

The offline scheduling algorithm we use [7] is based on heuristic search to handle complexity reduction, and provide for straightforward inclusion of additional constraints by providing an additional feasibility test. It works with precedence constraints as basic model, handles jitter constraints, and performs allocation and subtask deadline assignment. In addition to constraint transformation, the use of an offline scheduler provides for integration of offline and online scheduling as well.

3.2 Runtime guarantee of complex constraints

In the previous steps we created tasks with start-time and deadline constraints, which can be scheduled by EDF at runtime. The resulting feasible schedules represent the original complex constraints. Additional runtime tasks, however, can create overload situations, resulting in violations of the complex constraints. Thus, a mechanism is needed which ensures the feasible execution of these tasks, even in overload situations.

We propose an extension to EDF, *two level EDF* for this purpose. The basic idea is to schedule tasks according to EDF - "normal level", but give priority -"priority level" to an offline task when it needs to start at latest, similar to the basic idea of slack stealing [24] [5] for fixed priority scheduling. Thus, the CPU is not completely available for runtime tasks, but reduced by the amount allocated for offline tasks. So we need to know amount and location of resources available after offline tasks are guaranteed. Runtime efficiency demands simple runtime data structure and runtime maintenance.

Offline preparations After offline 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* 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. 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 spare capacities, as defined here, 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, or earlier (empty interval). Thus it is possible that a task executes outside its interval, i.e., earlier than the interval start, but not before its earliest starttime.

The spare capacities 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.

[1] We do not, however, use its runtime scheduling and handling of non periodic tasks.

Which intervals to update is derived from the negative spare capacities. The reader is referred to [7] for details.

Thus, we can represent the information about amount and distribution of free resources in the system, plus online constraints of the offline 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.

Online execution 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 - "normal level". $sc(I_c) = 0$ indicates that a guaranteed task has to be executed or else a deadline violation in the task set will occur. It will execute immediately - "priority level". Since the amount of time spent at priority level is known and represented in spare capacity, 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 offline task assigned to I_c executes spare capacity does not change. If an offline 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 ensures 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 [8] for more details.

4 Aperiodic Tasks

A first verion of slot shifting presented an algorithm to guarantee aperiodic tasks by inserting them into an offline schedule. Once guaranteed, the resources allocated for the aperiodic were removed by creating a new interval and adjusting spare capacity. While efficient for guaranteeing sequences of aperiodic tasks without removal, the runtime overhead for handling sporadic tasks efficiently is too high. Further, changes in the set of guaranteed tasks require costly deletion of intervals, recalculation of spare capacities, and new guarantees. Thus, flexible schemes for rejections, removal of guaranteed tasks, and overload handling induce prohibitively high overhead.

The new method presented here separates acceptance and guarantee. It eliminates the online modificiation of intervals and spare capacities and thus allows rejection strategies over the entire aperiodic task set.

4.1 Acceptance test

The basic idea behind our method is based on standard earliest deadline first guarantee, but sets it to work on top of the offline tasks. EDF is based on having full availability of the CPU, so we have to consider interference from offline scheduled tasks and pertain their feasibility.

Assume, at time t_1, we have a set of guaranteed aperiodic tasks $\mathcal{G}_{t_1}$ and an offline schedule represented by offline tasks, intervals, and spare capacities. At time $t_2, t_1 < t_2$, a new aperiodic A arrives. Meanwhile, a number of tasks of $\mathcal{G}_{t_1}$ may have executed; the remaining task set at t_2 is denoted $\mathcal{G}_{t_2}$. We test if $A \cup \mathcal{G}_{t_2}$ can be accepted, considering offline tasks. If so, we add A to the set of guaranteed aperiodics. No explicit reservation of resources is done, which would require changes in the intervals and spare capacities. Rather, resources are guaranteed by accepting the task only if it can be accepted *together* with the previous guaranteed and offline scheduled ones. This enables the efficient use of rejection strategies.

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 offline tasks, it is calculated the same way as in the EDF algorithm:

$$ft(A_i) = ft(A_{i-1}) + c(A_i)$$

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

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

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

$$R[t_1, t_2] = (t_2 - t_1) - \sum_{I_c \in (t1, t2)} max(sc[I_c], 0)$$

As $ft(A_i)$ appears on both sides of the equation, a simple solution is not possible. Rather, we present an algorithm for computation of finishing times of hard aperiodic tasks with complexity of $O(N)$.

4.2 Algorithm

Let A_i be a firm aperiodic task we want to guarantee. Let $\mathcal{G}$ denote the set of previously guaranteed but not yet completed firm aperiodic tasks, such as each task in $\mathcal{G}$ has a deadline later than or equal to $dl(A)$:

$$G = \{A_j \mid i < j < n, dl(G_j) \geq dl(A_i)\}$$

Here is the pseudo code for the acceptance test and algorithm for finishing time calculation:

$ft = getFinishingTime(max(ft(A_{i-1}), t), c(A_i));$
/* check if accepting A_i causes any of the previously guaranteed firm aperiodic tasks to miss its deadline */
$if(ft \leq dl(A_i))\{$
    $for(j = i+1; j < n; j++)\{$
        $ft = getFinishingTime(ft, remc(G_j));$
        $if(ft > dl(G_j)) \Rightarrow$ not feasible!
    $\}$
    $insert(A, \mathcal{G});$
$\}$
$else$ reject A

$getFinishingTime(ft_p, remc)\{$
    /* determine ft by "filling up"
    free slots until the c is exhausted. */
    $sc_r = start(I_c) + sc(I_c) - ft_p;$
    $while\ (remc > sc_r)\{$
        $if(sc_r\ (I_c) > 0)$
            $remc = remc - sc_r;$
        $c++;$
        $ft_p = start(I_c)$
        $sc_r = sc(I_c);$
    $\}$
    return $(ft_p + remc)$;
$\}$

$remc$ = remaining execution time
ft_p = the finnishing time of predecessor task

The complexity of algorithm is $O(N)$, because we go through all tasks only once, and calculate their finishing times on the way. More detailed description of the algorithm can be found in [12].

5 Sporadic Tasks

In the previous section we described how firm aperiodic tasks are guaranteed online assuming no sporadic tasks in the system. Now we will see how sporadic tasks can be included in the aperiodic guarantee.

We will discuss ways to handle sporadic tasks with periodic tasks with complex constraints. We present a new algorithm which keeps track of sporadic task arrivals and reduces pessimism about possible future arrivals to improve guarantees and response times of aperiodic tasks.

5.1 Handling sporadic tasks

Pseudo-periodic tasks – Sporadic tasks can be transformed offline into pseudo-periodic tasks [18] which can be scheduled simply at runtime. The overhead induced by the method, however, can be very high: in extreme cases, a task handling an event which is rare, but has a tight deadline may require reservation of all resources.

Offline test – In an earlier paper [11], we have presented an offline test for sporadic tasks on offline tasks. It ensured that the spare capacity available was sufficient for the worst case arrival of sporadic tasks without reflecting it in the spare capacity. Consequently, firm aperiodic tasks cannot be handled at runtime.

Offline test and online aperiodic guarantees – A better algorithm will perform the offline test and decrease the needed resources from spare capacity. The resulting pessimism can be reduced by reclaiming a slot upon non arrival of a sporadic task. Aperiodic guarantee will be possible, but have to consider worst case arrival patterns of the sporadic tasks at any time.

Offline test and online aperiodic guarantees and reduced pessimism – The algorithm presented here performs the offline test, but does not change intervals and spare capacity for runtime efficiency. At runtime, its keeps track of sporadic arrivals to reduce pessimism, by removing sporadic tasks from the worst case arrival which are known to not arrive up to a certain point. An aperiodic task algorithm utilizes this knowledge for short response times.

5.2 Interference window

We do not know when a sporadic task $S_i \in \mathcal{S}$, will invoke its instances, but once an instance of S_i arrives, we do know the minimum time until the arrival of the next instance — the minimum inter-arrival time of S_i. We also know the worst case execution time of each sporadic task in $\mathcal{S}$. We use this information for the acceptance test of firm aperiodic tasks.

Assume a sporadic task S_i invokes an instance at time t (see figure 1). Let S_i^k denote current invocation of S_i, and S_i^{k+1} the successive one. At time t we know that S_i^{k+1} will arrive no sooner than $t + \lambda$, where λ is the minimum inter-arrival time of S_i. So, when S_i^k has finished its execution, S_i will not influence any of the firm aperiodic tasks at least until S_i^{k+1} arrives. This means that, when calculating the amount of resources available for a firm aperiodic task with an execution that intersects with S_i's execution window, we do not need to take into account the interference from S_i

at least between the finishing time of its current invocation, S_i^k, and the start time on the next invocation, S_i^{k+1}, as depicted in figure 1.

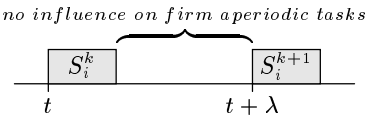


Figure 1. A sporadic task.

Let $EW(A_i)$ denote the execution window of A_i, i.e., the interval between A_i's arrival and its deadline:

$$EW(A_i) = [ar(A_i), dl(A_i)],\ |EW| = dl(A_i) - ar(A_i)$$

Now we will see how the execution of a previously guaranteed sporadic task $S_i \in \mathcal{S}$ can influence A_i's guarantee.

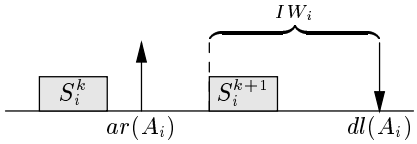
Let S_i^k denote the current invocation of S_i, i.e., the last invocation of S_i before A_i arrived. Let IW_i be the *interference window* of S_i, that is the time interval in which S_i may preempt and interfere with the execution of A_i. The following cases can be identified:

case 1: S_i^k is unknown, i.e., the sporadic task S_i has not started yet to invoke its instances. S_i can arrive any time and we must assume the worst case, that is S_i will start to invoke its instances at t, with maximum frequency. The interference window is the entire execution window of A_i, $IW_i = EW$.

case 2: S_i^k is known, i.e., S_i has invoked an instance before t. The following sub-cases can occur:

a) $start(S_i^k) + \lambda \leq t$, i.e., the last invocation completed before A_i arrived, and the next invocation, S_i^{k+1}, could have arrived but it has not yet. This means S_i^{k+1} can enter A_i's execution window at any time, thus the same as in case 1, $IW_i = EW$.

b) $end(S_i^k) \leq t < start(S_i^k) + \lambda$, i.e., the current invocation S_i^k has completed before t, and the next one has not arrived yet. But now we know that the next one, S_i^{k+1} will not arrive until λ time slots, counted from the start time of S_i^k.



This means the interference window can be decreased with the amount of slots in EW for which we know that S_i^{k+1} cannot possibly arrive:

$$IW_i = [start(S_i^k) + \lambda, dl(A_i)]$$

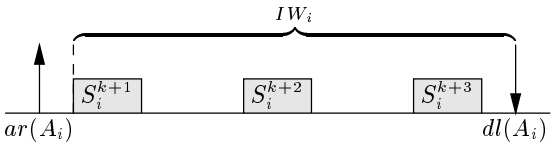
c) $t < end(S_i^k)$, i.e., the current invocation is still executing. In the worst case, the interference window is entire EW, $IW_i = EW$.

The processor demand approach, [15], can be used to determine the total processing time needed for all sporadic tasks, $c(\mathcal{S})$, which will invoke their instances within their interference windows:

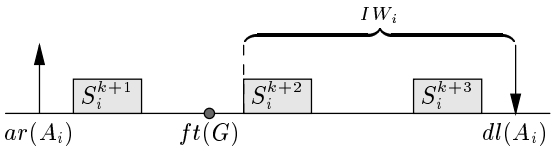
$$c(S) = \sum_{i=1}^{n} \left\lceil \frac{|IW_i|}{\lambda(S_i)} \right\rceil c(S_i) \tag{2}$$

Now we will see how the interference window can actually get "shrunk" when guaranteeing a firm aperiodic task A under runtime. It is usually not the case that A will start to execute as soon it arrives. This because of the offline tasks and previously guaranteed firm aperiodic tasks. In section 4, we presented a method for guaranteeing firm aperiodic tasks on top of offline tasks. The start time of the firm aperiodic task which is currently tested for acceptance is based on the finishing time of its predecessor, i.e., another firm aperiodic task with earlier deadline. Hence, in some cases the start of the interference window IW_i is set to the finishing time of A's predecessor.

Here is an example: assume a firm aperiodic task A to be accepted and a sporadic task S_i as in *case 2b* above. The interference window is defined as below:



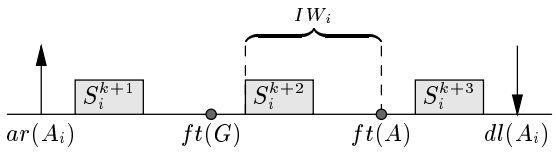
Assume another previously accepted firm aperiodic task G which will delay the execution of A:



We see that the earliest time A can start is set to the finishing time of its predecessor, $ft(G)$. So, all invocation of S_i that occured before $est(A)$ are taken care of when calculating $ft(G)$, and are not concidered when calculating $ft(A)$. The start of the interference window is now set to the start time of the first possible instance of S_i that can interfere with A, that is S_i^{k+2}.

Now we calculate the finishing time of A using the algorithm described in section 4. Without sporadic tasks, A would be guaranteed to finish at time $ft(A)$. Since A is

guaranteed to finish before its deadline, we do not need to take into consideration the impact from S_i after the finishing time of A. Hence, the end of the interference window IW_i is set to $ft(A)$.



So, what actually happens is that only one instance of S_i is concidered when calculating $ft(A)$.

Now we can formalize what the impact of a sporadic task S_i on a firm aperiodic task A_i: Let A_{i-1} be the predecessor of A_i, i.e., the last firm aperiodic task that is guaranteed to execute before A_i (task G in example above). If S_i has not yet started to invoke its instances at the time we start with the acceptance test for A_i, we must assume the worst case, that is the first instance of S_i will start at the same time as the earliest start time of A_i:

$$est(S_i^1) = est(A_i) = max(t, ft(A_{i-1}))$$

We have *max* because A_{i-1} could have completed before the current time t, or A_i has no predecessor at all.

On the other hand, if S_i has started to invoke its instances, we can calculate when the next one after the earliest possible time of A_i can occur (S_i^{k+2} in example above):

$$est(S_i^{k+m}) = est(S_i^k) + \left\lceil \frac{ft(A_{i-1}) - est(S^k)}{\lambda(S_i)} \right\rceil \lambda(S_i)$$

To conclude, the time interval IW_i in which a sporadic task S_i may preempt and interfere with the execution of a firm aperiodic task A_i is obtained as:

$$IW_i = [\delta, ft(A_i)] \tag{3}$$

where δ is the earliest possible time S_i could preempt A_i and is calculated as:

$$\delta = \begin{cases} est(S_i^{k+m}) & \text{if } S_i \text{ known} \\ max(t, ft(A_{i-1})) & \text{otherwise} \end{cases} \tag{4}$$

The index $k + m$ points out the first possible invocation of S_i which has earliest start time after the finishing time of A_i's predecessor.

5.3 Algorithm description

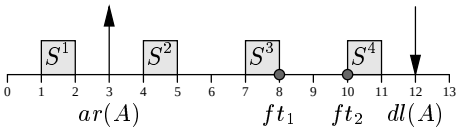
Assume a firm aperiodic task A_i that is tested for acceptance upon its arrival time, current time t. We want to make sure that A_i will complete before its deadline, without causing any of the guaranteed tasks to miss its deadline. A guaranteed task is either an offline task, a previously guaranteed firm aperiodic task or a sporadic task. Offline and sporadic tasks are guaranteed before the run-time of the system, [8], [11], while firm aperiodic tasks are guaranteed online, upon their arrival. The guarantee algorithm is performed as follows:

step 1: Assume no sporadic tasks and calculate the finishing time of A_i based only on offline tasks and previously guaranteed firm aperiodic tasks (as described in section 4).

step 2: Calculate the impact from all sporadic tasks that could preempt A_i before its finishing time calculated in the previous step (equation 2).

step 3: If the impact is greater than zero, the finishing time of A_i will be postponed (moved forward), because at run-time we need to execute all sporadic instances with deadlines[2] less than $dl(A_i)$. The impact reflects the amount of A_i that is to be executed after the finishing time calculated in step 1. Now we treat the remaining part of A_i as a firm aperiodic task and repeat the procedure (go to step 1). But this time we start calculating the sporadic impact at the finishing time of the first part of A_i. The procedure is repeated until there is no impact from sporadic tasks on A_i.

Example: assume a firm aperiodic task A which arrives at current time $t = 3$, with the execution demand $c(A) = 5$ and deadline $dl = 12$. Also assume a sporadic task S that has started to invoke its instances before t, in slot 1, with a minimum inter-arrival time $\lambda = 3$ and worst case computation time $c(S) = 1$. For simplicity reasons, assume no offline tasks and no previously guaranteed hard aperiodic tasks. First we calculate the finishing time of A, without considering the sporadic task S, i.e., $ft_1 = 8$.



The interference window of A is $IW_i = [4, 8]$. The impact of S in IW_i is equal to 2 (two instances). Now we take the impact (which tells us how much A is delayed by S) and calculate its finishing time, starting at time $t_1 = ft_1$, i.e., $ft_2 = 10$. We must check if we have any sporadic instances in the new interference interval $IW_i^{'} = [10, 10]$

[2]The deadline of a sporadic instance is set to the earliest start time of the next instance

(note that original $IW_i^{'}$ would be $[8, 10]$, but we always take the start time of the next instance after the previous finishing time, in this case $est(S^4) = 10$). The new impact is zero, which means that we can stop and the last calculated finishing time, $ft_2 = 10$, is the finishing time of A.

Implementation The first part of the algorithm is exactly the same as described in 4: first we locate the position of hard aperiodic task to be guaranteed, calculate its finishing time and check if any of previously guaranteed hard aperiodic tasks will miss its deadline. The second part, that calculates the finishing time, is extended to handle the impact from the sporadic tasks as follows:

$getFinishingTime(ft_{pred}, c)\{$
/*determine ft without sporadics by "filling up"
free slots until the c is exhausted.*/
$\forall S_i \in \mathcal{S}$
if S_i started to invoke
$\delta = est(S_i^{k+m})$ /*eq (4)*/
else
$\delta = max(t, ft_{pred})$
$IW_i = [\delta, ft]$ /*eq (3)*/
$sum = sum + \left\lceil \frac{|IW_i|}{\lambda(S_i)} \right\rceil c(S_i)$ /*eq (2)*/
if $sum \neq 0$
$getFinishingTime(ft, sum)$
else
return ft
$\}$

The recursive formulation was chosen for simplicity of explanation: our implemtentation uses a loop. In the loop, time is increased from current to finish time, without going back. Thus the complexity remains linear, similar to the finishing time algorithm in 4.

6 Simulation Analysis

We have implemented the described mechanisms and have run simulations with the RTSim simulator [3] for various scenarios. We have tested the acceptance ratio for firm aperiodic tasks with the methods to handle sporadic tasks described in 5: no sporadic tasks for reference purposes ("no sporadics"), worst case arrivals without knowledge about invocations ("no info"), and updated worst case with arrival info ("updated"). We randomly generate offline tasks, sporadic and aperiodic task loads. The results were obtained for an offline scheduled task load of 0.5 and schedule length of 100 slots. We studied the acceptance ratio AR of the randomly arriving aperiodic tasks under randomly generated arrival patterns for the sporadic tasks. The worst case sporadic load, i.e., all sporadic tasks arriving with maximum frequency was set to 0.2. The arrival frequencies of sporadic tasks were set according to a factor, F_{fact}, such that $interarrivaltime = minimum\ interarrivaltime \times F_{fact}$. Deadlines for firm aperiodic tasks were generated randomly within one schedule length. The maximum load demanded by the aperiodic tasks is 0.44.

Each point represents a sample size of some 1000 tests. 0.95 confidence intervals were smaller than 5%. We can see that our method improves the acceptance ratio of firm aperiodic tasks. This results from the fact that our methods reduces pessimism about sporadic arrivals by keeping track of arrivals.

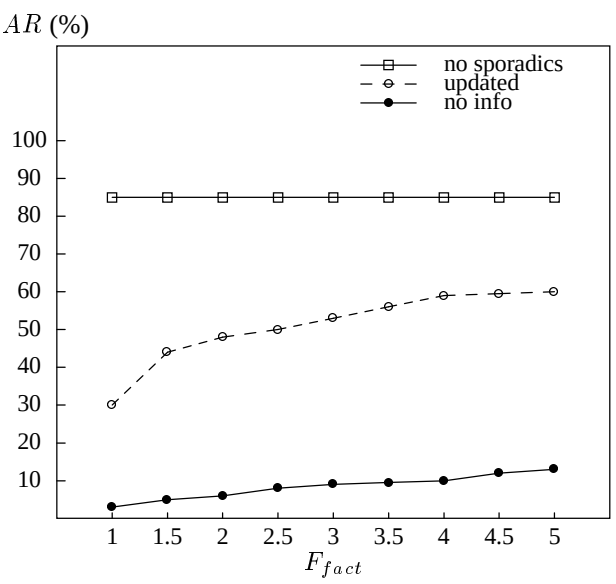


Figure 2. Guarantee Ratio for Firm Aperiodic Tasks

7 Summary and Outlook

In this paper we have presented methods to schedule sets of mixed types of tasks with complex constraints, by using earliest deadline first scheduling and offline complexity reduction. In particular, we have proposed an algorithm to handle sporadic tasks to improve response times and acceptance of firm aperiodic tasks.

We have presented the use of an offline scheduler to transform complex constraints of tasks into starttimes and deadlines of tasks for simple EDF runtime scheduling. We provided an extension to EDF, two level EDF, to ensure feasible execution of these tasks in the presence of additional tasks or overloads. During offline analysis our algorithm determines the amount and location of unused resources,

which we use to provide $O(N)$ online acceptance tests for firm aperiodic tasks. We presented an algorithm for handling offline guaranteed sporadic tasks, which keeps track of arrivals of instances of sporadic tasks at runtime. It uses this updated information to reduce pessimism about future sporadic arrivals and improve response times and acceptance of firm aperiodic tasks. Results of simulation study show the effectiveness of the algorithms.

Future research will deal with extending the algorithm to include interrupts, overload handling, and aperiodic and sporadic tasks with complex constraints as well. We are studying the inclusion of server algorithms, e.g., [1] into our scheduling model by including bandwidth as additional requirement in the offline transformation.

8 Acknowledgements

The authors wish to express their gratitude to Tomas Lennvall, Radu Dobrin, and Joachim Nilsson for useful discussions and to the reviewers for their helpful comments. The work presented in this paper was partly supported by the Swedish Science Foundation via ARTES.

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of Real-Time Systems Symposium*, Dec. 1998.

[2] S. Baruah, G. Buttazzo, S. Gorinsky, and G. Lipari. Scheduling periodic task systems to minimize output jitter. In *Sixth International Conference on Real-Time Computing Systems and Applications*, Dec. 1999.

[3] G. Buttazzo, A. Casile, G. Lamastra, and G. Lipari. A scheduling simulator for real-time distributed systems. In *Proceedings of the IFAC Workshop on Distributed Computer Control Systems (DCCS '98)*, 1999.

[4] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems Journal*, 2(3):181–194, Sept. 1990.

[5] R. Davis, K. Tindell, and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In *Proceddings of the Real-Time Symposium*, pages 222–231, Dec. 1993.

[6] M. DiNatale and J. Stankovic. Applicability of simulated annealing methods to real-time scheduling and jitter control. In *Proceedings of Real-Time Systems Symposium*, Dec. 1995.

[7] G. Fohler. *Flexibility in Statically Scheduled Hard Real-Time Systems*. PhD thesis, Technische Universität Wien, Austria, Apr. 1994.

[8] G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proc. 16th Real-time Systems Symposium*, Pisa, Italy, 1995.

[9] M. Garey, D. Johnson, B. Simons, and R. Tarjan. Scheduling unit-time tasks with arbitrary release times and dead lines. *IEEE Transactions on Software Engineering*, 10(2):256–269, May 1981.

[10] R. Gerber, W. Pugh, and M. Saksena. Parametric dispatching of hard real-time tasks. *IEEE Transactions on Computers*, 44(3), March 1995.

[11] D. Isovic and G. Fohler. Handling sporadic tasks in off-line scheduled distributed hard real-time systems. *Proc. of 11th EUROMICRO conf. on RT systems, York, UK*, June 1999.

[12] D. Isovic and G. Fohler. Online handling of hard aperiodic tasks in time triggered systems. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, June 1999.

[13] F. Jahanian, R. Lee, and A. Mok. Semantics of modechart in real time logic. In *Proc. of the 21st Hawaii International Conference on Systems Sciences*, pages 479–489, Jan. 1988.

[14] K. Jeffay. Scheduling sporadic tasks with shared resources in hard real-time systems. *Dept. of Comp. Sci., Univ. of North Carolina at Chapel Hill*, 1992.

[15] 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.

[16] H. Kopetz. Sparse time versus dense time in distributed real time systems. *In Proc. of the Second Int. Workshop on Responsice Comp. Sys., Saitama, Japan*, Oct. 1992.

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

[18] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, MIT, 1983. Report MIT/LCS/TR-297.

[19] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *RT Systems Journal*, 1989.

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

[21] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems Journal*, 1(1):27–60, June 1989.

[22] M. Spuri, G. C. Buttazzo, and F. Sensini. Robust aperiodic scheduling under dynamic priority systems. *Proceedings of Real-Time Systems Symposium*, Dec. 1995.

[23] J. Stankovic, K. Ramamritham, and C.-S. Cheng. Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems. *IEEE Trans. on comp., 34(12)*, Dec 1995.

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

[25] T. Tia, W. Liu, J. Sun, and R. Ha. A linear-time optimal acceptance test for scheduling of hard real-time tasks. *Dept. of Comp. Sc., Univ. of Illinois at Urbana-Champaign§*, 1994.

[26] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 50(2-3), 1994.

[27] M. Törngren. Fundamentals of implementing real-time control applications in distributed computer systems. *Real-Time Systems*, 1997.

[28] J. Wurtz and K. Schild. Scheduling of time-triggered real-time systems. Technical report, German Research centre for Artificial Intelligence - DKFI GmbH, 1997.

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