Handling Sporadic Tasks in Off-line Scheduled Distributed Real-Time Systems

Damir Isović and Gerhard Fohler
Department of Computer Engineering
Mälardalen University, P.O. Box 883, 721 23 Västerås, Sweden
{dic,gfr}@mdh.se

Abstract

Many industrial applications mandate the use of a time-triggered paradigm and consequently the use of off-line scheduling for reasons such as predictability, certification, cost, or product reuse. The construction of an off-line schedule requires complete knowledge about all temporal aspects of the application. The acquisition of this information may involve unacceptable cost or be impossible. Often, only partial information is available from the controlled environment.

In this paper[1]*, we present an algorithm to handle event-triggered sporadic tasks, i.e., with unknown arrival times, but known maximum arrival frequencies, in the context of distributed, off-line scheduled systems. Sporadic tasks are guaranteed during design time, allowing rescheduling or redesign in the failure case. At run-time, the sporadic tasks are scheduled dynamically, allowing the reuse of resources reserved for, but not consumed by the sporadic tasks. We provide an off-line schedulability test for sporadic tasks and apply the method to perform on-line scheduling on top of off-line schedules. Since the major part of preparations is performed off-line, the involved on-line mechanisms are simple. The on-line reuse of resources allows for high resource utilization.*

1. Introduction

Off-line scheduling is mandated by a number of industrial applications. Predictability, cost, product reuse, and maintenance are examples for reasons advocating a time-triggered approach. The off-line construction of schedules, however, requires complete knowledge about application characteristics before the run-time of the system. Often, such comprehensive information is inaccessible due to high cost of acquisition or unavailability. Rather, incomplete characteristics are available. One typical, partially known property is the arrival time of activities. Instead of exact arrival patterns, bounds on the arrival frequencies are known, e.g., derived from a model of a physical process. One such limit is the *minimum inter-arrival time* between subsequent instances of tasks. Tasks for which it is known are called *sporadic tasks* [13].

The use of off-line scheduling for sporadic tasks is problematic; they can be transformed into pseudo-periodic tasks [13], but with potentially prohibitive overhead.

An on-line algorithm for scheduling sporadic tasks with shared resources in hard real-time systems has been presented in [10]. Scheduling of sporadic requests with periodic tasks on an *earliest-deadline-first (EDF)* basis [12] has been presented in [17]. Handling of firm aperiodic requests using a Total Bandwidth Server has been presented in [15]. On-line guarantees of aperiodic tasks in firm periodic environments, where tasks can skip some instances, have been described in [5]. Systems containing hard real-time sporadic tasks have been analyzed for their worst case behavior in [2].

These algorithms above perform on-line guarantees. When a set of sporadic tasks arrives at run-time to the system, a scheduler performs an acceptance test. The test succeeds if each sporadic task in the set can be scheduled to meet its deadline, without causing any off-line guaranteed periodic tasks or previously accepted non-periodic tasks to miss its deadline, else it is rejected. A disadvantage with this approach is that if the set has been rejected, it is too late for countermeasures.

An off-line guarantee algorithm for sporadic tasks based on bandwidth reservation has been presented in [3] for single processor systems.

In this paper, we present a method providing an off-line feasibility test for sporadic tasks on top of an off-line scheduled, distributed periodic task set with general constraints, e.g., precedence; rescheduling or redesign can be performed, should the test fail. Given the deadline, maximum frequency and execution time for each task in the sporadic task set, we can create a worst case load pattern. Then,

[1] In proceedings of 11th EUROMICRO conference on real-time systems, York, UK

we try to guarantee this worst case sporadic demand within the periodic schedule before the system starts its execution. We have to account for arrivals at any time; it is, however, sufficient to investigate only some selected points in time. Our method is based on the *slot shifting* [8] method, which provides for the combination of off-line and on-line scheduling. After a static schedule for the distributed periodic tasks has been created off-line in a first step, the amount and distribution of unused resources and leeways in it is determined. These are then used to incorporate aperiodic tasks into the schedule by shifting the off-line scheduled tasks' execution, without violating their feasibility. An on-line mechanism is used to guarantee and schedule aperiodic tasks. Using the on-line scheduling algorithm of slot shifting, the resources reserved for, but not consumed by sporadic tasks can be reused. Should sporadic tasks arrive at less than their guaranteed maximum frequency, their resources can be reclaimed, e.g., for aperiodic tasks. Furthermore, the on-line algorithm of slot shifting has been modified to schedule guaranteed sporadic requests.

The method presented in this paper allows for the handling of distributed periodic tasks with general constraints, such as precedence, based on the time-triggered paradigm, together with the event-triggered scheduling of guaranteed sporadic tasks and online aperiodic tasks, possibly reclaiming resources.

The rest of this paper is organized as follows: First, a description of system and task model is given in section 2, and a brief summary of the slot shifting method in section 3. The off-line guarantee test for the sporadic tasks is presented in section 4, followed by the online mechanisms in section 5. An example in section 6 illustrates the discussed mechanisms. Finally, section 7 concludes the paper.

2. System description and task model

The system is considered to be *distributed*, i.e., one that consists of several processing and communication nodes [16].

2.1. Time model

We assume a discrete time model [11]. Time ticks are counted globally, by a synchronized clock with granularity of slot length, 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.

2.2. Off-line periodic schedule

A schedule is a sequence of *n* slots. For static schedules the number of slots is typically equal to the *least common multiple (LCM)* of all involved periods.

2.3. Task model

All tasks in the system are fully preemptive and 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 maximum execution time ($MAXT$) [14], period (P) and relative deadline of (Dl).
The k^{th} invocation of T_P is denoted T_P^k and is characterized by its earliest start time (est) and absolute deadline (dl). The absolute deadline of the k^{th} invocation of T_P is equal to the sum of the earliest start time of its preceding invocation and the relative deadline.

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

Sporadic tasks arrive to the system at random points in time, but with defined minimum inter-arrival times between two consecutive invocations. We do not know when they arrive to the system, but we do know their maximum frequency. A sporadic task T_S is characterized by its relative deadline, minimum inter-arrival time (λ) and maximum execution time.
The attributes above are known before the run-time of the system. The additional information that becomes available on-line, upon the arrival time of the k^{th} invocation is its arrival time and its absolute deadline.

3. Integrated off-line and on-line Scheduling

In this section, we briefly describe the slot shifting method which we use as a basis to combine off-line and on-line scheduling. It provides for the efficient handling and possibly on-line guarantee of aperiodic tasks on top of a distributed schedule with general task constraints. Slot shifting extracts information about unused resources and leeway in an off-line schedule and uses this information to add tasks feasibly, i.e., without violating requirements on the already scheduled tasks. A detailed description can be found in [8].

3.1. Off-line preparations

First, an off-line scheduler [4] creates scheduling tables for the periodic tasks. It allocates tasks to nodes and resolves precedence constraints by ordering task executions.

Start-times and deadlines The scheduling tables list fixed start- and end times of task executions, that are less flexible than possible. The only assignments fixed by specification, however, are the initiating and concluding tasks in the precedence graph, and, as we assume message transmission times to be fixed here[2], tasks sending or receiving inter-node messages. These are the only fixed start-times and deadlines, all others are calculated recursively, as the execution of all other tasks may vary within the precedence order, i.e., they can be shifted.

Intervals and spare capacities 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. Spare capacities 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.

The spare capacities of an interval I_i are calculated as given in formula 1:

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

The length of I_i minus the sum of the activities assigned to it is the amount of idle times 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.

3.2. On-Line mechanisms

During system operation, the on-line scheduler is invoked after each slot. It checks whether aperiodic tasks have arrived, performs the guarantee algorithm, and selects a task for execution. This decision is then used to update the spare capacities. Finally the scheduling decision is executed in the next slot.

Guarantee Algorithm 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 spare capacity 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 $dl(T_A)$, and
- $min(sc(I_{l+1}), dl(T_A) - start(I_{l+1}))$, 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 $MAXT(T_A)$, 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. This guarantee algorithm is $O(N)$, N being the number of intervals. It is shown in [7], that this acceptance test has equivalent results – but with simpler run-time handling – as to the ones presented in [9] and [6], which are optimal for single processors.

On-line scheduling On-line 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. Soft aperiodic tasks, i.e., without deadline, can be executed immediately if $sc(I_c) > 0$. After each scheduling decision, the spare capacities of the affected intervals are updated.

4. Acceptance test for a set of sporadic tasks

In this section we will introduce an off-line guarantee algorithm for a set of sporadic tasks. The set is said to be *feasible* with the already scheduled task set if it is possible to schedule all tasks in the sporadic set such that no scheduled periodic task misses its deadline.
Firstly, the off-line periodic schedule is created and analyzed for slot shifting. Secondly, the set of sporadic tasks is tried to fit into the periodic schedule, by investigating only selected time slots. If the sporadic set is not accepted, it is up to designer to redesign the system, i.e., reschedule periodic tasks or change the sporadic set.

4.1. Sporadic set

All tasks in the sporadic set are assumed to be invoked with their maximum frequency, creating the worst case scenario for the scheduler. If the deadline of a sporadic task can be guaranteed for the release with their maximum frequency, then all subsequent deadlines are guaranteed. Examples of this approach are given in [1]. The minimum time difference between successive releases of a sporadic task is its minimum inter-arrival time. It has been shown [2] that a sporadic task which is released with its maximum frequency behaves exactly like a periodic task with period

[2] We apply the same mechanisms to the network as well, i.e., shifting messages, as detailed in [8].

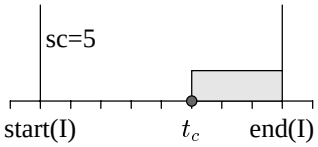


Figure 1. Example of a critical slot.

equal to its minimum inter-arrival time. Now we know the deadline, the maximum execution time and the 'period' of each sporadic task in the set and we can use that information to try to guarantee the set for its worst load pattern.

4.2. Critical slots

One way of investigating if the sporadic set fits into the periodic schedule is to investigate if it fits at each time slot of the periodic schedule, but this is impractical. It is sufficient to investigate only some selected points in time, called *critical slots (t_c)*. There is only one critical slot per interval[3] , and if the sporadic set can be guaranteed at the critical slot, it will be guaranteed at every other slot within the same interval.
The worst case for arrival of the sporadic set to an interval I is the slot where the execution of the sporadic tasks can be delayed maximally by the execution of the off-line scheduled tasks. This gives:

Proposition 1 Critical slot t_c for an interval I is calculated as:

$$t_c(I) = start(I) + sc(I)$$

as depicted in figure 1. If a sporadic task T_S can be guaranteed at the critical slot, it will be guaranteed at each other slot within the same interval I:

$$\forall t \in I, T_S\ guaranteed\ at\ t_c \Rightarrow T_S\ guaranteed\ at\ t$$

Proof. Let:

$t_c(I) = start(I) + sc(I)$ - the critical slot of I.
$t \in I, t \neq t_c$ - some other slot in I.

Assume the following is correct:

Assumption 1. *There is a slot t in interval I such that a sporadic task T_S can be guaranteed at the critical slot t_c, but not at t:*

$$\exists t \in I, (T_S\ guaranteed\ at\ t_c) \wedge (T_S \neg guaranteed\ at\ t)$$

Let δ denote the difference between spare capacities available for T_S at t_c and t, i.e., the amount of spare capacity that we may get or lose by shifting the arrival time of T_S from t_c to t. Assumption 1 states that T_S can be guaranteed at t_c but not at t, which means that there is more spare capacity available at t_c than at t and we lose spare capacity if we shift. This implies that δ is negative:

$$\delta < 0 \tag{2}$$

There are two possibilities for the arrival of the sporadic task T_S, $t = A(T_S)$, before or after the critical slot t_c:

Case 1: $t > t_c$, T_S arrives after t_c, as depicted in figure 2.

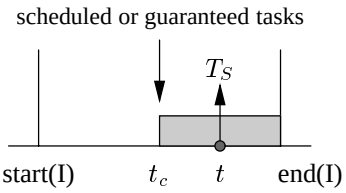


Figure 2. T_S arrives after t_c.

The requirement for T_S to be accepted is that the spare capacity available for T_S at its arrival time has to be greater or equal to the maximum execution time of T_S.
Let α and β denote the change of spare capacity caused by shifting in the arrival and deadline, resp., interval as depicted in figure 3:

- α - the difference in spare capacity of the arrival interval caused by shifting the arrival time of T_S from t_c to t.
- β - the difference in spare capacity of the deadline interval caused by shifting the deadline of T_S.

This gives:

$$\delta = \alpha + \beta \tag{3}$$

Shifting the arrival time of T_S from t_c to t means that the deadline of T_S is shifted to the right. In the arrival interval, $I_{arrival}$, slots from t_c to t are reserved for the execution of the scheduled periodic tasks, giving $\alpha = 0$. In the deadline interval, $I_{deadline}$, shifting the deadline of T_S may only increase the portion of available spare capacities in that interval. This gives that β has to be greater or equal to zero ($\beta \geq 0$).

The maximum value of δ occurs when the deadline of T_S does not intersect with any other activity, that is, execution of some other task. In other words, $\beta = t - t_c > 0$. If so, then:

$$\delta = \alpha + \beta > 0, (\alpha = 0, \beta > 0) \tag{4}$$

Otherwise, if $dl(T_S)$ occurs during the execution of some other task, the worst case scenario is that we do not get any

[3]Intervals are calculated as described in section 3.

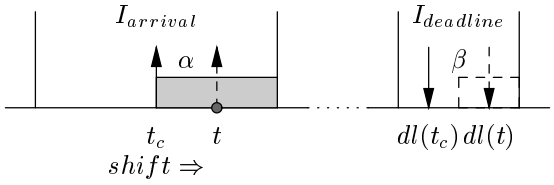


Figure 3. Arrival time of T_S shifted to the right.

new resources for T_S, that is:

$$\delta = \alpha + \beta = 0, (\alpha = 0, \beta = 0) \quad (5)$$

$(4) \wedge (5)$ gives:

$$\delta \geq 0$$

which is contradictory to (2), making assumption 1 false.

Case 2: $t < t_c$, T_S arrives before t_c, as depicted in figure 4.

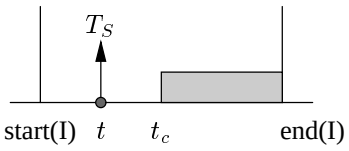


Figure 4. $\mathbf{T_S}$ arrives before $\mathbf{t_c}$.

Now we shift the arrival time of T_S to the left, that is before the critical point t_c. This is shown in figure 5.

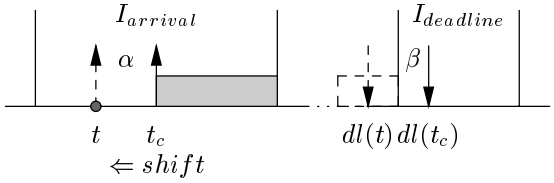


Figure 5. Arrival time of T_S shifted to the left.

Let α and β denote the same as in case 1. Shifting the arrival time of T_S results in a positive α that is equal to the difference between t_c and t, i.e., $\alpha = t_c - t > 0$. In the deadline interval, the amount of lost spare capacities caused by shifting can maximally be the same as the amount of gained spare capacities in the arrival interval, giving $\beta_{worst} = -\alpha$. This implies:

$$\delta = \alpha + \beta = \alpha + (-\alpha) = 0, (\beta = \beta_{worst}) \quad (6)$$

In a more optimistic scenario, we can even lose less spare capacities in the deadline interval than we get in the arrival interval, that is $\beta < \alpha$. In that case, we get:

$$\delta = \alpha + \beta > 0, (|\beta| < \alpha) \quad (7)$$

$(6) \wedge (7)$ implies:

$$\delta \geq 0$$

which is contradictory to (2). This implies the assumption 1 doesn't hold for case 2.
Assumption 1 doesn't hold either for case 1 or case 2. Therefore proposition 1 is true. This concludes the proof. □

Critical points are calculated off-line for each interval, and only those points are checked for the feasibility of the sporadic task set.

4.3. Off-line feasibility test for sporadic tasks

The feasibility test for the set of sporadic tasks works by creating a worst case load demand of the sporadic tasks as described in section 4.1. We assume that all sporadic tasks arrive with their maximum frequency and test if the demand created can be accommodated into the static schedule at all critical slots. Here follows the off-line guarantee algorithm for a set of sporadic tasks $\mathcal{S}$:

Let:

i	= index of $I_{arrival}$
k	= index of $I_{deadline}$
sc_a	= available sc for a sporadic task T_S from $a(T_S)$ to $dl(T_S)$
R	= an array containing slots reserved for previously guaranteed sporadic tasks
$initR()$	= initiates R to empty set
$countR(x, y)$	= returns number of reserved slots between slot x and slot y.
$reserveR(n, d)$	= reserves n slots as close to d as possible (as late as possible)

1: $\forall t_c$
2: $initR()$
3: $\forall T_S \in \mathcal{S}$
4: $\forall T_S^n \in LCM_{\mathcal{S}}$
5: $sc_a(T_S^n) = \sum_{j=i+1}^{k-1} sc(I_j)$
6: $+min(sc(I_k), dl(T_S^n) - start(I_k))$
7: $-countR(a(T_S^n), dl(T_S^n))$
8: $if \quad (sc_a(T_S^n) \geq MAXT(T_S))$
9: $then \quad reserveR(MAXT(T_S), dl(T_S^n))$
10: $else$ abort (set rejected)

Comments:
1: Investigate every critical slot.
2: No slots reserved yet.
3: Guarantee every sporadic task T_S in the set.

4: Guarantee every invocation T_S^n of T_S.
5: Calculate sc available for T_S from its arrival until its deadline. It is equal to the sum of sc for all full intervals between $I_{arrival}$ and the $I_{deadline}$ of T_S^n, increased by
6: the remaining sc of the $I_{deadline}$ available until $dl(T_S^n)$, decreased by
7: the amount of sc reserved for other, previously guaranteed sporadics that intersect with T_S^n.
8: If the available sc is greater or equal to the maximum execution time of T_S, then
9: reserve slots needed for T_S^n as close to its dl as possible, and continue.
10: If not enough spare capacity, abort the guarantee algorithm and report that the guaranteeing failed.

5. On-line mechanism

During the system operation, the on-line scheduler is invoked after each slot. It checks whether new dynamic tasks have arrived during the last slot. When a set of sporadic tasks arrives to the system, the ready set of slot shifting is expanded by sporadic tasks that are ready to execute. Soft aperiodic tasks can be executed if the spare capacity of the current interval is greater than zero, and there are no ready sporadic tasks.

Let:

t	= current time
$sc(I)_t$	= spare capacity of the current interval at time t.
$\mathcal{R}(t)$	= the ready set that consists of all periodic and guaranteed sporadic tasks that have earliest start time less or equal to the current time.

We identify the following cases:

1. $\mathcal{R}(t) = \{\}$: There are no tasks ready to be executed, the CPU remains idle.

2. $\mathcal{R}(t) \neq \{\} \wedge\ \exists T_A,\ T_A$ soft aperiodic:

   (a) $sc(I)_t > 0 \wedge \exists T_S \in \mathcal{R}(t), T_S$ sporadic $\Rightarrow$ execute T_S.

   (b) $sc(I)_t > 0 \wedge \neg\exists T_S \in \mathcal{R}(t) \Rightarrow$ execute T_A.

   (c) $sc(I)_t = 0$: a periodic task from ready set has to be executed. Zero spare capacities indicate that adding further activities will result in a deadline violation of the guaranteed task set.

3. $\mathcal{R}(t) \neq \{\} \wedge \neg\exists T_A, T_A$ soft aperiodic: The task of ready set with the shortest deadline is executed.

5.1. Maintenance of spare capacities

The decision of the scheduler is now used to update spare capacities, depending on which type of task was selected for execution:

- *Aperiodic execution*: one slot of the spare capacities is used to execute a slot of dynamic task. The spare capacity of the current interval has to be decremented by one.
- *Periodic execution*: Executing a static task only swaps spare capacities. Depending on the interval to which the executed task belongs to, the current interval I_i, or a subsequent one $I_j, j > i$ is affected. The amount of total spare capacities is unchanged.
- *Sporadic execution*: The spare capacity of the current interval is decremented by one.
- *No execution*: One slot of spare capacity is used without dynamic processing. Spare capacity has to be decremented by one.

6. Example

Assume the following periodic tasks with execution times and precedence constraints as described in figure 6.

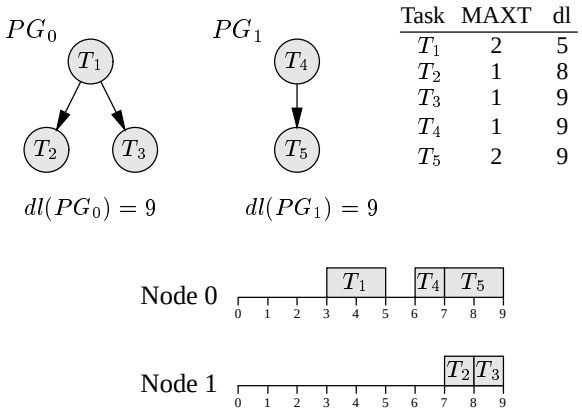


Figure 6. Example Tasks and Schedule.

We calculate intervals and spare capacities, as described in 3.1, and critical slots as described in 4.2:

$Interval$	$Node$	$start$	end	sc	t_c
I_0	0	0	5	3	3
I_1	0	5	9	1	6
I_2	1	6	8	1	7
I_3	1	8	9	0	8

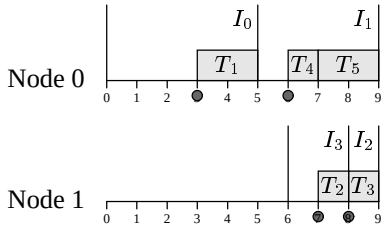


Figure 7. The schedule with intervals.

Intervals with their assigned tasks and critical slots are depicted in figure 7. Now assume a sporadic set $\mathcal{S} = \{S_1(1,5), S_2(3,10)\}$ where the first parameter is maximum execution time and the other one the minimum inter-arrival time at node 0. If we assume that sporadic tasks arrive with their maximum frequencies, then the deadline of each invocation is equal to the release of the next invocation. Now we apply the off-line guarantee algorithm on each task in the sporadic set $\mathcal{S}$. First, we try to guarantee S_1 and S_2 at critical slot 3, and if they can be guaranteed, we proceed with investigation of slot 6. The LCM of $\mathcal{S}$ is 10, which means that S_1 is invoked twice and S_2 once before the pattern is repeated. We now illustrate the guarantee test for S_1 and S_2. Numbers above columns represent steps in the guarantee algorithm described in 4.3 (see figure 8 in parallel):

1	3	4	5	8	9	10
$\mathbf{t_c}$	$\mathbf{T_S}$	**Inv.**	$\mathbf{sc_a}$	$\geq \mathbf{MAXT}(?)$	$\mathbf{R}$	
3	S_1	1	1	$\geq 1 \Rightarrow \top$	{5}	
		2	3	$\geq 1 \Rightarrow \top$	{5,11}	
	S_2	1	2	$\geq 3 \Rightarrow \bot$		abort

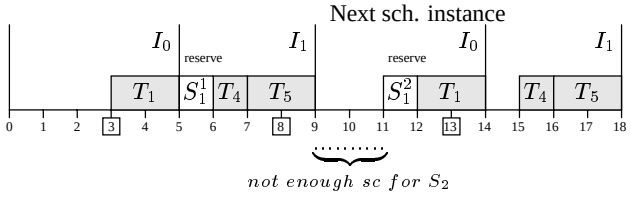


Figure 8. Steps in guarantee algorithm.

The sporadic set cannot be guaranteed at critical slot 3. What we can do now is to redesign the system and try again. Since we support distributed systems, we could reallocate some of the periodic tasks from node 0 to node 1, or allocate some of sporadics on node 1. In this example, we decide to schedule the periodic task T_4 on node 1 instead of node 0. The new periodic schedule is depicted in figure 9. Intervals remain the same, spare capacities and critical slots have to be recalculated for I_1 and I_2:

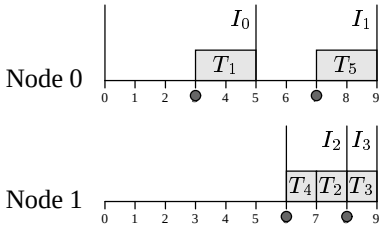
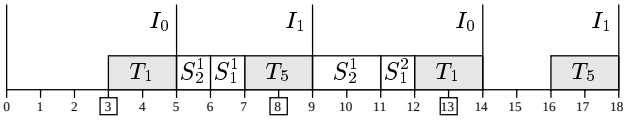


Figure 9. The schedule after redesign.

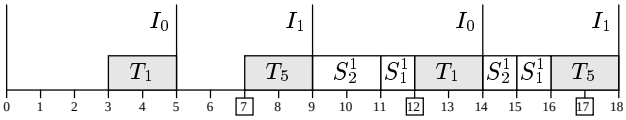
I_1 : $sc(I_1)$ = 1+1=2, $t_c(I_1)$ = 5+2=7 $\quad$ I_2 : $sc(I_2)$ = 1-1=0, $t_c(I_2)$ = 6+0=6

We try to guarantee $\mathcal{S}$ on node 0 again, but this time one critical point has changed; we got 7 instead for 6:

1	3	4	5	8	9
$\mathbf{t_c}$	**Task**	**Inv.**	$\mathbf{sc_a}$	$\geq (?)$	$\mathbf{R}$
3	S_1	1	2	$\geq 1 \Rightarrow \top$	{6}
		2	3	$\geq 1 \Rightarrow \top$	{6,11}
	S_2	1	3	$\geq 3 \Rightarrow \top$	{5,6,9,10,11}
7	S_1	1	3	$\geq 1 \Rightarrow \top$	{11}
		2	2	$\geq 1 \Rightarrow \top$	{11,15}
	S_2	1	3	$\geq 3 \Rightarrow \top$	{9,10,11,14,15}



a) Critical slot 3



b) Critical slot 7

Figure 10. Guaranteeing after redesign.

The sporadic set is guaranteed at both critical slots and we can accept it.

Next follows the description of on-line execution (as described in section 5) on node 0 for the periodic and sporadic tasks described above, extended with one soft aperiodic task, $A_1(2,2)$, where the first parameter is arrival time, and the other one is execution time. Assume $\mathcal{S}$ arrives at slot 3. $\mathcal{R}(t)$ containts tasks that are ready to execute in slot t. The execution trace is depicted in figure 11 ("case" refers to the cases described in 5):

t	$\mathcal{R}(t)$	case	exe.	sc
0	$\{T_1, T_5\}$	3	T_1	unchanged
1	$\{T_1, T_5\}$	3	T_1	unchanged
2	$\{T_5, A_1\}$,	2b	A_1	$sc(I_0)$ decreased
3	$\{T_5, S_1^1, S_2, A_1\}$	2a	S_1^1	$sc(I_0)$ decreased
4	$\{T_5, S_2, A_1\}$	2a	S_2	$sc(I_0)$ decreased
5	$\{T_5, S_2, A_1\}$	2a	S_2	$sc(I_1)$ decreased
6	$\{T_5, A_1\}$	2b	A_1	$sc(I_1)$ decreased
7	$\{T_5\}$	3	T_5	unchanged
8	$\{T_5, S_1^2\}$	2c	T_5	unchanged

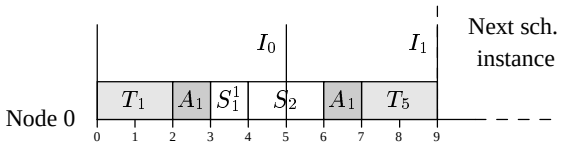


Figure 11. On-line execution on node 0.

7. Conclusion

In this paper, we presented an algorithm to handle event-triggered sporadic tasks, i.e., with unknown arrival times, but known maximum arrival frequencies, in the context of time-triggered, distributed schedules with general constraints. The sporadic tasks are guaranteed during design time, allowing rescheduling or redesign in the case of failure. At run-time, resources reserved for sporadic tasks can be reclaimed and used for efficient aperiodic task handling.

Our algorithm is based on the slot shifting method, which provides for the combination of time-triggered off-line schedule construction and on-line scheduling of aperiodic activities. It analyzes constructed schedules for unused resources and leeway in task executions first. The run-time scheduler uses this information to include aperiodic tasks, shifting other task executions (”slots”) to reduce response times without affecting feasibility. It can also be used to perform online guarantees.

We provided an off-line schedulability test for sporadic tasks based on slot shifting. It constructs a worst case scenario for the arrival of the sporadic task set and tries to guarantee it in the off-line schedule. The guarantee algorithm is applied at selected slots only. At run-time, it uses the slot shifting mechanisms to feasibly schedule sporadic tasks in union with the off-line scheduled periodic tasks, while allowing resources to be reclaimed for aperiodic tasks.

Since the major part of preparations is performed off-line, the involved on-line mechanisms are simple. Furthermore, the reuse of resources allows for high resoure utilization.

Acknowledgements

The authors wish to thank the reviewers for their fruitful comments which helped to improve the quality of the paper. Further thanks go to Jukka Mäki-Turja and Björn Lindberg for their careful reviewing and stimulating discussions.

References

[1] N. Audsley, A. Burns, M. Richardson, and A. Wellings. Deadline monotonic scheduling theory. *WRTP'92. Preprints of the IFAC Workshop. Pergamon Press, U.K*, 1992.

[2] A. Burns, N. Audsley, M. Richardson, and A. Wellings. Hard real-time scheduling: the deadline monotonic approach. *Proc. of the IFAC/IFIP Workshop, UK*, 1992.

[3] G. Buttazzo, G. Lipari, and L. Abeni. A bandwidth reservation algorithm for multi-application systems. *Proc. of the Intl. Conf. on Real-time Computing Systems and Applications, Japan*, 1998.

[4] G. C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[5] M. Caccamo and G. C. Buttazzo. Exploiting skips in periodic tasks for enhancing aperiodic responsiveness. *Proc. of the 18th Real-Time Systems Symposium, USA*, Dec. 1997.

[6] M. Chetto and H. Chetto. Scheduling periodic and sporadic tasks in a real-time system. *Inf. Proc. Letters*, Feb. 1989.

[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 deadlines. *IEEE Trans. on Soft. Eng.*, May 1981.

[10] 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.

[11] 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.

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

[13] A. Mok. *Fundamental Design Problems for the Hard Real-Time Envs.* PhD thesis, MIT, May 1983.

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

[15] M. Spuri, G. C. Buttazzo, and F. Sensini. Robust aperiodic scheduling under dynamic priority systems. *In Proc. of the IEEE RTSS*, Dec. 1995.

[16] 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.

[17] 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.