HANDLING NON-PERIODIC EVENTS TOGETHER WITH COMPLEX CONSTRAINED TASKS IN DISTRIBUTED REAL-TIME SYSTEMS

Radu Dobrin[1], **Gerhard Fohler**[2]

[1] *Department of Computer Engineering Mälardalen University, Västerås, Sweden*
[2] *FB Elektrotechnik und Informationstechnik, Technische Universität Kaiserslautern, Germany*

Abstract: In this paper we show how off-line scheduling and fixed priority scheduling (FPS) can be combined to get the advantages of both - the capability to cope with complex timing constraints while providing run-time flexibility. We present a method to take advantage of the flexibility provided by FPS while guaranteeing complex constraint satisfaction on periodic tasks. We provide mechanisms to include FPS servers to our previous work, to handle non-periodic events, while still fulfilling the original complex constraints on the periodic tasks. In some cases, e.g., when the complex constraints can not be expressed directly by FPS, we split tasks into instances (artifacts) to obtain a new task set with consistent FPS attributes. Our method is optimal in the sense that it keeps the number of artifacts minimized.

Keywords: distributed real-time systems, fixed priority scheduling, off-line scheduling, complex timing constraints

1. INTRODUCTION

Off-line scheduling and fixed priority scheduling (FPS) are often considered as having incompatible paradigms, but complementing properties (Xu and Parnas, 2000). FPS has been widely studied and used in a number of applications, mostly due its simple run-time scheduling, small overhead, and good flexibility for tasks with incompletely known attributes. Modifications to the basic scheme to handle semaphores (Sha *et al.*, 1990), aperiodic tasks (Sprunt *et al.*, 1989), static (Tindell, 1994) and dynamic (Palencia and Harbour, 1998) offsets, and precedence constraints (Harbour and Lehoczky, 1991), have been presented. Temporal analysis of FPS algorithms focuses on providing guarantees that all instances of tasks will finish before their deadlines. The actual start and completion times of execution of tasks, however, are generally not known and depend largely on run-time events, compromising predictability.

Off-line scheduling for time-triggered systems, on the other hand, provides determinism, as all times for task executions are determined and known in advance. In addition, complex constraints can be solved off-line, such as distribution, end-to-end deadlines, precedence, jitter, or instance separation. As all actions have to be planned before startup, run-time flexibility is lacking.

In this work we present methods to combine FPS with off-line schedule construction. A previous work (Dobrin *et al.*, 2001) takes an off-line schedule, constructed to fulfill original complex constraints, and assigns FPS attributes, i.e., priority,

offset, and period, to the tasks, such that their runtime FPS execution matches the off-line schedule. It does so by deriving priority inequalities, which are then resolved by using integer linear programming (ILP). In this paper we extend the previous work to provide for inclusion of existing FPS servers such that non-periodic events can be handled at run-time, while reenacting the original off-line schedule. Thus, it combines FPS runtime flexibility with the capability of off-line scheduling to resolve complex constrained tasks.

Our method is particulary suitable, by modeling faults as non-periodic events, in complex constrained distributed systems with high fault tolerance (FT) requirements where the FT strategy employed is the re-execution of the faulty task.

FPS cannot reconstruct all schedules with periodic tasks and servers with the same priorities for all instances directly. The constraints expressed via the off-line schedule may require that instances of a given set of tasks need to be executed in different order on different occasions. Obviously, there exist no valid FPS priority assignment that can achieve these different orders. Our algorithm detects such situations, and circumvents the problem by splitting a task (or a server) into its instances. Then, the algorithm assigns different priorities to the newly generated "artifact" tasks, the former instances.

Key issues in resolving the priority conflicts are the number of artifact tasks created, and the number of priority levels. Depending on how a priority conflict is resolved, the number of resulting tasks may vary, depending on the periods of the split tasks. Our algorithm minimizes the number of artifact tasks and priority levels. By using an ILP solver for the derivation of priorities, additional demands such as reducing number of preemptions levels can be added by inclusion in the goal function.

Priority assignment for FPS tasks has, for example, been studied in (Audsley, 1991) and (Gerber *et al.*, 1995). (Seto *et al.*, 1998) study the derivation of task attributes to meet overall constraints, e.g., demanded by control performance. In FPS, non-periodic events are commonly handled by servers, e.g., *background scheduling*, *polling server* (Lehoczky *et al.*, 1987; Sprunt *et al.*, 1989) *deferrable servers* (Lehoczky *et al.*, 1987), or *slack stealing* (Lehoczky and Ramos-Thuel, 1992). However, in 1995, Tia et. al. (Tia *et al.*, 1995) proved the non optimality of the existing fixed-priority servers, i.e., no existing approach can minimize the response time of non-periodic tasks while still guaranteeing the feasibility of the periodic ones. A recent paper (Davis and Burns, 2005) provides an exact schedulability test for application tasks associated with servers in hierarchical scheduling systems.

While lower priority non-periodic tasks can be easily scheduled together with periodic tasks with attributes derived by using our previous approach, we propose the inclusion of existing FPS servers to our method to provide non-periodic events a better service than background scheduling. Our goal is to to provide aperiodic tasks a good response time, while not jeopardizing the complex timing constraints of the periodic tasks guaranteed by our transformation algorithm.

The rest of the paper is organized as follows: In Section 2 we briefly describe the previous work. In Section 3 we present our approach to include non-periodic events. We exemplify our method in Section 5 and we discuss several aspects of our approach in Section 4. Section 6 concludes the paper.

2. TASK ATTRIBUTE ASSIGNMENT ALGORITHM

In this section we present an algorithm to transform off-line schedules to attributes for FPS in the absence of non-periodic events, by giving a brief description of the method presented in (Dobrin *et al.*, 2001). The approach determines attributes for tasks assigned to *target windows* and associated chains such that, if executed according to fixed priority scheduling, they will execute inside their target window and obey the order constraint of the task chains. Later on, we will present novel adaptations to incorporate FPS servers to handle run-time, non-periodic tasks.

2.1 Target windows

A target window of a task instance is defined as the interval of time in which the instance will execute and complete at run-time. For example, the target window of a task scheduled by the RM algorithm will be the period of the task. The target window of a task scheduled off-line will consist of the time slots that the off-line scheduler assigned to the task.

2.2 Task and system model

We assume a distributed system where tasks are statically allocated to processors. Tasks $\tau_i \in \{Original\_Tasks\}$ are fully preemptive and off-line scheduled to guarantee complex constraints expressed by target windows TW_i, $i = 1, 2, \ldots$. Each task has a period, $T(\tau_i)$, and a known, upper bound, execution requirement $C(\tau_i)$.

2.3 Algorithm overview

The *input* to our method is a taskset $\tau_i \in \{Original\_Tasks\}$, with complex constraints expressed in:

- Off-line schedule, up to LCM, expressing the original task constraints, that gives off-line scheduled start times, $st(\tau_i^j)$, and finishing times, $ft(\tau_i^j)$, for each instance τ_i^j of each task τ_i
- Target windows, $TW(\tau_i^j)$, representing earliest start times and deadlines for each instance τ_i^j of each task τ_i:

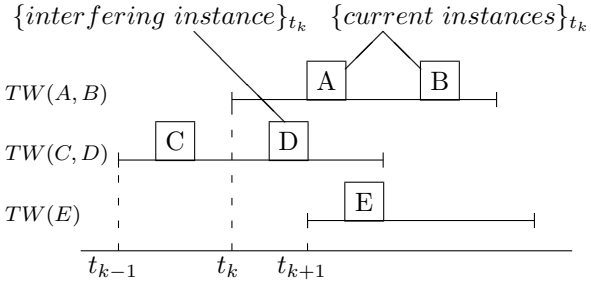
Method core: we analyze overlappings between target windows and derive sequences of instances based on the order of execution specified in the off-line schedule. We translate order constraints into priority constraints between the new FPS tasks. We may not be able to find a FPS schedule with the same number of tasks as the original one, but we may have to create new tasks by splitting some of the original off-line tasks. The resulting number of FPS tasks is to be minimized.

Output: we are looking for a set of tasks, $\overline{\tau_i} \in \{FPS\ Tasks\}$, with priorities, $P(\overline{\tau_i})$, offsets, $O(\overline{\tau_i})$ and periods, $T(\overline{\tau_i})$ such that:

(1) Each instance of each task $\overline{\tau_i}$ will execute at run time inside its target window
(2) The order of execution enforced by the original task constraints is preserved

2.4 Derivation of the inequalities

Given the target windows representing the original constraints and expressed in the off-line schedule, we derive *sequences of tasks* corresponding to the start of each target window(figure 1).



$SEQ_k =< DAB >$

Fig. 1. Sequence of tasks.

A **sequence of tasks** SEQ_k consists of task instances ordered by increasing scheduled start times according to the off-line schedule. A sequence may contain instances τ_i^j of tasks τ_i such that $est(\tau_i^j) = t_k$, referred to as $\{current\ instances\}_{t_k}$, or instances τ_s^q of tasks τ_s from overlapping target windows such that $est(\tau_s^q) < t_k$ and $ft(\tau_s^q) > t_k$, which we refer to as $\{interfering\ instances\}_{t_k}$.

The priority assignment has to preserve the execution order expressed in the off-line schedule. Therefore, from each sequence of tasks $SEQ_k = \overset{k}{S}_1, \overset{k}{S}_2, \ldots, \overset{k}{S}_N$, $N, k = 1, 2, \ldots$, we derive priority relations between the task instances within SEQ_k.

$$P(\overset{k}{S}_1) > P(\overset{k}{S}_2) > \ldots > P(\overset{k}{S}_N) \qquad (1)$$

The priority inequality system derived from the sequences of tasks, includes all task instances in the off-line schedule.

2.5 Attribute assignment - conflicts

Based on the order of execution expressed by the inequalities derived in Section 2.4, we derive attributes - priorities and offsets - for each task.

Our goal is to provide tasks with fixed offsets and fixed priorities. It may happen, however, that we have to assign different offsets/priorities to different instances of the same task, in order to reenact the off-line schedule at run time. In these cases, we split the conflicting task into instances such that, further on, each instance will be considered as an independent task with one instance during LCM.

To find the optimal solution to the problem, we use integer linear programming (ILP). The aim of the given attribute assignment problem is to find a task set, i.e., a minimum number of tasks together with their priorities, that fulfills the priority relations of the sequences of the schedule. The LP solver will provide information about the optimal splits that satisfy the inequalities, and the minimum number of priority levels. For a complete description of the ILP problem formulation, we refer to the results presented in (Dobrin *et al.*, 2001).

2.6 Periods and offsets

Based on the information provided by the LP-solver, we assign periods and offsets to each task in $\overline{\tau_i} \in \{FPS\_Tasks\}$, in order to ensure the run time execution within their respective target windows, as following:

$$T(\overline{\tau_i}) = \frac{LCM}{nr\_of\_instances(\overline{\tau_i})} \qquad (2)$$
$$O(\overline{\tau_i}) = begin(TW(\overline{\tau_i^1}))$$

$$D(\overline{\tau_i}) = end(TW(\overline{\tau_i^1}))$$

3. USING FPS SERVERS TO HANDLE NON-PERIODIC EVENTS

So far, we have shown how to translate off-line schedules to attributes for FPS, i.e., how to exploit the advantages provided by off-line scheduling in FPS. However, one of the main advantages obtained by using FPS is the capability to handle non-periodic events, e.g., aperiodic tasks.

For analysis purposes, we classify the FPS servers in two categories, depending on the behavior of the server upon the presence (or absence) of a non-periodic event: servers that do not preserve their capacity during their period if no aperiodic requests are pending, e.g., background scheduling or polling server, and servers that do, e.g., deferrable server. In the first case, if no aperiodic tasks are waiting to be served at the beginning of the server period, the server capacity is wasted and replenished at the beginning of the next period.

3.1 Motivating example

Ideally, we would like to assign the servers the highest priority to ensure that aperiodic requests are provided a fast service. However, we have to take into consideration the schedulability of the periodic tasks as well, i.e., we can not minimize the reaction time to serve aperiodic requests on the expense of any of the periodic, constrained tasks. Moreover, in our task model we have to deal not only with periodic RM scheduled tasks, but with tasks with complex constraints for which the attribute assignment has been performed according to the previously described method.

Let's assume we have a set of 3 tasks (table 1) and we want to fix the execution times of, e.g., task B at fixed points in time, e.g., an instance separation of 11 time units between the first two instances of B.

Table 1. Original tasks

Task	T	C
A	5	1
B	10	3
C	20	6

However, since B does not have the shortest period, priority assignment according to RM would not guarantee the constraint satisfaction. Instead we use our method and assign the second instance of B an offset equal to 1 (table 2). As an attribute inconsistency occurs between B's instances, we create artifacts for these as shown in table 2.

Table 2. FPS attributes

Task	T	C	P	O	D
A	5	1	2	0	5
B1	20	3	3	0	10
B2	20	3	3	11	20
C	20	6	1	0	20

In this system, if we do not need to handle non-periodic events, the tasks will execute under FPS fulfilling the original constraint on task B (figure 2). Let us now assume we want to be able to

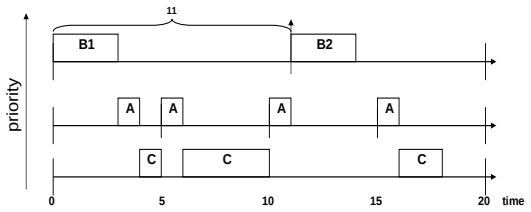


Fig. 2. Motivating example - original task set

handle non-periodic events as well, together with our tasks. The processor utilization in the system is 75%. Thus, there are 5 time units of slack up to LCM (20), if the tasks execute for WCET. We can use, for example, a polling server with a period of 5 and a capacity equal to 1 to distribute the capacity evenly over LCM. Now, if the server is not aware of the constraint on the instances of B, it could have the highest priority and, if there are any aperiodic jobs pending at the start of the server period, they will be served immediately while the periodic tasks will meet their deadlines. However, if an aperiodic task J arrives at time 0, it will be scheduled by the server and the constraint on B's instances will no longer hold (figure 3). In our example, the server was one

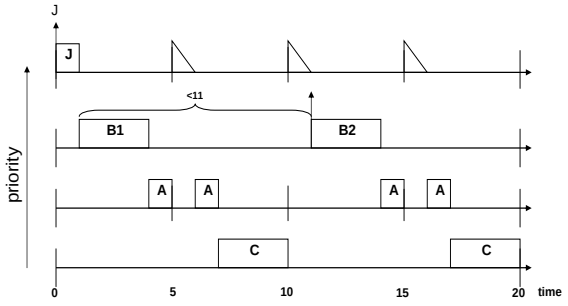


Fig. 3. Motivating example - problem

that does not preserve its capacity. In the case that the chosen server is a capacity-preserving one, e.g., deferrable server, things could get even worse. That is because the server will not become active either at the beginning of its period or not at all, but anywhere within its period. In that case its potential interference interval is drastically increased. The server execution will interfere with the periodic tasks for a period of time not longer than its capacity, but starting anywhere within its period

3.2 Proposed solution - overview

What we can conclude from the example above is that if the server is not aware of the constraints on the periodic tasks, i.e., the server parameter assignment is performed after the attribute assignment to the periodic tasks, the constraints cannot be guaranteed. However, if the server is aware of the original constraints, i.e., the system designer assigns server parameters by taking into consideration the constraint on the periodic tasks, the server execution can be controlled such that its interference with the constrained periodic tasks will not cause any violations to the constraint satisfaction achieved so far.

To do so, we propose to take the server(s) into account together with the constrained period tasks when constructing the off-line schedule. Thus, both complex constrained periodic tasks and server(s) will be used as input to our method.

3.3 Server attribute assignment

To perform the server attribute assignment, by taking into account the available resources in the system, as well as the original constraints on the periodic tasks, we divided the servers in two groups: servers that preserve their capacity during their period, and servers that do not.

Servers that do not preserve their capacity In this case the server will become active at the beginning of its period, if any aperiodic request is pending, and will execute when it will have the highest priority among the tasks in the ready queue. In that sense, the server will behave exactly as a periodic task with a *worst case execution time* equal to its capacity and a *best case execution time equal* to 0, making the procedure of off-line schedule construction for the periodic tasks and server quite trivial. Then, our method takes the off-line schedule and assigns FPS attributes to both tasks and server(s) such that their run-time execution under FPS reenacts the off-line schedule.

In cases of attribute inconsistencies between different instances of the same task, or between different server instances, we create artifacts for the instances to achieve consistent FPS attributes, as described earlier.

Capacity preserving servers In this case, the server can start to serve an aperiodic request anywhere within its period. That makes it difficult to schedule it off-line, i.e., to construct an off-line schedule for both constrained periodic tasks and server while taking full advantage of the server flexibility to handle aperiodic tasks. Instead, we propose to limit the execution scenarios for the server, in order to guarantee the constraints on the periodic tasks, while still providing for flexibility.

Ideally, we would like to have a priority as high as possible on the server, but not on the expense of the periodic tasks and their constraints.

At the same time, the system designer should be aware of the behavior of the server as well, i.e., the mapping of complex constraints, resolved in off-line schedules, to FPS attributes should be performed by taking into account the possibility that the server may start executing at any time during its period.

In figure 4, A and B do not interfere with each other and no priority relation is derived between them. Hence, ILP could assign A a priority higher than B or vice versa, depending on the rest of the priority inequalities. In case A is assigned a priority higher than B and if a deferrable server is executing at a higher priority than A, then the execution of A can be delayed, interfering with B and, thus, B will miss its deadline. On the other hand, if A has been assigned a lower priority than B, the order of execution specified in the off-line schedule will be changed, as A will be preempted by B. A possible solution to this

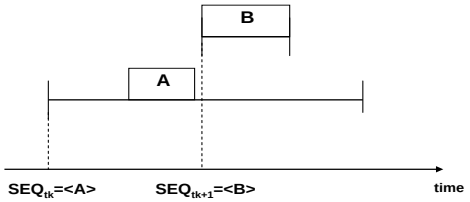


Fig. 4. Sequences in the absence of servers - no priority relation between A and B

issue is to take into account *possible interference* between instances caused by server executions. In our case a new priority inequality, e.g., between A and B, will be added to the IPL formulation. This, however, is a highly pessimistic assumption that will over-constrain the ILP solver. That is, the more priority inequalities we send to ILP, the lower chance we have to obtain a good solution in terms of the final number of FPS tasks and priority levels.

Instead, we propose to assign deadlines to the tasks to reflect the latest time at which the tasks must complete in order to guarantee the original complex constraints and to prevent unexpected additional interference:

$$D(\tau_i^j) = min(end(TW(\tau_i^j)), t_k) \qquad (3)$$

where

$$t_{k-1} < ft(\tau_i^j) \leq t_k$$

and t_k, $k = 1, 2, \ldots$, represent the starts of the target windows in ascending order, $t_1 < t_2 < \ldots$. In our example in figure 4, A will never become an *interfering instance* at time t_{k+1} without missing its *new* deadline (figure 5). Hence, no modifications to the original sequence derivation are required and no extra priority inequalities will be added to the ILP formulation. At this point,

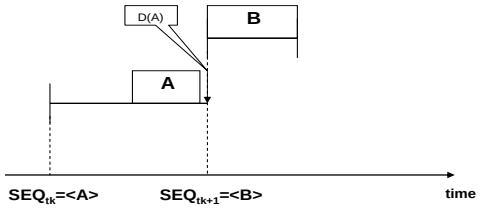


Fig. 5. Deadline assignment to prevent interference in the presence of servers

any feasible attribute assignment can be used on the servers as long as the designer can guarantee the completion of the periodic tasks before their deadlines. That is because, in our off-line to FPS transformation, we specified the release times and deadline of the tasks, i.e., the target windows, in which if the tasks execute and complete, the original constraints are guaranteed. Further more, by controlling the server impact on periodic task executions, the order of execution of the period tasks, specified in the off-line schedule, will be preserved. Thus, the original constraints are guaranteed when scheduling the tasks by FPS.

3.4 Evaluation

Due to the lack of space, we refer to (Dobrin *et al.*, 2001) for the complete proof of correctness for the proposed methodology.

4. DISCUSSION

From the implementation point of view, when we create artifacts for task instances to resolve attribute conflicts, we create only new TCB's with fixed FPS attributes with pointers to the same code and data segments as the original task.

If we use servers that do not preserve their capacities, we can easily include them in the off-line schedule construction and our method will find optimal FPS attributes to them together with the periodic tasks. On the other hand, if the user choose to use capacity-preserving servers, we provide deadlines to the tasks to prevent their execution to interfere with "later" target windows, that would potentially lead to changes in the order of execution specified in the off-line schedule. Then, the server can be assigned any feasible attributes as long as the periodic tasks are guaranteed to complete before their deadlines. Hence, the original complex constraints are guaranteed. Slack stealing (Lehoczky and Ramos-Thuel, 1992; Davis *et al.*, 1993), for example, can be successfully used with our method as it keeps track and uses the available slack in the system without causing any deadline misses among the periodic tasks. Since the task deadlines represent the latest point in time at which periodic tasks can complete their execution such that their complex constraints are fulfilled, slack stealing can be successfully used to handle non-periodic events.

In our method, we specify new deadlines for the task only if we know that a capacity-preserving server is to be used with our tasks. By specifying new deadlines for the tasks, we do not introduce additional splits, as the purpose of the deadline is only to aid the user to assign feasible attributes to the server without jeopardizing the complex constraints guaranteed so far.

In our method we propose to choose server periods and capacities to distribute the exploitations of the amount of slack in the system over LCM. The main advantage of equally distributing the server execution over LCM, i.e., short periods that imply an increased number of server instances up to LCM, is that we increase the chances to provide the aperiodic tasks a prompt service by assigning the server a high priority. In some cases, we may have to create artifacts for the server instances to achieve a high priority setting. This is because the server can have temporary high priorities over short period of times, but not during the entire LCM. If, on the other hand, we choose to assign a server a period equal to LCM and a capacity up to full processor utilization together with the periodic tasks, we may end up in a situation where the server must be assigned the lowest priority, since it interferes with all periodic task instances, ending up with background scheduling.

5. EXAMPLE

We illustrate the ability of the method to deal with periodic tasks with complex constraints togeher with non-periodic ones, with an example.

Table 3. Periodic tasks

Task	T	C	Node
A	15	2	1
B	15	1	1
C	15	5	1
D	10	3	1
E	15	3	2
F	15	4	2

We assume we have a taskset distributed on two nodes as described in Table 3. A first set of

constraints consist of a number of precedence relations as following:

$$A \rightarrow B \rightarrow C; E \rightarrow F; E \rightarrow C$$

and it takes one slot to send a message between 2 nodes. Secondly, we want the FPS execution of task A to be fixed between $(est(A) + 2)$ and $(est(A)+4)$. Additionally, we want to use a polling server on node 1 in the final FPS scheme to be able to handle run-time events as well. As the utilization on node 1 is 83%, we can, for example, use a server with a period of 6 and capacity of 1 to fairly allocate the slack in the schedule over LCM for servicing non periodic tasks. The off-line schedule and the target windows (delimited by the arrows) for tasks and the polling server (S) are illustrated in Figure 6.

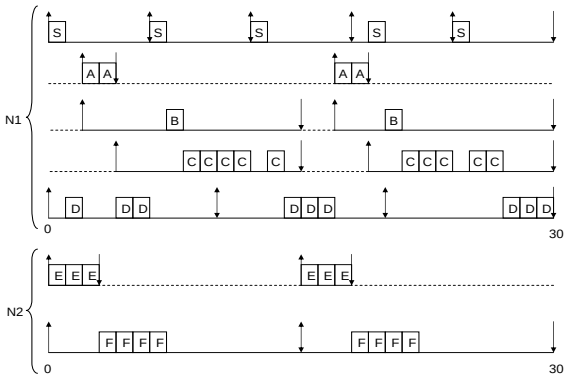


Fig. 6. Off-line schedule and target windows for periodic tasks and one polling server

The priority inequalities between the instances are derived in the same way as in the example presented in Section 2 and shown in Figure 4.

Table 4. Derivation of inequalities

t_k	Node	$\left\{ \begin{matrix} current \\ inst. \end{matrix} \right\}_{t_k}$	$\left\{ \begin{matrix} intf. \\ inst. \end{matrix} \right\}_{t_k}$	SEQ_k	inequalities
0	1	S^1, D^1	None	S^1, D^1	$P(S^1) > P(D^1)$
	2	E^1, F^1	None	E^1, F^1	$P(E^1) > P(F^1)$
2	1	A^1, B^1	D^1	A^1, D^1, B^1	$P(A^1) > P(D^1)$
					$P(D^1) > P(B^1)$
4	1	C^1	B^1, D^1	D^1, B^1, C^1	$P(D^1) > P(B^1)$
					$P(B^1) > P(C^1)$
6	1	S^2	B^1, C^1	S^2, B^1, C^1	$P(S^2) > P(B^1)$
					$P(B^1) > P(C^1)$
10	1	D^2	C^1	C^1, D^2	$P(C^1) > P(D^2)$
12	1	S^3	C^1, D^2	S^3, C^1, D^2	$P(S^3) > P(C^1)$
					$P(C^1) > P(D^2)$
15	2	E^2, F^2	None	E^2, F^2	$P(E^2) > P(F^2)$
17	1	A^2, B^2	None	A^2, B^2	$P(A^2) > P(B^2)$
18	1	S^4	A^2, B^2	A^2, S^4, B^2	$P(A^2) > P(S^4)$
					$P(S^4) > P(B^2)$
19	1	C^2	S^4, B^2	S^4, B^2, C^2	$P(S^4) > P(B^2)$
					$P(B^2) > P(C^2)$
20	1	D^3	B^2, C^2	B^2, C^2, D^3	$P(B^2) > P(C^2)$
					$P(C^2) > P(D^3)$
24	1	S^5	C^2, D^3	S^5, C^2, D^3	$P(S^5) > P(C^2)$
					$P(C^2) > P(D^3)$

Next step is to analyze the overlappings between the target windows and to derive sequences of instances corresponding to each start of each target windows. In our example, these time points are: 0, 2, 4, 6, 10, 12, 15, 17, 18, 19, 20, and 24. The sequences and associated sequences are presented in table 4.

From the inequalities, we can see that we have a number of priority assignment conflicts, i.e., $P(D^1) > P(B^1) > P(C^1)$ resulting from SEQ_3 corresponding to $t_3 = 4$, and $P(C^1) > P(D^2)$ from SEQ_5 corresponding to $t_5 = 10$, meaning that we have a cycle of inequalities consisting of $P(D^1) > P(B^1) > \ldots > P(D^2)$. We formulate the optimization problem with a goal function to minimize the number of artifact tasks, in case any have to be created. In exchange, the LP solver provides us information about which task(s) to split, in this case D, and priorities for the final FPS tasks. Finally we assign offsets and periods to the FPS tasks in order to ensure the execution within their original target windows (Figure 5).

Table 5. FPS tasks and a polling server

τ_i	T	C	O	D	P
S	6	1	0	6	5
A	15	2	2	4	6 (highest)
B	15	1	2	15	3
C	15	5	4	15	2
D1	30	3	0	10	4
D2	30	3	10	20	1
D3	30	3	20	30	1
E	30	2	0	10	2
F	15	3	0	3	1

The schedule obtained by scheduling the final taskset by FPS is illustrated in Figure 7 where, additionally, we have two non-periodic requests, $J1$ and $J2$ at time 5 and 23 respectively. The execution requirements of J1 and J2 are 2 and 1 respectively.

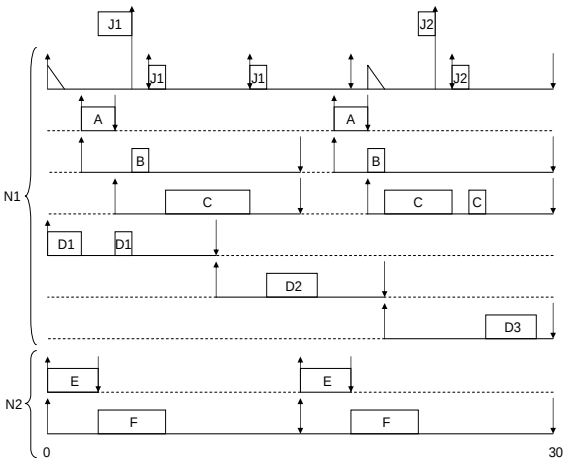


Fig. 7. FPS tasks at runtime

As we can see, at run-time, the tasks will execute flexibly obeying the FPS policy, meeting their original complex constraints, while providing for inclusion of events with incompletely known attributes.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a method that combines off-line schedule construction with fixed priority run-time scheduling. We use off-line schedules and target windows to express complex constraints and predictability, and derive task attributes, such that if applying FPS at run-time, the tasks will execute within the specified target windows and fulfil the original constraints. However, to fully take advantage of the flexibility offered by FPS, we proposed the inclusion of existing FPS servers to our method to provide a good service for non-periodic requests, while guaranteeing the original complex constraints on the periodic tasks.

We have shown that servers that do not preserve their capacity during their periods can be easily incorporated in our method while guaranteeing the constraints of the periodic tasks. In this case, the servers are treated as periodic tasks that do not suspend themselves. The only difference between the server and the periodic tasks is that the server may not execute at all if no aperiodic requests are pending at the beginning of its period.

At the same time, we provide users additional information to be able to handle non-periodic events by using capacity preserving servers. The difficulty of scheduling this type of servers together with periodic, complex constrained tasks, is that the server execution, i.e., at which time the server starts its execution within its period, can not be predicted such that it can be included in the off-line schedule construction. The additional information provided to the users consist in new deadlines to reflect the latest points in time at which periodic tasks must complete to fulfill their complex requirements. At this point, any feasible server attribute assignment can be used, as long as the designer can guarantee the completion of the periodic tasks before their new deadlines. Hence, the original complex constraints expressed in the off-line schedule are guaranteed when scheduling the tasks by FPS.

Future work will include extensions of the work to complex constrained systems with high fault tolerance requirements.

REFERENCES

Audsley, N.C. (1991). Optimal Priority Assignment and Feasibility of Static Priority Tasks With Arbitrary Start Times. Technical report. Departament of Computer Science, University of York.

Davis, R. I. and A. Burns (2005). Hierarchical fixed priority pre-emptive scheduling. In: *Proceedings of the 26th IEEE International Real-Time Systems Symposium*. pp. 389–398.

Davis, R.I., K.W. Tindell and A. Burns (1993). Scheduling slack time in fixed priority preemptive systems. In: *Proceddings of the Real-Time Symposium*. pp. 222–231.

Dobrin, Radu, Gerhard Fohler and Peter Puschner (2001). Translating off-line schedules into task attributes for fixed priority scheduling. In: *Proceedings of the 22nd IEEE Real-Time Systems Symposium*. London, UK.

Gerber, R., S. Hong and M. Saksena (1995). Guaranteeing Real-Time Requirements with Resource-Based Calibration of Periodic Processes. *IEEE Transactions on Software Engineering*.

Harbour, M. Gonzalez and J.P. Lehoczky (1991). Fixed Priority Scheduling of Periodic Task Sets with Varying Execution Priority. In: *Proceedings of Real-Time Systems Symposium*. pp. 116–128.

Lehoczky, J.P. and Sandra Ramos-Thuel (1992). An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In: *Proceddings of the Real-Time Systems Symposium*. pp. 110–123.

Lehoczky, J.P., L. Sha and J.K. Strosnider (1987). Enhanced aperiodic responsiveness in hard real-time environments. In: *Proceddings of the Real-Time Symposium*. pp. 261–270.

Palencia, J.C. and M. Gonzalez Harbour (1998). Schedulability Analysis for Tasks with Static and Dynamic Offsets. In: *Proceedings of 19th IEEE Real-Time Systems Symposium*. pp. 26–37.

Seto, D., J.P. Lehoczky and L. Sha (1998). Task Period Selection and Schedulability in Real-Time Systems. In: *Proceedings of Real-Time Systems Symposium*. pp. 188–198.

Sha, L., R. Rajkumar and J.P. Lehoczky (1990). Priority Inheritance Protocols: an Approach to Real-Time Synchronization. *IEEE Transactions on Computer* **39**(9), 1175–1185.

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

Tia, T., W.S. Liu and M. Shankar (1995). Algorithms and optimality of scheduling aperiodic requests in fixed-priority preemptive systems. *Journal of Real-Time Systems*.

Tindell, K. (1994). Adding Time Offsets to Schedulability Analysis. Technical report. Departament of Computer Science, University of York.

Xu, J. and D. L. Parnas (2000). Priority Scheduling versus Pre-run-time Scheduling. *Real-Time Systems*.