On The Time-Interval Problem

Fábio Rodrigues de la Rocha

[1]Santa Catarina State University
Electrical Engineering Department
Joinville, Brazil

Abstract. *In this paper we present a modeling technique to capture the actual interference among segments in the time-interval problem. Also, we explore the subject of fixed priority assignment for segments through both optimal and sub-optimal algorithms. As a result, we develop a less pessimistic offline feasibility test which results in higher QoS values comparing to previous studies.*

1. Introduction

The problem of scheduling tasks which must finish up to a deadline is an old issue in real-time systems. The deadline for a task is a time-limit to conclude a computation. As long the computation has been concluded before the deadline, the result is timely correct and its finishing-time is not important. Although many applications can be represented by that model, there are some situations in which tasks have special constraints unachieved by periodic task models and mainly by the concept of deadline [Ravindran et al. 2005]. In the time-interval model [de la Rocha and de Oliveira 2006] an earlier/later execution may be useless for application constraints. In that task model, the start of the time-interval is adjusted on-line. Inside this time-interval there is an ideal time-interval where the execution results the highest benefit. The benefit decreases before and after the ideal time-interval according to time-utility functions.

As an example, consider the logic diagram in Figure 1. A task must configure an electronic device (segment A) and perform operations (read, write, etc.) (segment B). However, after the configure process is finished it is necessary to wait for at least t_1 clock pulses before perform operations on that device. The waiting-time is determined online in each task activation and it is related to the operation to be performed on device and in computations carried out in the configuration process. Also, the operations should be performed no later than t_2 clock pulses. In case of sucess, the read/write operations are performed inside a time-interval which results the maximum benefit for that task, otherwise the benefit is lower or null.

In [de la Rocha and de Oliveira 2007] it was presented an offline feasibility test based on response times which besides an accept/reject answer gives a minimum and maximum expected benefit for tasks. Unfortunately, the pessimistic scheduling approach and the simple priority assignment rule for segment B led to high interferences among segments B resulting in high response-times and low benefits. In this paper we return to that scheduling solution paying more attention to the actual interference among tasks (priority assignmentfor subtasks B). Using a more precise interference model, we reduce the pessimism in the offline response-time test. Also, we explore the subject of fixed priority assignment for segments `B` through an optimal and a suboptimal algorithms. As a result we create a new offline feasibility test resulting in higher *QoS* values than the previous approach.

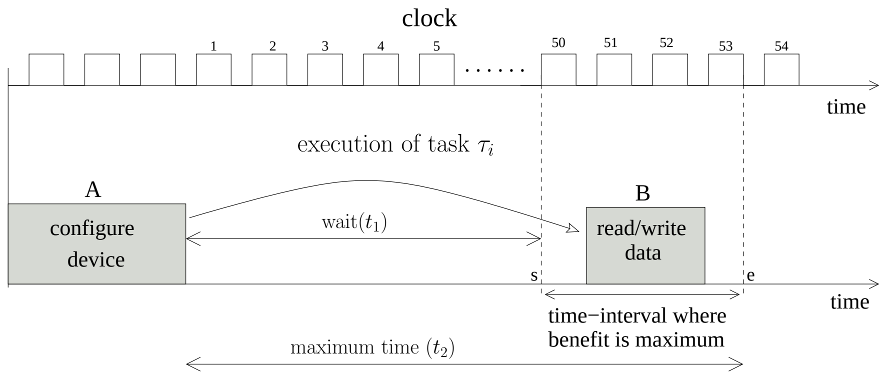


Figure 1. Operations on device.

Organization

This paper is organized as follows. Section 2 presents a brief summary of the time-interval model. Section 3 presents a summary of the scheduling approach and Section 4 presents some experimental evaluations. Finally, we conclude our paper and give an outlook on future work in Section 5.

2. Summary of the Time-Interval Model

The time-interval model is composed by tasks τ_i, $i \in \{1 \ldots n\}$ which are described by a worst-case execution time W_i, period T_i, a deadline D_i ($T_i = D_i$). Task τ_i is composed by three segments named A_i, B_i and C_i. The worst-case execution time of A_i is W_{A_i}, of B_i is W_{B_i} and of C_i is W_{C_i}. The execution of segments follows the order A_i, B_i and C_i and is subject to the deadline D_i. Segment A_i is responsible for performing some computations and may require or not the execution of segment B_i which must perform operations on devices. Segment B_{ij} (where the index j is the activation or job of B_i) is subject to a **time-interval** $[s_{i,j}, e_{i,j}]$ which is defined by segment A_{ij} during run-time and can change for each job $\tau_{i,j}$, i.e: segment B_{ij} must execute inside this time-interval to generate a positive benefit. The length of $[s_{i,j}, e_{i,j}]$ is constant and named ρ_i. Inside the time-interval $[s_{i,j}, e_{i,j}]$, there is an **ideal time-interval** $[ds_{i,j}, de_{i,j}]$ (Figure 2) with constant length named ψ_i where the execution of segment B_{ij} results in the highest benefit to τ_i ($W_{B_i} \leq \psi_i \leq \rho_i$).

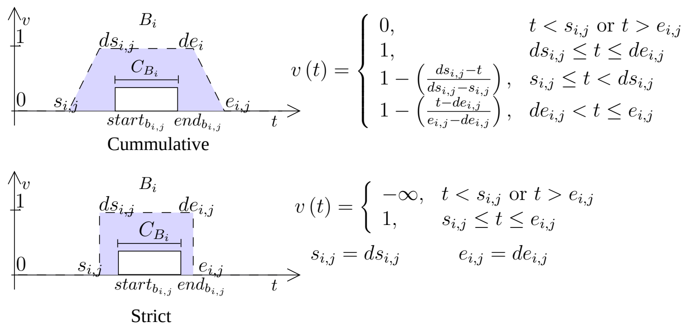


Figure 2. Segment B_i executing inside the ideal time-interval.

We assume two kinds of *QoS* metrics: **Strict** and **Cumulative** (Figure 2). By strict the segment B_i must execute inside the ideal time-interval $[ds_{i,j}, de_{i,j}]$, otherwise the benefit v is $-\infty$, meaning a catastrophic consequence. By cumulative, the benefit decreases from maximum (inside the ideal time-interval) to zero at the time-interval limits. In equation 1 the *QoS* is shown as the cumulative benefit by the execution of segment B_{ij} inside the time-interval. The choice of a particular metric for a task is an application constraint which also determines the values of $s_{i,j}$,$e_{i,j}$,$ds_{i,j}$ and $de_{i,j}$.

$$QoS(B_{i,j}, start_{B_{i,j}}, end_{B_{i,j}}) = \frac{\int_{start_{B_{i,j}}}^{end_{B_{i,j}}} v\,(t)\,dt}{end_{B_{i,j}} - start_{B_{i,j}}} \cdot 100 \quad (1)$$

3. Summary of the Scheduling Approach

For implementation purposes it is useful to map all the segments of task τ_i into subtasks keeping the same names A_i, B_i and C_i. Subtasks A_i and C_i are scheduled using a preemptive *EDF*(Earliest Deadline First) [Layland and Liu 1973] scheduler by its capacity to exploit full processor bandwidth [Buttazzo 2005]. A distinction is made to subtask B_i, which is non-preemptive and scheduled in a fixed priority fashion, higher than A_j and C_j. In a broad sense, when a task τ_i is divided into subtasks each subtask possesses its own deadline and the last subtask has to respect the task's deadline, in this case an end-to-end deadline D_i. Even though the task τ_i has a deadline equal to period ($D_i = T_i$), the subtasks require inner deadlines, which must be assigned using a deadline partition rule. In a simple approach, this rule is determined using the problem constraints. It is assumed a lower bound and an upper bound for the release time of segment B_i $[Bmin_i, Bmax_i]$ and set the deadline $D_{A_i} = Bmin_i$ and $D_{B_i} = Bmax_i + \rho_{B_i}$ as in Figure 3. The time interval in which the segment B_i can be active is $[Bmin_i, D_{B_i}]$ and named **time-window**.

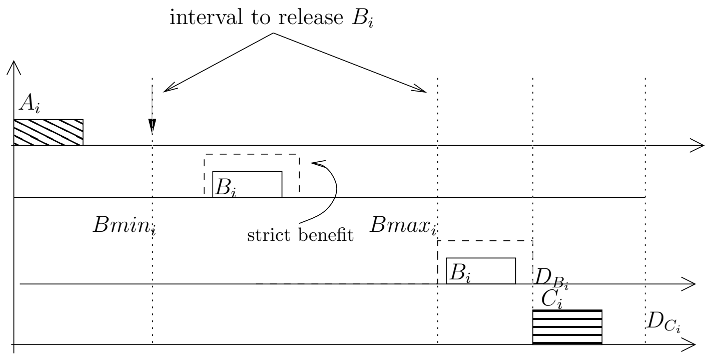


Figure 3. Limits to Release B_i.

3.1. Offline Feasibility Test

The schedulability of a task set τ is verified by splitting the problem in two parts as shown in Figure 4. In the first part we test the schedulability of subtasks A_i and C_i in face of non-preemptive interferences by subtasks B_i. A negative answer (reject) means that all task set is unfeasible. In contrast, a positive answer (accept) means that all subtasks A_i and C_i will finish up to their deadlines even suffering interference by non-preemptive subtasks.

The next part applies a second test based on a response-time to verify if the strict subtasks B_i are schedulable. A negative answer means that all task set is unfeasible. Otherwise, all strict subtasks B_i will execute inside their ideal time-intervals and receive the maximum *QoS*. Using the same response-time test, we determine offline the minimum and maximum *QoS* which can be achieved by all cumulative subtasks.

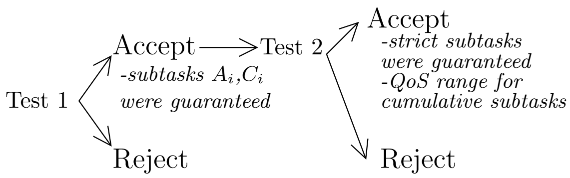


Figure 4. Feasibility Tests.

3.1.1. Feasibility Test for Subtasks A and C

The feasibility of test of subtasks A and C is performed using the processor demand approach [k. Baruah et al. 1990]. The processor demand of a task in a time-interval $[t_1, t_2]$ is the cumulative time necessary to process all k task instances which were released and must be finished inside this time-interval. The schedulability of an asynchronous task set with deadline less than or equal to period can be verified by $\forall t_1, t_2$ $g(t_1, t_2) \leq (t_2 - t_1)$. In asynchronous task sets the schedule must be verified up to $2H + \Phi$ [Leung and Merill 1980] where H is the hyper-period ($H = lcm(T_1, \ldots, T_n)$) and Φ is the largest offset among tasks ($\Phi = max(\Phi_1, \ldots, \Phi_n)$). Hence, the schedulability test must check all busy periods in $[0, 2H + \Phi]$, which has an exponential time complexity $O(H^2)$ [Goossens 1999].

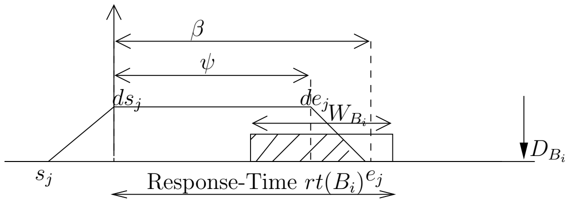
Accounting the Interference of Subtasks B

In [Jeffay and Stone 1993] the authors have shown a schedulability condition to ensure the schedulability of *EDF* in the presence of interrupts. Basically, they assume interrupts as higher priority tasks which preempt every application task. Therefore, they model the interrupt handler interference as a time that is stolen from the application tasks. So, if tasks can finish before their deadlines even suffering the interference from the interrupt handler, the task set is schedulable. The task set is composed by n application tasks and m interrupt handlers. Interrupts are described by a computation time CH and a minimum time between jobs TH. The least upper bound on the amount of time spent executing interrupt handlers in any interval of length L is $f(L)$. Using this method, subtask B_i is modeled as an interrupt handler, subtasks A_i and C_i are implemented as *EDF* subtasks.

3.1.2. Feasibility Test Based on Response-Time

The scheduling approach assigns a priority to each subtask B_i according to some fixed priority assignment algorithm. It is assumed pk priority levels $(1, 2, \ldots, pk)$, where pk

is the lowerest priority. The schedulability of B_i is verified by computing its **response-time** (rt), assuming that all subtasks B_i are always released at ds_j as shown in Figure 5. In the same figure, we use β to describe the time-interval between the release at ds_j up to e_j. In subtasks with cumulative criticality it is possible to finish after the ideal time-interval, resulting in a lower *QoS*. In contrast, subtasks with a strict criticality demand the execution inside the ideal time-interval i.e: it is necessary to verify if in the worst possible scenario $rt(B_i) \leq \psi$. Note that in a strict subtask B_i, $s_j = ds_j$, $de_j = e_j$.



$rt(B_i)$	QoS
$rt \leq \psi$	100%
$rt \geq \beta + W_{B_i}$	0%
$\psi < rt < \beta + W_{B_i}$	$QoS(B_i, rt(B_i) - W_{B_i}, rt(B_i))$%

Figure 5. *QoS* According to the rt.

The response-time can be divided into worst-case response-time ($wcrt$) and best-case response-time ($bcrt$). The $wcrt$ provides the worst possible scenario for the execution of B_i and in this sense the *QoS* is the minimum possible. On the other hand, the $bcrt$ provides the best possible scenario for B_i resulting in the maximum *QoS*.

Computing the $wcrt$ and the $bcrt$ of subtask B_i makes it possible to obtain a *QoS* as shown in Figure 5. Therefore, applying the $wcrt$ of a subtask B_i as a response-time in Figure 5 results in the minimum possible *QoS*. In contrast, applying the $bcrt$ as a response-time results in the maximum possible *QoS*. The first line in the table inside Figure 5 covers the case where all B_i runs inside the ideal time-interval $[ds_j, de_j]$. The second line covers the case where the execution takes place outside the time-interval $[ds_j, e_j]$ (remember that we are now considering all subtasks B_i released at ds_j) and the third line covers the case where part of B_i runs inside the time-interval $[ds_j, e_j]$. In case B_i represents a subtask with strict criticality, $rt(B_i)$ must be $\leq \psi$, otherwise the task set is rejected.

Computing the Response-Time

The best-case response time of non-preemptive sporadic subtasks B_i occurs when B_i does not suffer any interference from other subtasks B_j. As a result, $bcrt_{B_i} = W_{B_i}$. On the other hand, the worst-case response time can be determined by the sum of three terms.

$$wcrt_{B_i} = W_{B_i} + \max_{j \in lp(i)}(W_{B_j}) + \sum_{j \in hp(i)} W_{B_j} \tag{2}$$

The first term in equation 2 is the worst-case execution time of subtask B_i. The second term is the maximum blocking time due to subtasks running at moment B_i is released. We account this value as the maximum execution time among the subtasks B_j with a lower priority (lp) than B_i, leaving the interference of higher priority (hp) subtasks for the next term. The last term is the maximum blocking time due to subtasks B_j with higher priorities. This value adds all subtasks B_j with higher priorities than B_i.

Unfortunately, in some situations the time-windows, in which B_i and B_j can be activate may not overlap. In this case, it is impossible for B_j to produce interference upon B_i, even though it has a higher priority. For instance in :

subtask	W	T	$Bmin$	$Bmax$	D	$Prio$
B_i	2	50	10	20	30	1
B_j	5	50	35	45	55	2

The time-windows do not overlap, so there is no interference between B_j and B_i as shown in Figure 6 item a). However, if we change $Bmin_{B_j} = 15, Bmax_{B_j} = 35, D_{B_j} = 45$ the time-windows overlap and there is interference between B_i and B_j to account as shown in Figure 6 item b). We extend the equation 2 to take into account only

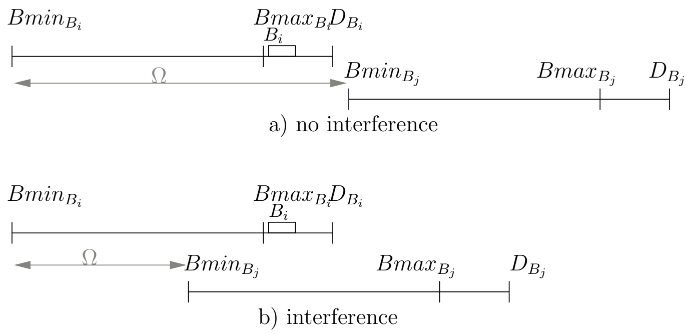


Figure 6. Interference of B_j upon B_i.

the subtasks which produce interference upon B_i(equation 3 and Algorithm 1). The Ω in equation 4 gives the smallest time-length between the earliest release time of B_i and B_j. If Ω is smaller than the interval $[D_{B_i}, Bmin_{B_i}]$, the time-windows overlap resulting in interference accounted as the worst-case execution time of B_j. Although equation 3 results in a smaller $wcrt$ comparing to equation 2, it is still pessimistic in the sense the interference upon B_i is computed assuming that the time-length between B_i and B_j is always the smallest possible.

$$wcrt_{B_i} = W_{B_i} + \max_{j \in lp(i)} (I_{(B_j,B_i)}) + \sum_{j \in hp(i)} I_{(B_j,B_i)} \tag{3}$$

$$\Omega = Bmin_{B_j} - Bmin_{B_i} + \left\lceil \frac{Bmin_{B_i} - Bmin_{B_j}}{gcd(T_{B_i}, T_{B_j})} \right\rceil \cdot gcd(T_{B_i}, T_{B_j}) \tag{4}$$

Priority Assignment for Subtasks *B*

For fixed priorities systems, Rate Monotonic *RM* [Layland and Liu 1973] and Deadline Monotonic *DM* [Leung and Whitehead 1982] are optimal in the sense that no other fixed-priority algorithms can schedule a task set that cannot be scheduled by *RM* and *DM*. The optimality criterion assumes all tasks are synchronous and preemptive. When these assumptions are removed, the *RM* and *DM* are no longer optimal [Audsley 1991].

Algorithm 1 Compute Interference.

Procedure $I(B_i, B_j)$
{`Compute the interference caused by i upon j.`}
$interference \leftarrow 0$
$d \leftarrow \Omega(B_j, B_i)$
if $d < D_{B_j}$ **then**
  if $(d < Bmax_j)$**or**$((prio(i) < prio(j))$**and** $(d \geq Bmax_j))$ **then**
    {`prio(i) < prio(j) In the sense than i has a higher priority than j.`}
    $interference \leftarrow W_{B_i}$
  end if
end if
$d \leftarrow \Omega(B_i, B_j)$
if $d < D_{B_i}$ **then**
  $interference \leftarrow W_{B_i}$
end if
return $interference$
end procedure

Optimal Priority Assignment

In [Audsley 1991] it shows an algorithm with complexity $O(n^2)$ (where n is the number of tasks) to find an optimal priority assignment for a task set. In each step, the algorithm chooses a task to possess the lower priority available and test if the task is schedulable. In case it is not schedulable, it chooses another task and test again. After the algorithm finds a schedulable task, the process is repeated. In that case, an optimal assignment is a priority assignment among tasks which results in a task set where all tasks finish up to their deadlines. Different priority assignments may result in different schedulable task sets. Moreover, as the only concern is the schedulability, different priority assignments may result in optimal assignments.

Differently from [Audsley 1991] where the optimal criterion is the feasibility and priority assignments can result in feasible/unfeasible, in the time-interval problem the optimal criterion is connected to the *QoS* metric. Every priority assignment can result in a different solution and in such case, the only way to find an optimal priority assignment is to enumerate all $n!$ possible priority orderings and pick the one or ones which result in an optimal solution. Unfortunately, to generate all possible priority assignments has $O(n!)$ complexity which is many cases is prohibitive for practical applications.

In the time-interval problem, the minimum release times B_i $\{Bmin_1, \cdots, Bmin_n\}$ characterize the B_i scheduling as an asynchronous system. Subtasks B_i have *QoS* values and in this case, an optimal priority assignment would assign priorities to optimize the global *QoS*. A metric H_{prio} must be used to select among all task sets, the one which is considered probably the best according to some criteria.

A possible criteria is to select the task set with the priority assignment r where the average *QoS* $\overline{x}_{QoS_r}$ is high and the *QoS* of all subtasks present a small dispersion (standard deviation) s_{QoS_r} from the average (equations 5,6).

$$\overline{x}_{QoS_r} = \frac{1}{n}\sum_{i=1}^{n} QoS(B_i) \tag{5}$$

$$s_{QoS_r} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(QoS(B_i) - \overline{x}_{QoS_r}\right)^2} \tag{6}$$

In this particular metric, $H_{prio}(r) \geq H_{prio}(s)$ (in the sense H_{prio} for an priority assignment r is a better solution than H_{prio} for a priority assignment s) if and only if:

$$\begin{gathered}\overline{x}_{QoS_r} \geq \overline{x}_{QoS_s} \text{ and} \\ ((s_{QoS_r} \leq s_{QoS_s}) \text{ or } (s_{QoS_r} \leq \frac{\overline{x}_{QoS_r}}{\overline{x}_{QoS_s}} s_{QoS_s}))\end{gathered} \tag{7}$$

Equation 7 compares the average *QoS* values in both priority assignments r and s. Also, it verifies if the dispersion decreased in r comparing to s or at most increased proportionally to the variation in the average *QoS*. Algorithm 2 presents the optimal solution.

Algorithm 2 Optimal Priority Assignment.

$\{S\} \leftarrow$ all permutations with n priorities
$r \leftarrow$ take one priority assignment from $\{S\}$
$\{S\} \leftarrow \{S\} - r$
while {S} is not empty **do**
    $s \leftarrow$ take one priority assignment from $\{S\}$
    $\{S\} = \{S\} - s$
    if $H_{prio}(r) < H_{prio}(s)$ **then**
        $r \leftarrow s$
    end if
end while
return r

Suboptimal Priority Assignment

Algorithm 3 is a greedy algorithm with complexity $O(n^2)$ to assign priorities to subtasks B_i. The algorithm is suboptimal in the sense it is not guaranteed if it gives the best priority assignment. The algorithm finds a solution picking a subtask q (line 5) with the highest *QoS* (assuming all subtasks in S have higher priorities) to receive the lowerest priority available. Every time a new subtask B_i (q) is chosen, the interferences by higher and lower priorities for all remaining subtasks which receive interference by q are recomputed. The process is repeated until all subtasks have priorities assigned. Subtasks with strict criticality only can be chosen (line 5) when their *QoS* is 100%.

Algorithm 3 Suboptimal Priority Assignment Algorithm.

```
{S} ← all subtasks B_i ∀i ∈ {1...n}
p ← n  {Lowerest priority.}
Compute all interferences(B_i) ∀i ∈ {1...n}
while {S} is not empty do
   q ← choose a subtask B_i with the highest QoS in {S}
   in such way, all B_j have higher priorities
   prio(q) = p
   {Assigns the lowerest priority available.}
   p ← p − 1
   {S} ← {S} − q
   for all subtasks B_i in {S} which interfere with q do
      recompute interferences(B_i)
   end for
end while
```

4. Experimental Evaluation

This section illustrates the proposed feasibility test by comparing its result against a simulation performed on the same task set. In the experiment, the task-set Γ is composed by four tasks (τ_1,τ_2,τ_3,τ_4), each of them subdivided into three subtasks. The worst-case execution times, periods, deadlines, offsets and criticality are presented in Table 1. Tables 2,3,4 and 5 present the steps to assign priorities to subtasks B using Algorithm 3 where $\max$,$\sum$, min *QoS* and prio (priority) represent the maximum interference by lower priority subtasks, the sum of interferences by all higher priority subtasks, the worst-case response-time, the minimum expected *QoS* and the subtask's priority.

Table 1. Example With Four Tasks.

τ	subtask	W_i	D_i	T_i	Φ_i	criticality
	A_1	2	6	40	0	
τ_1	B_1	4	20	40	7	cumulative
	C_1	2	40	40	20	
	A_2	3	9	40	0	
τ_2	B_2	3	31	40	9	strict
	C_2	2	40	40	31	
	A_3	2	25	80	0	
τ_3	B_3	6	38	80	28	cumulative
	C_3	1	80	80	38	
	A_4	3	23	120	0	
τ_4	B_4	6	35	120	23	cumulative
	C_4	3	120	120	35	

The specific parameters of subtasks B such as ρ, ψ, $Bmin$ and $Bmax$ are presented in Table 6. The results of the offline test (using the priorities from Table 5) can be seen in Table 7. The subtask B_2 (with strict criticality) always runs inside the ideal time-interval, resulting in the maximum *QoS*. The other three subtasks have cumulative criticality and present a minimum *QoS* of 87.5%, 11.11% and 16.66% respectively. Due to a pessimistic offline test, the $wcrt$ shown in Table 7 is an upper bound of the rt values. Therefore,

Table 2. Chooses Subtask B_1.

B_i	max	$\sum$	wcrt	min QoS	prio
B_1	0	3	7	87.5	4
B_2	0	16	19	0.0	-
B_3	0	9	15	11.1	-
B_4	0	9	15	16.6	-

Table 3. Chooses Subtask B_4.

B_i	max	$\sum$	wcrt	min QoS	prio
B_1	0	3	7	87.5	4
B_2	4	12	19	0.0	-
B_3	0	9	15	11.1	-
B_4	0	9	15	16.6	3

Table 4. Chooses Subtask B_3.

B_i	max	$\sum$	wcrt	min QoS	prio
B_1	0	3	7	87.5	4
B_2	10	6	15	0.0	-
B_3	6	3	15	11.1	2
B_4	0	9	15	16.6	3

Table 5. Chooses Subtask B_2.

B_i	max	$\sum$	wcrt	min QoS	prio
B_1	0	3	7	87.5	4
B_2	6	0	9	100.0	1
B_3	6	3	15	11.1	2
B_4	0	9	15	16.6	3

we should expect that the actual minimum *QoS* (obtained by simulation) might be higher than (or equal to) the values given by the offline test. In the same way, the $bcrt$ is a lower bound for the rt and the actual maximum *QoS* might be lower than (or equal to) the values given by the offline test.

The task set was simulated for 10.000 time units, assuming the release time uniformly chosen between $Bmin$ and $Bmax$ (Table 8). It is assumed that subtasks B_i and C_i are required in 90% of τ_i activations. The simulation shows a consistent result where the minimum *QoS* values are equal or higher than the values given by the offline test. Thus, the offline test can guarantee that during its execution no task will ever obtain a lower *QoS* than computed by the offline test.

Table 6. Parameters of Subtasks B.

B_i	ρ	ψ	Bmin	Bmax
B_1	8	6	6	13
B_2	9	9	9	23
B_3	14	8	25	27
B_4	10	10	23	27

Table 7. Offline Results.

B_i	wcrt	bcrt	min QoS	max QoS
B_1	7	4	87.5	100.0
B_2	9	3	100.0	100.0
B_3	15	6	11.1	100.0
B_4	15	6	16.6	100.0

Table 8. Simulation Results.

B_i	wcrt	bcrt	min QoS	max QoS
B_1	7	4	87.5	100.0
B_2	6	3	100.0	100.0
B_3	11	6	75.0	100.0
B_4	12	6	66.6	100.0

5. Conclusions and Future Work

This paper makes two contributions to the time-interval problem. The first contribution is an improvement on how interference among B segments are accounted. As a result, we presented a new offline schedulability test which is less pessimistic than the previous one. The second contribution is a study about priority assignment for B segments. We presented an optimal and also a heuristic algorithm to assign priorities to subtasks B which increases the global *QoS*. As a future work, we intend to extend our scheduling approach to deal with preemptive B segments where in spite of its preemptive behavior the access of devices must obey an exclusive access scheme to prevent inconsistences.

References

Audsley, N. (1991). Optimal priority assignment and feasibility of static priority tasks with arbitrary start times - ycs164, dept. computer science, university.

Buttazzo, G. C. (2005). Rate Monotonic vs. EDF: Judgment Day. In *Real-Time Systems*, pages 2–26.

de la Rocha, F. R. and de Oliveira, R. S. (2006). Time-Interval Scheduling and its Applications to Real-Time Systems. In *Proceedings of the* 27^{th} *Real-Time Systems Symposium-WiP*.

de la Rocha, F. R. and de Oliveira, R. S. (2007). Real-Time Scheduling Under Time-Interval Constraints. In *Embedded and Ubiquitous Computing - EUC 2007*, pages 158–169. Lecture Notes in Computer Science.

Goossens, J. (1999). *Scheduling of Hard Real-Time Periodic Systems with Various Kinds of Deadline and Offset Constraints*. PhD thesis, Université Libre de Bruxelles.

Jeffay, K. and Stone, D. L. (1993). Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems. In *Proceedings of the* 14^{th} *IEEE Symposium on Real-Time Systems*, pages 212–221.

k. Baruah, S., Howell, R. R., and Rosier, L. E. (1990). Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor. *Real-Time Systems*, 2:301–324.

Layland, J. and Liu, C. (1973). Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61.

Leung, J. and Merill, M. (1980). A Note on the Preemptive Scheduling of Periodic, Real-Time Tasks. *Information Processing Letters*, 11(3):115–118.

Leung, J. Y. T. and Whitehead, J. (1982). On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, 2:237–250.

Ravindran, B., Jensen, E. D., and Li, P. (2005). On Recent Advances In Time/Utility Function Real-Time Scheduling And Resource Management. In 8^{th} *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 55–60.