Real-Time Scheduling Under Time-Interval Constraints*

Fábio Rodrigues de la Rocha and Rômulo Silva de Oliveira

{frr,romulo}@das.ufsc.br
Graduate Program in Electrical Engineering
Federal University of Santa Catarina, Florianópolis, Brazil

Abstract. This paper presents a new task model where jobs are divided into segments `A`, `B` and `C`. Segment `B` has a specific time-interval where it should execute to fulfill some application constraints. We consider the execution of `B` as valid if performed inside that time-interval, otherwise, its contribution may be valueless to its task. We adapt some scheduling approaches from the literature and present a feasibility test in terms of expected QoS for our scheduling problem.

Key words: Real-Time, Scheduling, Task Model, Time-Interval, QoS

1 Introduction

In most scheduling problems the main interest is to ensure that the task deadline will not be missed. In these cases, the deadlines as well as the periods are embedded constraints in the problem definition with a direct correspondence in physical world. As the years pass by, new task models, scheduling algorithms and feasibility tests were created to expand the algorithmic knowledge available to both the researcher and the system developer. The DM [1], RM [2] and EDF [2] are some well-known algorithms to assign priorities frequently using the periodic task model. In this model, every task τ_i has a fixed period T_i, a worst-case execution time W_i and a relative deadline D_i [2] (in many cases $D_i = T_i$).

The deadline for a task τ_i is a time-limit to τ_i finish its computation. As long as the computation ended before the deadline, the result is timely correct and its finishing time is unimportant. 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 [3]. In some application, tasks demand part of their code to run inside a specific time-interval. The time-interval is a time window inside which the execution must take place and its start time is usually computed online. We present some real-world examples where we can have a time-interval.

① In embedded systems, tasks can send messages using an embedded protocol controller such as i^2c, RS232, USB, CAN. In low cost microcontrollers, during the

* This research was supported by CNPq and CAPES.

data transmission the CPU is kept busy moving data from memory to controller port and waiting for the response/end of transmission. Therefore, both the tasks and the data transmission must be scheduled. Moreover, the data transmission is non-preemptive and sometimes has to be delivered inside a time-interval.

② In ad hoc mobile systems the transmission of packets can be subject to route constraints. Assume that at time t_1 a source device S has a packet to transmit to a destination X. There is a chance that the radio signal from device S cannot reach the destination (there is no feasible route between the source and the destination X at time t_1). In these cases, the packet could be dropped due to a limited buffer space in S. A better solution would schedule the packet transmission to a future time t_2 when there will be a feasible route. However, as the routes dynamically change, time t_2 is only known during run-time.

None of these use cases show a time-limit as the main concern. In fact, they present examples where computations must take place inside a time-interval and maybe inside an inner ideal time-interval where the execution results in the highest benefit. In such cases, the concept of deadline as well as a periodic model are inappropriate to model applications. Unfortunately, by the lack of theoretical study and suitable task models, applications are implemented with conventional schedulers leading to a lack of predictability.

We present a new task model to fulfill a gap in the real-time literature. In our task model, tasks may request that part of their computations execute inside a time-interval to fulfill applications constraints. The start of this time-interval is adjusted on-line and the computations performed before or after the time-interval may be useless for applications purposes. Inside the 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. We integrate and modify some scheduling approaches from the real-time literature in order to obtain a solution for our scheduling problem. As a result, we created an offline feasibility test which provides an accept/reject answer and a minimum and maximum expected benefit for tasks.

Related Work

A classic approach to obtain a precise-time execution is achieved through a time-driven scheduler [4]. However, that approach does not work in face of dynamic changes in task properties [5] such as the start time of our time-interval. The subject of value-based scheduling is studied in many papers. In [6] the authors give an overview of value based-scheduling, their effectiveness to represent adaptive systems and present a framework for value-based scheduling. A study about scheduling in overload situations is presented in [7]. In that paper, tasks have a deadline and also a quality metric. The scheduler performance is evaluated by the cumulative values of all tasks completed by their deadlines and the paper shows that in overload situations scheduling tasks by its value results in better performance. In [8] a task model is composed by tasks with a mandatory and an optional part that increases the benefit as the task executes. However, it is acceptable to execute only the mandatory parts. Also, the optional part is unrelated to a specific time to execute. The Time Utility Function model in which

there is a function to assign a benefit obtained according to the task's completion time is presented in [9] and an extension is presented in [10] with the concept of Joint Utility Function in which an activity utility is specified in terms of other activity's completion time. Differently from these work about task rewarding, our reward criteria is connected to the specific moment the job executes instead of its finish time or the amount of computation performed. An on-line interval scheduling problem in which a set of time-intervals are presented to a scheduling algorithm is presented in [11]. The time-intervals are non-preemptive, have start and end times and cannot be scheduled either early or late. As a future work, the authors discuss a slightly different problem in which the release times are more general and a task could request a given time-interval to be delivered in a determined time as in our problem. The time-interval would be allowed to slide slightly to accommodate other tasks. In the Just in Time Scheduling an earlier execution of a task is as bad as a later execution. The scheduling algorithm tries to minimize the earliness and tardiness (E/T). An overview of E/T problems and a polynomial algorithm for problems with non-execution penalties is presented in [12]. Similarly as in our model, in [13] tasks can adjust during run-time, when the next activation should execute to fulfill some applications constraints. A previous version of our work was presented in [14].

Organization

This paper is organized as follows. Section 2 presents the time-interval model. Section 3 presents a scheduling approach and section 4 presents some experimental evaluations. Section 5 presents the conclusions and future work.

2 Time-Interval Model

We propose a task model in which a task set τ is composed by tasks τ_i, $i \in \{1 \dots n\}$. Tasks τ_i are described by a worst-case execution time W_i, period T_i, a deadline D_i and $T_i = D_i$. Each τ_i consists of an infinite series of jobs $\{\tau_{i1},\dots,\tau_{ij},\dots\}$, the jth such job τ_{ij} is ready at time $(j-1) \cdot T_i, j \geq 1$ and must be completed by time $(j-1) \cdot T_i + D_i$ or a timing fault will occur. We define by **segment** a sequential group of instructions inside τ_i (as shown in Fig. 1). Task τ_i is composed by three segments named A_i, B_i and C_i. We denote the first index of a segment as the task and the second index as the job, thus the first job of segment A_i is named A_{i1}, the second job is A_{i2} and so on for all segments. 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 sum of the worst-case execution time of all segments is equal to the worst-case execution time of task τ_i (W_{A_i}+W_{B_i}+W_{C_i}=W_i). We assume that there is a precedence relation among segments $A_i \prec B_i \prec C_i$.

The execution of segments A_i, B_i and C_i is subject to the deadline of task τ_i, D_i which in this sense is an end-to-end deadline. Segment A_i is responsible for performing its computations and it may require or not the execution of segment B_i. Hence, the arrival time of segment B_i is determined on-line by segment A_i. In case segment B_i is required to execute, segment C_i (which is a **housekeeping code**) will also execute. Therefore, even though the execution of segment A_i is

periodic with period T_i, segments B_i and C_i are sporadic. In case neither B_i nor C_i are required to execute, segment A_i can execute up to the deadline D_i. Otherwise, as soon as segment B_i concludes, segment C_i is released to run. As we consider an uniprocessor system, segments cannot overlap in time.

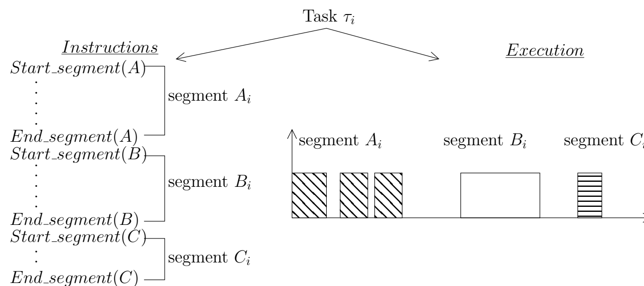


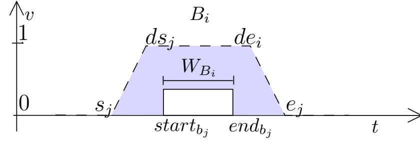
Fig. 1. Task τ_i with Segments

2.1 QoS Metric

The execution of segment B_{ij} is also 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** $[ds_{i,j}, de_{i,j}]$ with constant length named ψ_i where the execution of segment B_{ij} results in the highest benefit to τ_i ($W_{B_i} \leq \psi_i \leq \rho_i$).

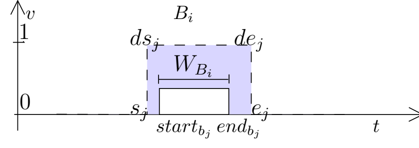
The functions in Fig. 2 and 3 were made to represent ordinary applications requirements and so they also represent different real-time constraints. Figure 3 represents a **strict** benefit where the segment B_i must execute inside the ideal time-interval $[ds_j, de_j]$, otherwise the benefit is $-\infty$, meaning a catastrophic consequence. Figure 2 represents a **cumulative** benefit where the benefit decreases from maximum (inside the ideal time-interval) to zero at the time-interval limits. The choice of a particular function for a task is an application constraint which also determines the values of $s_{i,j}$,$e_{i,j}$,$ds_{i,j}$ and $de_{i,j}$.

In those figures, the y axis represents the benefit v and the x axis is the activate time t. The segment B_{ij} is presented as running with its worst-case execution time (W_{B_i}), starting at $start_{bj}$ and ending at end_{bj}. The benefit $v(t)$ as a function of time is given by the equations in each figure. Equation 1 shows the QoS value as the cumulative benefit by the execution of segment B_{ij} inside the time-interval. The range in $[0\%, 100\%]$ represents the percentage of the maximum benefit. The maximum benefit is only achieved when B_i runs all its code inside the **ideal time-interval** $[ds_{i,j}, de_{i,j}]$. The goal is to maximize the QoS for each job B_i. In case B_i is not required there is no QoS value to account.



$$v(t) = \begin{cases} 0, & t < s_j \text{ or } t > e_j \\ 1, & ds_j \leq t \leq de_j \\ 1 - \left(\frac{ds_j - t}{ds_j - s_j}\right), & s_j \leq t < ds_j \\ 1 - \left(\frac{t - de_j}{e_j - de_j}\right), & de_j < t \leq e_j \end{cases}$$

Fig. 2. Cumulative Benefit



$$v(t) = \begin{cases} -\infty, & t < ds_j \text{ or } t > de_j \\ 1, & ds_j \leq t \leq de_j \end{cases}$$

$$e_j = de_j$$

$$s_j = ds_j$$

Fig. 3. Strict Benefit

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

Besides its time constraints, the time-interval problem may also present constraints regarding exclusive access during segment B execution inside the time-interval. The nature of the resource under segment B control imposes access constraints to ensure the consistence during resource's operation. We assume that during the execution inside the ideal time-interval the CPU is busy controlling the resources. In this paper we consider segment B as non-preemptive, which fulfills the access constraints. Therefore, the execution of B_i from task τ_i cannot be preempted by other task τ_j. The start of B_i may be postponed, however, once started it cannot be disturbed.

3 Scheduling Approach

For implementation purposes, it is natural to represent the segments in our model as a set of subtasks. Therefore, we 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 scheduler by its capacity to exploit full processor bandwidth. A distinction is made to subtask B_i, which is non-preemptive and scheduled in a fixed priority fashion. 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.

The first challenge in our scheduling approach is the **deadline partition** among subtasks. The time-interval definition states that segment B_i must start adjusted by A_i. The minimum time to release B_i is a problem constraint and this value may be used as a deadline for the previous segment. Therefore, in the time-interval problem the deadline partition rule is determined using the problem constraints. We assume 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 Fig. 4. The time interval in which the segment B_i can be active is $[Bmin_i, D_{B_i}]$ and named **time-window**. The second challenge is the **release time** of subtasks. It is necessary to enforce in the schedulability test that a subtask must be release only after a predetermined time. In this situation, we apply offsets [15] to control the release of subtasks and an offset oriented test to fulfill the requirement. The third challenge is the **non-preemptive** aspect of subtask B_i. The feasibility test must ensure that the execution of the non-preemptive subtask cannot be preempted by any other subtask. In this aspect, in [16] it is proposed a schedulability test to verify the schedulability of tasks in the presence of non-preemptive tasks with the higher priorities.

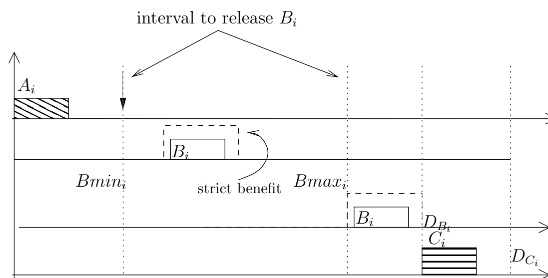


Fig. 4. Limits to Release B_i

3.1 Offline Feasibility Test

We verify the schedulability of a task set τ splitting the problem in two parts. 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 though suffering interference by non-preemptive subtasks. The second part applies a test based on response-time to verify if the strict subtasks B_i are schedulable. A negative answer means all task set is unfeasible. Otherwise, all strict subtasks B_i will execute inside their ideal time-intervals and receive the maximum QoS value. Using the same response-time test, we determine the minimum and maximum QoS value which can be achieved by all cumulative subtasks B_i.

Feasibility Test for Subtasks *A* and *C* The feasibility of test of subtasks A and C is performed using the processor demand approach [17]. 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. We assume $g_i(t_1, t_2)$ as the processing time of τ_i.

The processor demand approach works by observing that the amount of processing time requested in $[t_1, t_2]$ must be less than or equal to the length of the time-interval. Therefore, $\forall t_1, t_2 \; g(t_1, t_2) \leq (t_2 - t_1)$.

Lets assume a function $\eta_i(t_1, t_2)$ as the number of jobs of task τ_i with release and deadline inside $[t_1, t_2]$. $\eta_i(t_1, t_2) = max\{0, \lfloor \frac{t_2 + T_i - D_i - \Phi_i}{T_i} \rfloor - \lceil \frac{t_1 - \Phi_i}{T_i} \rceil\}$. Where T_i is the period of task i, Φ_i is the offset (phase) of task i and D_i is the deadline of task i. In Fig. 5 the only jobs accounted by η_i are jobs $\tau_{i,2}$ and $\tau_{i,3}$. Job $\tau_{i,1}$ has a release time before t_1 and $\tau_{i,4}$ has a deadline after t_2.

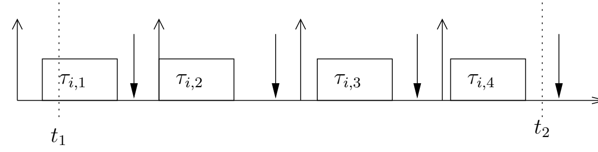


Fig. 5. Jobs of τ_i

The processor demand inside the time-interval is equal to the number of jobs which were activated and must be completed inside the time-interval multiplied by the computation time W_i. Therefore, $g_i(t_1, t_2) = max\{0, \lfloor \frac{t_2 + T_i - D_i - \Phi_i}{T_i} \rfloor - \lceil \frac{t_1 - \Phi_i}{T_i} \rceil\} W_i$ and the processing demand for all task set is :

$$g(t_1, t_2) = \sum_{i=1}^{n} max \left\{ 0, \left\lfloor \frac{t_2 + T_i - D_i - \Phi_i}{T_i} \right\rfloor - \left\lceil \frac{t_1 - \Phi_i}{T_i} \right\rceil \right\} W_i \qquad (2)$$

The schedulability of an asynchronous task set with deadline less than or equal to period can be verified by $\forall t_1, t_2 \quad g(t_1, t_2) \leq (t_2 - t_1)$. In asynchronous task sets the schedule must be verified up to $2H + \Phi$ [18] where H is the hyperperiod ($H = lcm\{T_1, T_2, \ldots, T_n\}$) and Φ is the largest offset among tasks ($\Phi = max\{\Phi_1, \Phi_2, \ldots, \Phi_n\}$). Hence, the schedulability test must check all busy periods in $[0, 2H + \Phi]$, which has an exponential time complexity $O(H^2)$ [19].

Accounting the Interference of Subtasks B

In [16] Jeffay and Stone 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 the interrupt handler interference is considered 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 processing time for execute interrupts is $f(L)$.

Theorem 1. *A set* τ *of* n *periodic or sporadic tasks and a set* ι *of* m *interrupt handlers is schedulable by EDF if and only if*

$$\forall L \geq 0 \qquad g(0, L) \leq L - f(L)$$

where the upper bound $f(L)$ is computed by:

$f(0) = 0$

$$f\left(L\right) = \begin{cases} f(L-1)+1, & if \sum_{i=1}^{m} \left\lceil \frac{L}{TH_i} \right\rceil CH_i > f(L-1) \\ f(L-1), & otherwise \end{cases} \tag{3}$$

The proof is similar to the proof in [17]. The difference is that in any interval of length L, the amount of time that the processor can dedicate to the application tasks is equal to $L - f(L)$.

Using this method, subtask B_i is modeled as an interrupt handler, subtasks A_i and C_i are implemented as EDF subtasks and the feasibility checked using Theorem 1. The Theorem to account for interrupt handlers as expressed by Jeffay and Stone assumes a synchronous task set with deadlines equal to periods.

We extend this Theorem using the processor demand approach to represent asynchronous systems and deadlines less than periods. In this case, subtasks A_i arrives at time zero ($\Phi_{A_i} = 0$) and C_i arrives at time Φ_{C_i}. We assume that at a specific time an interrupt starts B_i (using the designation in [16]). To ensure subtask C_i runs only after subtask B_i, subtask C_i has the offset set to $\Phi_{C_i} = D_{B_i}$. We assume $F(t_1, t_2)$ as the processor demand due to interrupts in $[t_1, t_2]$. The new feasibility test in which all subtasks C_i have offsets and subtasks B_i are modeled as interrupts is: $\forall L \geq 0 \qquad g(t_1, t_2) \leq (t_2 - t_1) - F(t_1, t_2)$

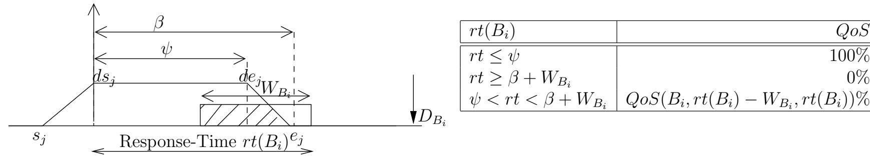
Subtasks B have a time-window during which can be active. So, applying [16] is pessimistic due to the accounting of interrupts where they cannot execute. An improvement is obtained by inserting an offset ΦH_i as in $\sum_{i=1}^{m} \lceil \frac{L - \Phi H_i}{TH_i} \rceil CH_i$ to represent the fact an interrupt cannot happen before $Bmin$. Algorithm 1 has complexity $O(H^2)$. Unfortunately, in the worst case (where all periods are prime numbers) the hyper-period is the product of all periods $\prod_{i=1}^{n} T_i$. Therefore, in practical situations the algorithm can be applied only when task periods result in a small hyper-period.

Algorithm 1 Feasibility test - first test

for all t_1 such that $0 \leq t_1 \leq 2H + \Phi$ **do**
  for all t_2 such that $t_1 \leq t_2 \leq 2H + \Phi$ **do**
    $g(t_1, t_2) = \sum_{i=1}^{n} max\{0, \lfloor \frac{t_2 + T_i - D_i - \Phi_i}{T_i} \rfloor - \lceil \frac{t_1 - \Phi_i}{T_i} \rceil\} W_i$
    $F(t_1, t_2) = f(t_2) - f(t_1)$
    if $g(t_1, t_2) > (t_2 - t_1) - F(t_1, t_2)$ **then**
      return nonfeasible
    end if
  end for
end for
{It is feasible. Apply the second test}
return feasible

Feasibility Test Based on Response-Time Differently from subtasks A_i and C_i which are scheduled by EDF, our scheduling approach assigns a **fixed** priority to all subtasks B_i according to a heuristic rule. The heuristic rule has two metrics, **criticality** and **slide factor**. In the first metric, tasks can be strict or cumulative. Subtasks with the strict criticality correspond to a group of subtasks with higher priorities than subtasks with cumulative criticality. Inside each group, priorities are given inversely proportional to the sliding factor computed as $sf_i = \frac{\psi}{W_{B_i}}$. The sliding factor is related to the capacity to slide inside the ideal time-interval and obtain the highest QoS value.

We intend to verify the schedulability of B_i computing its response-time (rt), assuming that all subtasks B_i are always released at ds_j as shown in Fig. 6. 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 (as shown in Fig. 2) it is possible to finish after the ideal time-interval, resulting in a lower QoS value. In contrast, subtasks with a strict criticality (Fig. 3) 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%
$\psi < rt < \beta + W_{B_i}$	$QoS(B_i, rt(B_i) - W_{B_i}, rt(B_i))$%

Fig. 6. QoS Value 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 value is the minimum possible. On the other hand, the $bcrt$ provides the best possible scenario for B_i resulting in the maximum QoS value. Computing the $wcrt$ and the $bcrt$ of subtask B_i makes it is possible to obtain a QoS value as shown in Fig. 6. Therefore, applying the $wcrt$ of a subtask B_i as a response-time in Fig. 6 results in the minimum possible QoS value. In contrast, applying the $bcrt$ as a response-time results in the maximum possible QoS value. The first line in the table inside Fig. 6 covers the case where all B_i runs inside the ideal time-interval. The second line covers the case where the execution takes place outside the time-interval and the third line covers the case where part of B_i runs inside the time-interval.

Computing the Response-Time

The worst-case response time of non-preemptive sporadic subtasks can be determined by the sum of three terms.

$$wcrt_{B_i} = W_{B_i} + \max_{j \in lp(i)}(W_{B_j}) + \sum_{j \in hp(i)} W_{B_j} \tag{4}$$

The first term in (4) 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. We account this value adding 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 B_i ($W = 2$, $T = 50$, $Bmin = 10$, $Bmax = 20$, $D = 30$, $prio = 1$) and B_j ($W = 5$, $T = 50$, $Bmin = 35$, $Bmax = 45$, $D = 55$, $prio = 2$). The time-windows do not overlap, so there is no interference between B_j and B_i (Fig. 7 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 (Fig. 7 item b).

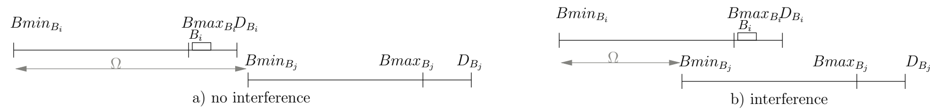


Fig. 7. Interference of B_j upon B_i

We extend (4) to take into account only the subtasks which produce interference (represented as I) upon B_i (5). The Ω in (6) gives the smallest time-length between the earliest release time of B_i and B_j. If Ω is smaller than the distance between $[D_{B_i}, Bmin_{B_i}]$, the time-windows overlap resulting in interference accounted as the worst-case execution time of B_j.

$$wcrt_{B_i} = W_{B_i} + \max_{j \in lp(i)} (I_{(B_j,B_i)} \cdot W_{B_j}) + \sum_{j \in hp(i)} I_{(B_i,B_j)} \cdot W_{B_j} \quad (5)$$

$$I_{(B_i,B_j)} = \begin{cases} 1, & if(\Omega < (D_{B_i} - Bmin_{B_i})) \\ 0, & otherwise \end{cases}$$

$$\Omega = Bmin_{B_j} - Bmin_{B_i} + \left\lceil \left(\frac{Bmin_{B_i} - Bmin_{B_j}}{gcd(T_{B_i}, T_{B_j})} \right) \right\rceil \cdot gcd(T_{B_i}, T_{B_j}) \quad (6)$$

The best-case response time for subtasks B_i occurs when B_i does not suffer any interference from other subtasks B_j. As a result, $bcrt_{B_i} = W_{B_i}$.

4 Experimental Evaluation

In this section we illustrate the effectiveness of the proposed feasibility test comparing its result with a simulation performed on the same task set.

Our experiment is composed by three tasks (τ_1,τ_2,τ_3), each of them subdivided into three subtasks. The worst-case execution times, periods, deadlines

and offsets are presented in Table 1. Also, it show the specific parameters of subtasks B such as criticality, priority, ρ,ψ,$Bmin$ and $Bmax$. The results of the offline test can be seen in Table 2. The subtask B_2 (with strict criticality) always runs inside the ideal time-interval, resulting in the maximum QoS. The other two subtasks have cumulative criticality and present a minimum QoS of 41.6% and 25.0% respectively. Due to a pessimistic test, the *wcrt* shown in Table 2 is an overestimation of the actual values. Therefore, we should expect that the actual minimum QoS might be higher than the values given by the offline test.

Table 1. Example With Three Tasks

τ	subtask	W_i	D_i	T_i	Φ_i	criticality	prio	ρ	ψ	Bmin	Bmax
	A_1	4	10	40	0						
τ_1	B_1	6	31	40	11	cumulative	3	12	10	10	20
	C_1	2	40	40	31						
	A_2	3	20	40	0						
τ_2	B_2	2	34	40	20	strict	1	8	8	20	26
	C_2	2	40	40	34						
	A_3	2	15	60	0						
τ_3	B_3	6	31	60	18	cumulative	2	14	8	15	20
	C_3	1	60	60	31						

Table 2. Offline Feasibility Results

subtask	wcrt	bcrt	min QoS	max QoS
B_1	14	6	41.6%	100.0%
B_2	8	2	100.0%	100.0%
B_3	14	6	25.00%	100.0%

Table 3. Simulation Results

subtask	wcrt	bcrt	min QoS	max QoS
B_1	14	6	41.6%	100.0%
B_2	7	2	100.0%	100.0%
B_3	13	6	41.6%	100.0%

We simulate the same task set for 10.000 time units, assuming the release time uniformly chosen between $Bmin$ and $Bmax$ (Table 3). 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 using the offline test.

5 Conclusions and Future Work

This paper presented a useful model for a class of real-world problems which by the lack of theoretical study, are implemented with conventional schedulers resulting in lack of predictability. We applied some approaches from the real-time literature with adaptations to our scheduling problem to create an offline test

which providing an accept/reject answer for tasks with critical benefit constraints and a minimum/maximum expected benefit for non-critical tasks. As a future work, we intend to investigate the scheduling problem where the B segments are preemptive and need exclusive access upon resources.

References

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

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

[3] Ravindran, B., Jensen, E.D., Li, P.: On Recent Advances In Time/Utility Function Real-Time Scheduling And Resource Management. In: 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. (2005)

[4] Locke, C.D.: Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives. Real-Time Systems **4** (1992) 37–53

[5] Tokuda, H., Wendorf, J.W., Wan, H.: Implementation of a Time-Driven Scheduler for Real-Time Operating Systems. In: 8th RTSS. (1987) 271–280

[6] Burns, A., Prasad, D., Bondavalli, A., Giandomenico, F.D., Ramamritham, K., Stankovic, J., Strigini, L.: The Meaning and Role of Value in Scheduling Flexible Real-Time Systems. Journal of Systems Architecture **46** (2000) 305–325

[7] Buttazzo, G.C., Spuri, M., Sensini, F.: Value vs. Deadline Scheduling in Overload Conditions. In: 16th IEEE Real-Time Systems Symposium. (1995) 90–99

[8] Liu, J., Shih, W.K., Lin, K.J., Bettati, R., Chung, J.Y.: Imprecise Computations. In: Proceeding of the IEEE. (1994)

[9] Jensen, E., Locke, C., Tokuda, H.: A Time-Driven Scheduling Model for Real-Time Operating Systems. In: Real-Time Sytems Symposium. (1985)

[10] Wu, H., Ravindran, B., Jensen, E.D., Balli, U.: Utility Accrual Scheduling under Arbitrary Time/Utility Functions and Multi-unit Resource Constraints. In: Proceedings of the 10th RTCSA. (2004)

[11] Lipton, R.J., Tomkins, A.: Online Interval Scheduling. In: 5th annual ACM-SIAM Symposium on Discrete Algorithms, Philadelphia, PA, USA (1994) 302–311

[12] Hassin, R., Shani, M.: Machine Scheduling with Earliness, Tardiness and Non-Execution Penalties. Computers and Operations Research **32** (2005) 683–705

[13] Velasco, M., Martí, P., Fuertes, J.M.: The Self Triggered Task Model for Real-Time Control Systems. In: 24th RTSS (WiP). (2003)

[14] de la Rocha, F.R., de Oliveira, R.S.: Time-Interval Scheduling and its Applications to Real-Time Systems. In: 27th Real-Time Systems Symposium (WiP). (2006)

[15] Gutierrez, J.P., Harbour, M.G.: Offset-Based Response Time Analysis of Distributed Systems Scheduled under EDF. In: 15th ECRTS. (2003)

[16] Jeffay, K., Stone, D.L.: Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems. In: 14th RTSS. (1993)

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

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

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