Uniprocessor Scheduling Under Time-Interval Constraints

Fábio Rodrigues de la Rocha Rômulo Silva de Oliveira Carlos Montez
Graduate Program in Electrical Engineering - PGEEL
Federal University of Santa Catarina - UFSC
Florianópolis, Brazil
Email: {frr,romulo,montez}@das.ufsc.br

Abstract

This paper proposes a new task model for expressing timing constraints that do not naturally admit expression in terms of deadlines and periods. In our task model, jobs are divided into segments `A`*,* `B` *and* `C` *which must execute following this order. Segment* `A` *is responsible for performing its computations and compute a time-interval wherein segment* `B` *should execute to fulfill some application constraints. Segment* `C` *is released after segment* `B` *has finished. We consider the execution of* `B` *as valid if performed inside that time-interval, otherwise, its contribution may be considered valueless to its task. The model uses benefit functions to express when a given action should be performed for the maximum benefit. We adapt some scheduling approaches from the literature and present a feasibility test for our scheduling problem.*

1. Introduction

The problem of scheduling tasks which must finish up to a deadline is an old issue in real-time systems. 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* [20], *RM* [18] and *EDF* [18] 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 [18] (albeit in some cases it is assumed $D_i = T_i$).

The meaning of a deadline for a task τ_i is a time-limit to τ_i finish its computation. As long as its computation has finished before the deadline, the result is always correct (at least timely correct) and its finish time does not matter. The existence of many applications such as data acquisition, control and actuation of mechanical components which fit in this description brought a tremendous success to that model. A common scenario is an embedded system composed by a single processor that cyclically and concurrently interacts with the external environment acquiring data from sensors and controlling devices.

Although many applications can be suitably 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 [27]. In some application classes, 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 during run-time. We present some real-world use cases, mainly connected to data transmission in embedded systems.

① In embedded systems, tasks can send messages using an embedded protocol controller such as i^2c, *RS232*, *USB*, *CAN*. In many low cost microcontrollers, during the data transmission the *CPU* is kept busy moving data from memory to controller port and waiting for the response or for the finishing of the transmission. Therefore, both the tasks and the data transmission must be scheduled. Moreover, the data transmission cannot be preempted 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.

③ In high speed networks with resource reservation, there are some applications such as video on demand that need a dedicated connection between nodes (the well-known call control problem) [1]. This connection has a time during which the resource is in use and also has a starting and an ending time. The scheduling algorithms are on-line and the jobs (transmission of a message) are considered non-preemptive.

Clearly, 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 con-

cept 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.

In this paper 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 besides an accept/reject answer gives a minimum and maximum expected benefit for tasks.

For implementation purposes, our scheduling approach is built using the *EDF* due to its capacity to exploit full processor bandwidth [6]. For preemptive systems with dynamic priorities, *EDF* is optimum in the sense that if a task set is feasible it can be scheduled using *EDF*. A previous version of our model was presented in [10].

1.1 Related Work

A classic approach to obtain a precise-time execution is achieved through a time-driven scheduler [24]. However, that approach does not work in face of dynamic changes in task properties [29] such as the start time of our time-interval. The subject of value-based scheduling is studied in many papers. In [5] 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. A slight different approach is presented in the classical paper [23] where a task model is composed by tasks with a mandatory and an optional part that increases the benefit as the task executes. However, in that model it is acceptable to execute only the mandatory parts, also, the optional part is unrelated to a specific time, within it must execute. A special case of imprecise computation where the reward increases as the task executes until its deadline is shown in [11]. 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 [16]. An extension of that work is presented in [31] with the concept of Joint Utility Function in which an activity utility is specified in terms of other activity's completion time. Also, it represents the activity's utility as a function of its progress. A heuristic scheduling algorithm to schedule tasks with arbitrary time-utility shapes is presented in [21]. A potential utility density metric gives an expected benefit for executing a task and all other tasks it depends upon. In [8] the authors provide an overview about scheduling tasks described by time value functions to maximize the benefit. Even though they present a general introduction about the topic, their model is limited to non-decreasing time-value functions using non-preemptive *EDF*. 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.

A good example of on-line scheduling appears in the call control problem [2] where a sequence of requests is dynamically made to allocate a virtual circuit between two nodes in a network. The request is composed by the start and the end times of that circuit. For applications such as video and audio transmission the network has to guarantee a minimum bit rate between the nodes. The aim is to maximize the number of accepted calls using an on-line acceptance algorithm. An on-line interval scheduling problem in which a set of time-intervals are presented to a scheduling algorithm is presented in [22]. 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 where a task could request a given time-interval to be delivered within "x" time units 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. In [3] the authors provide a comprehensive review of the *JIT* literature up to that date. An overview of earliness-tardiness problems and a polynomial algorithm for E/T problems with non-execution penalties is presented in [14]. In [25] the authors present a heuristic algorithm to schedule jobs minimizing the total earliness and tardiness penalties in non-preemptive jobs.

Similarly as in our model, in [30] tasks can inform during run-time, when the next activation should execute to fulfill some applications constraints, in that case control systems applications. In [9] tasks are divided into subtasks according to their semantic and these subtasks have specific times to execute according to the application. However, differently from our model, the time to start a subtask is always constant.

Organization

The remainder of 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

2.1. Definitions

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 j^{th} 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 figure 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 which is responsible by performing operations on devices (resources). 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.

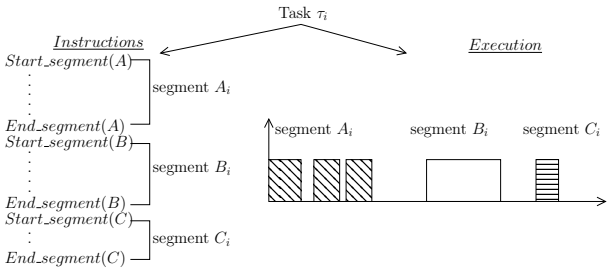


Figure 1. Task τ_i With Segments.

The time-interval problem may also present constraints regarding exclusive access during segment B execution inside the time-interval. Clearly, the nature of the resource under segment B control imposes access constraints to ensure the consistence during resource's operation. Also, we assume that during the execution inside the ideal time-interval the CPU is busy controlling the resources. We assume 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. For instance, in real-time tracking problems and industrial equipment's control the sensors/actuators cannot be shared among tasks while there is an ongoing access. Moreover, to ensure the correct timing behavior the operation cannot be preempted.

2.2. 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 ($W_{B_i} \leq \psi_i \leq \rho_i$). Figure 2 shows two jobs of task τ_i execution (upper section) and the benefit function of segment B_i (lower section).

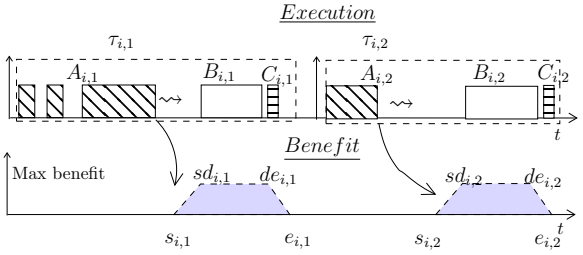
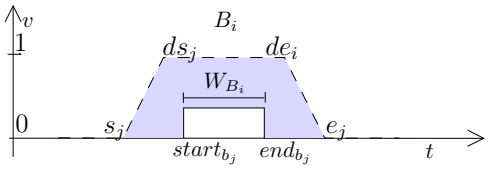


Figure 2. Execution of Task τ_i With a Cumulative QoS Function.

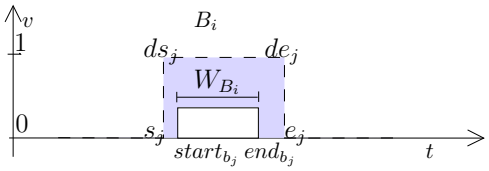
The functions in figures 3 and 4 were made to represent ordinary applications requirements and so they also represent different real-time constraints. Figure 4 represents a **strict** benefit where the segment B_i must execute inside the ideal time-interval $[ds_j, de_j]$, otherwise the benefit is $-\infty$, meaning a catastrophic consequence. Figure 3 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 achieved 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. In equation 1 the *QoS* is shown as the cumulative benefit by the execution of segment B_{ij} inside the time-interval. The equation results in a value in $[0\%, 100\%]$ and 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. As previously stated, the execution of segment



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

Figure 3. *QoS* For a Cumulative Benefit.



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

Figure 4. *QoS* For a Strict Benefit.

B_i is not always required and in this case there is not an associated *QoS* value.

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

3. Scheduling Approach

In most scheduling problems the main interest is to ensure the tasks' deadline will not be missed. In this cases, the deadlines as well as the periods are embedded constraints in the problem definition with a direct correspondence in physical world.

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 [6]. 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 **precedence** among subtasks. In *EDF*, the correct precedence can be enforced through the deadlines of each subtask [4]. Considering the original deadline of task τ_i as D_i, it is possible to assign new deadlines to each subtask such that $D_{A_i} < D_{B_i} < (D_{C_i} = D_i)$. Thus, the execution order will agree with the precedence constraint.

A different problem appears when instead of only a precedence relation among subtasks there is a temporal constraint, such as in the time-interval problem. The time-interval definition states that segment B_i must start inside a time window 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.

Without loss of generality 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 figure 5.

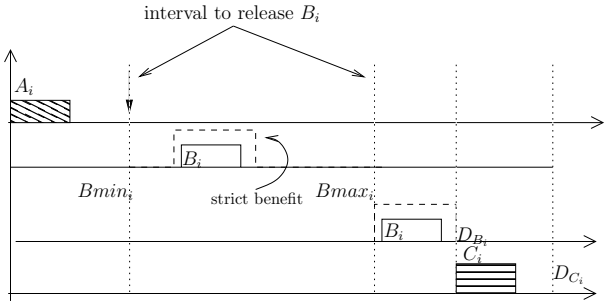


Figure 5. Limits to Release B_i.

The second challenge is the **release time** of subtasks. We need to enforce in the schedulability test that a subtask must be release only after a predetermined time. In this situation, we apply offsets[28],[13],[26] 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. In a similar way, the feasibility test must ensure the execution of the non-preemptive subtask cannot be preempted by any other subtask. Jeffay and Stone in [15] proposed a schedulability test to verify the schedulability of tasks in the presence of interrupts, which are non-preemptive tasks with the highest priority.

3.1. Offline Feasibility Test

In the following subsections, we verify the schedulability of a task set τ splitting the problem in two parts as shown in figure 6. 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 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 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 the minimum and maximum QoS which can be achieved by all cumulative subtasks B_i.

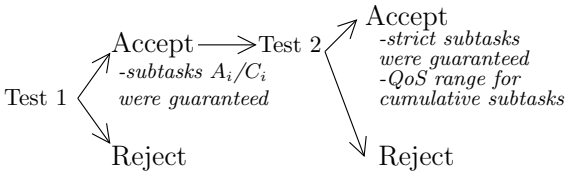


Figure 6. Feasibility Tests.

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 [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 $g_i(t_1, t_2)$ 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 figure 7 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.

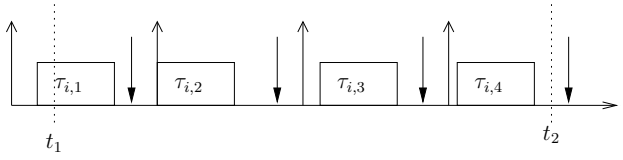


Figure 7. Jobs of τ_i.

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 :

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

The schedulability of an asynchronous task set with deadline less than or equal to period can be verified by equation 3. In asynchronous task sets the schedule must be verified up to $[2H + \Phi]$ [19] where H is the hyper-period ($H = lcm\{T_1, T_2, \ldots, T_n\}$) 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 $O(H^2)$ [12].

$$\forall t_1, t_2 \qquad g(t_1, t_2) \leq (t_2 - t_1) \quad (3)$$

Clearly, for the special case of synchronous task sets, the offsets $\Phi_i = 0\ \ \forall i \in \{1 \ldots n\}$, $t_1 = 0$, $t_2 = L$ and the schedule repeats itself every hyper-period H. Furthermore, for the specific case where the deadlines are equal to periods, the test based on processor demand is equivalent to the test based on processor utilization.

Accounting the Interference of Subtasks B

An important step to check the feasibility of preemptive and non-preemptive tasks was made by Jeffay and Stone in [15]. The authors have shown a schedulability condition in a model to ensure the schedulability of *EDF* in the presence of interrupts. Basically, the authors 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 $f(L)$.

Theorem 3.1 *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:

$$f(0) = 0$$

$$f(L) = \begin{cases} f(L-1) + 1, \\ \quad if \sum_{i=1}^{m} \lceil \frac{L}{TH_i} \rceil CH_i > f(L-1) \\ f(L-1), \\ \quad otherwise \end{cases} \quad (4)$$

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 3.1. 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}. 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:

$\forall t_1, t_2 \geq 0 \qquad g(t_1, t_2) \leq (t_2 - t_1) - F(t_1, t_2)$

In the time-interval problem, subtasks B have a time-window during which can be active. So, applying [15] 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 $Bmin$.

The algorithm 1 has complexity $O(H^2)$. Unfortunately, in the worst case 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.

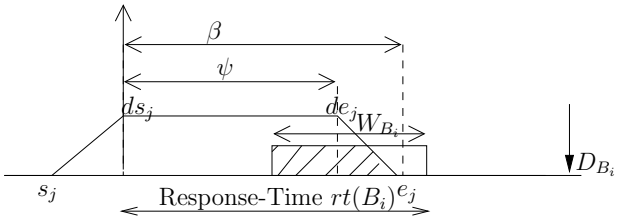
Algorithm 1 Feasibility test - first test.

```
for all t1 such that 0 ≤ t1 ≤ 2H + Φ do
  for all t2 such that t1 ≤ t2 ≤ 2H + Φ do
    g(t1,t2) = Σ_{i=1}^{n} max{0, ⌊(t2+Ti−Di−Φi)/Ti⌋ − ⌈(t1−Φi)/Ti⌉}Wi
    F(t1,t2) = f(t2) − f(t1)
    if g(t1,t2) > (t2 − t1) − F(t1,t2) then
      return nonfeasible
    end if
  end for
end for
{It is feasible. Apply the second test}
return feasible
```

3.1.2 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 a subtask has to slide inside the ideal time-interval and obtain the highest QoS.

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 figure 8. 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 figure 3) it is possible to finish after the ideal time-interval, resulting in a lower QoS. In contrast, subtasks with a strict criticality (figure 4) 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 8. 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 8. Therefore, applying the $wcrt$ of a subtask B_i as a response-time in figure 8 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 8 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 now we are 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 $\leq \psi$ (resulting in $QoS = 100\%$), otherwise the QoS is $-\infty$ and the task set is rejected.

Computing the Response-Time

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

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

$$bcrt_{B_i} = W_{B_i} \quad (6)$$

The first term in equation 5 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-intervals of B_i and B_j may not overlap and it may be impossible for B_j to produce interference upon B_i, even though it has a higher priority.

The best-case response time for subtasks B_i (shown in equation 6) occurs when B_i does not suffer any interference from other subtasks B_j (e.g: when all other subtasks B_j are not required to execute). As a result, the best-case response time of B_i is its own worst-case execution time. Under the assumption that $W_{B_i} \leq \psi_i$ (section 2.2) and $bcrt_{B_i} = W_{B_i}$, the maximum QoS given by the offline test will be always 100% if the task set is feasible.

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.

τ	subtask	W_i	D_i	T_i	Φ_i
τ_1	A_1	4	10	40	0
	B_1	6	31	40	11
	C_1	2	40	40	31
τ_2	A_2	3	20	40	0
	B_2	2	34	40	20
	C_2	2	40	40	34
τ_3	A_3	2	15	60	0
	B_3	6	31	60	18
	C_3	1	60	60	31

Table 1. Example With Three Tasks.

The specific parameters of subtasks B such as criticality, priority($prio$), ρ,ψ,$Bmin$ and $Bmax$ are presented in table 2. The results of the offline test can be seen in table 3. 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 offline test, the $wcrt$ shown in table 3 is an upper bound of the rt values. Therefore, we should expect that the actual minimum *QoS* (obtained by simulation) might be higher than 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 the values given by the offline test.

We simulate the same task set for 10.000 time units, assuming the release time uniformly chosen between $Bmin$ and $Bmax$ (table 4). 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.

subtask	criticality	prio	ρ	ψ	Bmin	Bmax
B_1	cumulative	3	12	10	10	20
B_2	strict	1	8	8	20	26
B_3	cumulative	2	14	8	15	20

Table 2. Parameters of Subtasks B.

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. Offline Feasibility 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%

Table 4. Results Through Simulation.

5. Conclusions and Future Work

xxx This paper presents the time-interval model for expressing timing constraints that do not naturally admit expression in terms of deadlines and periods. The model is useful for a niche of real-world problems which by the lack of theoretical study, are implemented with conventional schedulers resulting in lack of predictability. It applies benefit functions to express when an action should be performed for the maximum benefit.

We apply some approaches from the real-time literature with adaptations to our scheduling problem to create an offline test. Besides an accept/reject answer for tasks with critical benefit constraints, the offline test gives a minimum and maximum expected benefit for non-critical benefit tasks. As a future work, we intend to investigate how to dynamically re-order the non-preemptive sections to increase the benefit during run-time.

ACKNOWLEDGMENT

The authors thank the CNPq and CAPES for financial support.

References

[1] Awerbuch, Bartal, Fiat, and Rosen. Competitive Non-Preemptive Call Control. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1994. 1

[2] B. Awerbuch, R. Gawlick, F. T. Leighton, and Y. Rabani. On-line Admission Control and Circuit Routing for High Performance Computing and Communication. In *IEEE Symposium on Foundations of Computer Science*, pages 412–423, 1994. 2

[3] K. R. Baker and G. D. Scudder. Sequencing with Earliness and Tardiness Penalties: A Review. *Operations Research*, 38:22–36, 1990. 2

[4] J. Blazewicz. Scheduling Dependent Tasks with Different Arrival Times to Meet Deadlines. In *Proceedings of the International Workshop on Modelling and Performance Evaluation of Computer Systems*, pages 57–65. North-Holland, 1976. 4

[5] A. Burns, D. Prasad, A. Bondavalli, F. D. Giandomenico, K. Ramamritham, J. Stankovic, and L. Strigini. The Meaning and Role of Value in Scheduling Flexible Real-Time Systems. *Journal of Systems Architecture*, 46:305–325, 2000. 2

[6] G. C. Buttazzo. Rate Monotonic vs. EDF: Judgment Day. In *Real-Time Systems*, 2005. 2, 4

[7] G. C. Buttazzo, M. Spuri, and F. Sensini. Value vs. Deadline Scheduling in Overload Conditions. In *IEEE RTSS*, pages 90–99, 1995. 2

[8] K. Chen and P. Muhlethaler. A Scheduling Algorithm For Tasks Described By Time Value Function. *Real-Time Systems*, 10(3):293–312, May 1996. 2

[9] A. Crespo, I. Ripoll, and P. Albertos. Reducing Delays in RT Control: the Control Action Interval. In 14^{th} *IFAC World Congress on Automatic Control*. Elsevier Science, 1999. 2

[10] F. R. de la Rocha and R. S. de Oliveira. Time-Interval Scheduling and its Applications to Real-Time Systems. In *Proceedings of the 27th Real-Time Systems Symposium-WiP*, 2006. 2

[11] J. Dey, K. J., and D. Towsley. On-line scheduling policies for a class of IRIS (increasing rewardwith increasing service) real-time tasks. In *IEEE Transactions on Computer*, 1996. 2

[12] J. Goossens. *Scheduling of Hard Real-Time Periodic Systems with Various Kinds of Deadline and Offset Constraints*. PhD thesis, Université Libre de Bruxelles, 1999. 5

[13] J. P. Gutierrez and M. G. Harbour. Offset-Based Response Time Analysis of Distributed Systems Scheduled under EDF. In *15th Euromicro Conference on Real-Time Systems (ECRTS'03)*, 2003. 4

[14] R. Hassin and M. Shani. Machine scheduling with earliness, tardiness and non-execution penalties. *Computers and Operations Research*, 32:683–705, 2005. 2

[15] K. Jeffay and D. L. Stone. 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, December 1993. 4, 5, 6

[16] E. Jensen, C. Locke, and H. Tokuda. A Time-Driven Scheduling Model for Real-Time Operating Systems, 1985. 2

[17] S. k. Baruah, R. R. Howell, and L. E. Rosier. Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor. *Real-Time Systems*, 2:301–324, 1990. 5

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

[19] J. Leung and M. Merill. A Note on the Preemptive Scheduling of Periodic, Real-Time Tasks. *Information Processing Letters*, 11(3):115–118, Nov 1980. 5

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

[21] P. Li. *Utility Accrual Real-Time Scheduling: Models and Algorithms*. PhD thesis, Virginia Polytechnic Institute and State University, 2004. 2

[22] R. J. Lipton and A. Tomkins. Online Interval Scheduling. In *SODA '94: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 302–311, Philadelphia, PA, USA, 1994. 2

[23] J. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. In *Proceeding of the IEEE*, volume 82, pages 83–94, 1994. 2

[24] C. D. Locke. Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives. *Real-Time Systems*, 4(1):37–53, 1992. 2

[25] R. Mazzini and V. A. Armentano. A Heuristic For Single Machine Scheduling With Early And Tardy Costs. *European Journal of Operational Research*, 1:129–146, 2001. 2

[26] R. Pellizzoni and G. Lipari. A New Sufficient Feasibility Test For Asynchronous Real-Time Periodic Task Sets. In *Proceedings. 16th Euromicro Conference on Real-Time Systems*, 2004. 4

[27] B. Ravindran, E. D. Jensen, and P. Li. 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, 2005. 1

[28] K. Tindell. Using offset information to analyse static priority pre-emptively scheduled task sets. Technical report, University of York, YCS-92., 1992. 4

[29] H. Tokuda, J. W. Wendorf, and H. Wan. Implementation of a time-driven scheduler for real-time operating systems. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 271–280, 1987. 2

[30] M. Velasco, P. Martí, and J. M. Fuertes. The Self Triggered Task Model for Real-Time Control Systems. In *In Work-in-Progress Session of the 24th IEEE Real-Time Systems Symposium (RTSS03)*, 2003. 2

[31] H. Wu, B. Ravindran, E. D. Jensen, and U. Balli. Utility Accrual Scheduling under Arbitrary Time/Utility Functions and Multi-unit Resource Constraints. In *Proceedings of the* 10^{th} *RTCSA*, 2004. 2