Performance optimization for hard real-time fixed priority tasks

Joël Goossens
Université Libre de Bruxelles
Brussels, Belgium

Pascal Richard
LISI-ENSMA
Poitiers, France

Abstract: Many real-time systems must simultaneously handle hard real-time constraints and Quality of Service constraints. Thus, a performance criterion must be optimized for some tasks while meeting all deadlines for hard real-time tasks. For that purpose, we present a generic Branch & Bound method dedicated to hard real-time fixed-priority schedulers. Our approach is based on a generic algorithm that defines common functions to any performance criteria. Only few functions need to be specialized to deal with a specific criterion. We then present such a specialization for minimizing the weighted average response time of synchronously released and constrained deadline tasks. This performance measure aims to minimize work-in-progress in the system.

Contents

Key works Scheduling, Fixed-Priority, Performance Optimization

1 Introduction

The use of computers to control safety-critical real-time functions has increased rapidly over the past few years. As a consequence, **real-time systems** — computer systems where the correctness of a computation is dependent on both the logical results of the computation *and* the time at which these results are produced — have become the focus of much study.
Since the concept of **"time"** is of such importance in real-time application systems, and since these systems typically involve the sharing of one or more resources among various contending processes, the concept of **scheduling** is integral to real-time system design and analysis. Scheduling theory as it pertains to a finite set of requests for resources is a well-researched topic. However, requests in real-time environment are often of a recurring nature. Such systems are typically modelled as finite collections of simple, highly repetitive **tasks**, each of which generates *jobs* in a very predictable manner. These jobs have upper bounds upon their worst-case execution requirements, and associated deadlines. In this work, we consider periodic task systems, where each task makes a resource request at regular periodic intervals. The processing time and the time elapsed between the request and the deadline are always the same for each request of a particular task. In our model of computation, each periodic task is characterized by the 4-tuple (C_i, D_i, T_i, w_i), with $0 < C_i \leq D_i$, $C_i \leq T_i$ and $w_i \geq 0$, i.e., by a period T_i, a hard deadline delay D_i, an execution time C_i, and the weight w_i. The requests of τ_i are separated by T_i time units and occur at time $(k-1)T_i$ ($k = 1, 2, \dots$). The execution time required for each request is C_i time units; C_i can be considered as the worst-case execution time for a request of τ_i. The execution of the k^{th} request of task τ_i, which occurs at time $(k-1)T_i$, must finish before or at time $(k-1)T_i + D_i$; the deadline failure is fatal for the system: the deadlines are considered to be hard. All timing characteristics of the tasks in our model of computation are assumed to be natural integers (it may be noticed that the value w_i's can be real numbers as well). Interesting sub-cases of periodic tasks are *implicit deadline* systems, where each deadline coincides with the period (i.e., each request must simply be completed before the next request of the same task occurs); *constrained deadline* systems, where the deadlines are not greater than the periods and *arbitrary deadline* systems, where no constraint exists between the deadline and the period (notice that if the deadline is greater than the period, many requests of a single task may be simultaneously active, even if the system is feasible, i.e., all deadlines are met).

Real-time scheduling theory has traditionally focused upon the development of algorithms for *feasibility analysis* (determining whether all jobs can complete execution by their deadlines) and *run-time scheduling* (generating schedules at run-time for systems that are deemed to be feasible) of such systems.
In some real-time applications, tasks are not only subject to complete by their deadlines, but more control of the on-line schedule is required. Typically, we consider additionally the notion of Quality of Service (QoS), interpreted in many ways (e.g., response time, jitters, etc.). In the real-time environment, the objective is then to optimize a performance measure *while* respecting the deadlines of jobs. The parameters w_i defines the importance of the tasks τ_i in terms of the optimization criterion. If there are no particular constraints on the criterion, then this weight is set to 0.
Most uniprocessor scheduling algorithm operate as follows: at each instant, each active job is assigned a distinct priority, and the scheduling algorithm chooses for execution the currently active job (if any) with the highest priority. Some scheduling algorithms permit that periodic tasks τ_i and τ_j both have active jobs at time t and t' such that at time t, τ_i's job has higher priority than τ_j's while at time t', τ_j's job has higher priority than τ_i's. Algorithms that permit such "switching" of priorities between tasks are known as **dynamic** priority algorithms. An example of dynamic priority scheduling algorithm is the earliest deadline first scheduling algorithm [8].
By contrast, **static** priority algorithms satisfy the property that for every pair of tasks τ_i and τ_j, whenever τ_i and τ_j have both active jobs, it is always the case that the same task's job has higher priority. An example of a static-priority scheduling algorithm for periodic scheduling is the rate-monotonic scheduling algorithm [8].

This research The focus of our current research is the scheduling of periodic task set using a fixed priority assignment which (if any) meets all the deadlines *and* optimizes (minimizes or maximizes) an optimization criterion. Our contribution is threefold: (i) we propose a Branch & Bound algorithm which is generic in the sense that the technique is not dedicated to a particular optimization criterion nor a specific periodic task (sub-)model (implicit/constrained/arbitrary deadlines), (ii) we resolve the following specific case: minimizing the average response time for constrained deadline periodic task sets. This performance measure aims to minimize work-in-progress in the system [9]. In particular we propose an efficient lower bound based on the

work of Redell [10], (iii) we also complete the iterative process (considered in [10]) to compute the best case response time.

Organization of the document The remainder of the document is organized as follows. In Section 2, we shall present our generic Branch & Bound algorithm. In Section 3, we apply our generic technique to a specific case: minimizing the average response time for constrained deadline periodic task sets.

2 Generic algorithm: B & B algorithms for fixed priority schedulers

Branch & Bound algorithms are very used to solve combinatorial problems. These algorithms are based on the idea of cleverly enumerating all feasible solutions [3]. Such algorithms are also used to schedule tasks subjected to hard deadline constraints. In [5] a parameterized Branch & Bound that schedules precedence-constrained tasks on a multiprocessor system, to minimize the maximum lateness of tasks is considered. Recently, Pan [9], proposed a Branch & Bound algorithm that schedules non recurring and non-preemptive deadline constrained tasks and minimizes total weighted completion time of jobs.

In this research, we focus our study to static priority algorithms (also known as fixed priority assignments). Partial solutions are partial priority assignment (i.e., there exist prioritized and unprioritized tasks). The enumeration principle is based on the exploration in a search tree. Each vertex in the search tree corresponds to a task priority assignment. Vertices are partial priority assignments and they are stored in a search tree. Leaves of the search tree define feasible solutions of the problem: all tasks have been assigned a priority. Since the number of feasible solutions is exponential due to the computational complexity of the problem, the only way to find a solution in an acceptable amount of time is to detect as soon as possible vertices that will not lead to an improvement of the best known solution for the optimized criterion.

Without loss of generality, we only minimize performance measures (since maximizing an objective function f is equivalent to minimizing $-f$). A preliminary step is needed before starting the Branch & Bound algorithm: an heuristic have to compute an *upper bound* of the objective function. Implementation of a Branch & Bound method is usually based on a set of non explored vertices, called the *Active Set*. At each step of the main loop of the

Algorithm 1: Generic Branch & Bound algorithm

```
Compute the upper bound;
Initialize ActiveSet A with vertices 1 . . . n (the first level of the search
tree);
while A ≠ ∅ do
    Select a vertex in A according to the vertex selection rule;
    if it is a leaf vertex and the criterion is improved (using the lower
    bound) then
        update the best solution;
    else if it is not a leaf vertex then
        Generate a set B of child vertices according to the vertex
        branching rule;
        Calculate the lower bounds for each vertex in B;
        Eliminate vertices in B according the vertex elimination rule;
        Sort the vertices in B in non-decreasing order of the vertex
        selection rule;
        Moves all remaining vertices in B to the active set A;
    Remove the selected vertex from A;
end
```

Branch & Bound algorithm, the explored vertex is separated, that is to say its children are defined by assigning a priority to one task in the previous partial solution. To each of them is computed a *lower bound* of the objective function. These vertices are pruned (deleted) if the lower bound of the criterion is not smaller than the best known solution (the upper bound) or if the constraints of the problem are not satisfied. Algorithm 1 presents the pseudo-code of the Branch & Bound algorithm.
The Branch & Bound algorithm is based on the following characteristics:

- vertex selection rule: this rule selects the next candidate vertex in the Active Set.
- vertex branching rule: this rule defines the traversal strategy in the search tree.
- upper bound: it is an upper bound of the objective function, computed first, in order to obtain a reachable value of the criterion while respecting

all constraints of the problem. The upper bound is then updated during the search when a leaf leads to an improvement of the best known solution.

- lower bound: it is used to evaluate a partial solution. Every time a lower bound is not smaller than the best known solution (upper bound), then the candidate vertex is pruned (i.e., its children will not be created in the search tree). When a leaf is reached, then the objective function is exactly computed, e.g., by building up the schedule using a simulator of the fixed-priority scheduler.

- elimination rule: this rule checks that generated children will lead to a feasible solution (i.e., all constraints are satisfied). Moreover, the rule checks if the lower bound is not smaller than the best known solution (upper bound). Every time it is not the case, then these vertices and all their successors will not be considered during the search (they are pruned of the search tree).

We now detail the generic features of these components.

2.1 Vertex selection rule

The vertex selection rule defines the next candidate vertex in the Active Set. Classical rules are FIFO (First-In First-Out), LIFO (Last-In First-Out) and LLB (Least Lower Bound first). This rule is closely related to the vertex branching rule to defines the traversal of the search tree.

2.2 Vertex branching rule

The branching rule defines the order of vertices in the Active Set. Different ordering techniques of the Active Set lead to different searching strategies in the tree. Well-known strategies are depth-first search or breadth-first search. From an algorithmic point of view, a depth-first search strategy is obtained by managing the Active Set as a stack, while a breadth-first search is obtained by implementing the Active Set as a queue. For convenience, we consider next the search tree rather than a list of unexplored vertices that stores partial priority assignments.
According to our priority assignment problem, a vertex is characterized by task index. Levels of the vertices in the tree are interpreted as priority levels

of tasks. The first level in the tree defines the tasks assigned to the highest priority level, and in the same way, the leaves define tasks assigned to the lowest priority level. The number of vertices in a search tree is clearly exponential in the size of the problem. Since the number of vertices at level ℓ is the number of permutations of k elements among n, then the number of vertices at level k is: $\frac{n!}{(n-k)!}$. Thus, the total number of vertices in the search tree is: $\sum_{i=0}^{n-1} \prod_{k=0}^{i} (n-k)$.
The traversal strategy has an important impact on the largest size of the Active Set. The most commonly used strategy to solve scheduling problems is the depth-first search one. The main interest of the latter method is the size of the Active Set which is polynomially bounded by the size of the problem (which is not the case for other strategies, especially when a breadth-first search is used). In that way, the maximum number of vertices stored simultaneously in the Active Set is $n(n+1)/2$, generated while reaching the first leaf of the search tree. In order to implement a depth-first search strategy in our Branch & Bound, the vertex selection rule is the LIFO strategy.

2.3 Upper bound

The first step of the Branch & Bound procedure consists in applying an heuristic algorithm to compute a feasible priority assignment. The value of the performance measure for an initial solution defines an upper bound. This value is used until a better one is found during the optimization stage.
The performance of a Branch & Bound method is clearly related to tightness of upper bounds. The upper bound must be as tight as possible. Otherwise, most of vertices in search tree will be explicitly enumerated. For that reason, the choice of the upper bound and consequently the efficiency of the Branch & Bound is closely related to the specific objective function.

2.4 Lower bound

Lower bounds is used to evaluate a partial priority assignment. This function is specialized to the minimized objective function. But, every lower bound function must be non-decreasing, i.e., lower bounds of children must be not smaller than the one of the father. Otherwise, the Branch & Bound algorithm cannot be an optimal method. A specific objective function will be detailed for the average weighted response time of tasks in the Section 3. Once again,

the choice of the upper bound and consequently the efficiency of the Branch & Bound is closely related to the specific objective function.
When a leaf is reached, then in order to allow the computation of any objective function, the schedule is built up from time 0 to $\text{lcm}_{i=1\ldots n} T_i$, since we know that we know that the schedule itself is periodic with a period $H \stackrel{\text{def}}{=} \text{lcm}\{T_i | i = 1, \ldots, n\}$ (see for instance [4], Corollary 2.49, p. 47 for a proof).

2.5 Elimination rule

Elimination rule is of prime interest in Branch & Bound methods because they can drastically prune the search tree. We next detail three generic conditions to prune a vertex:

- immediate selection;
- a deadline is not met;
- the lower bound is greater than the best known solution (upper bound).

2.5.1 Immediate selection

The immediate selection rule detects some children that will not lead to a feasible solution, before generating them. The following result leads to such a rule.

Theorem 1 *[11] For all pair τ_i, τ_j of tasks, if:*

$$\left\lceil \frac{T_i}{T_j} \right\rceil \times C_j + C_i > D_i \tag{1}$$

then the priority of τ_i must be higher than the priority of τ_j

According to this result, we can define a set of constraints that must be satisfied during the branching step. For instance, if we detect that τ_i must have a higher priority than τ_j according to Theorem 1, then task τ_i must be generated before τ_j in the search tree. Furthermore, τ_j can be assigned a priority if, and only if, τ_i has already been assigned one in a predecessor vertex. In practice, a matrix is defined before starting the Branch & Bound algorithm that stores precedence constraints, associated to Theorem 1. These precedence constraints will be enforced while generated children in the search tree. They are obviously computed and checked in polynomial times.

2.5.2 Checking deadlines

We now detail the feasibility of deadlines for the current vertex. The vertex branching rule generates children of the vertex currently explored while respecting the immediate selection rule. If the vertex of level $i-1$ is separated, then all its children are assigned to the priority level i. The first step is to verify that the task assigned to this priority level ($\tau_{[i]}$) meet their deadlines. Since the vertex is newly generated, then we always know that tasks assigned to the priority levels $1 \dots k$ are feasible. The other tasks do not interfere with the execution of $\tau_{[i]}$. We can check the feasibility of $\tau_{[i]}$ by calculating the worst-case response time of the task.

The worst-case response time of a task can be computed in pseudo-polynomial time by using the method presented in [6] for constrained deadline tasks. We consider tasks with arbitrary deadlines. An extension has been proposed to calculate the worst-case response times of tasks for this kind of task set [7, 12]. This method is still running in pseudo-polynomial time since it is related to the length of the level-i busy period (an interval of time where the processor executes only tasks having a priority higher than or equal to i). But, as far as we know, checking feasibility of given priorities cannot be performed in polynomial time (i.e., the computational complexity status of this problem is unknown). Let $[i]$ denote the index of the task at priority level i. The worst-case response time of a task $\tau_{[i]}$ depends on the workload of the higher priority tasks. The time demand of tasks having a priority higher than i at time t in a busy period containing q releases of $\tau_{[i]}$ is $W(t)$:

$$W(t) = (q+1)C_i + \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_{[j]}} \right\rceil C_{[j]} \tag{2}$$

The worst-case response time of τ_i is then obtained by considering $q = 1, 2 \dots$ and for each value of q the recurrent equation $t = W(t)$ is solved, that is to say:

$$\begin{cases} R_i^{(0)} &= C_i \\ R_i^{(k)} &= W(R_i^{(k-1)}) \end{cases} \tag{3}$$

$$R_i = R_i^{(k)} = R_i^{(k-1)} \tag{4}$$

Computations stop when the following inequality is satisfied: $R_i \leq (q+1)T_i$
To summarize, we use the following rule:

If $R_i > D_i$ for a vertex, then it is pruned, as well as all its successors.

We now have to consider unprioritized tasks associated to the current vertex. Whatever the priorities are for prioritized tasks, the interference due to these tasks is still the same for unprioritized tasks. One can check that there exists at least one feasible schedule by assigning priority levels using the Deadline Monotonic rule. Feasibility can then be checked by using the previous feasibility test.

2.5.3 Lower bound constraint

The last way to prune a vertex is performed when the best known solution (i.e., an upper bound) is lower than the lower bound of the current vertex. Generation of the children can only increase lower bounds. So, when a leaf should be reached, the value of the objective function will not be improved, and it will be greater than or equal to the best known upper bound.
To conclude, one can remark that all these tests are performed in pseudo-polynomial time. This complexity is only due to the computation of worst-case response times of tasks.

3 The specific case of the average response time

In this section, we shall apply our generic technique to the following specific case: we consider the scheduling of constrained deadline periodic task sets for which we would like to minimize the (weighted) average response time. In a previous paper, we studied the problem of minimizing the weighted sum of worst-case response times of tasks [11]. This criterion only captures the average performance of a real-time systems by taking into account only worst-case response times. In the remainder, we exactly computes the average response time of tasks (i.e., we consider all jobs of all tasks).
As mentioned in the introduction, this performance criterion aims to minimize work-in-progress in the system [9]. Minimizing the average response time of jobs is equivalent to minimize the average completion time of jobs, since these two performance measures only differ by an additive term : the sum of release dates of jobs. Furthermore, minimizing the average response time leads

to minimize the average number of uncompleted jobs in the system and it also minimizes the average waiting time of jobs to start their executions [2]. Thus, the system is more responsive. When jobs are not subjected to deadlines, minimizing the average response time of jobs upon a single processor is achieved by scheduling jobs according to the SRPT rule: Shortest Remaining Processing Time first [2]. Obviously, this rule is not optimal to meet deadlines of jobs. Lastly, we must notice that is inserted idle-times in the schedule are not useful to minimize the average response time of jobs [2].
In Section 3.1 we shall present our optimization criterion, in Section 3.2, the priority assignment procedure, in Section 3.4 the lower bound used to eliminate vertex and experimental results in Section 3.5.

3.1 The optimization criterion

In the remainder $R_{i,k}$ denote the response time of the k^{th} request of the task τ_i, $\bar{R}_i$ denote the average response time for the requests of task τ_i. In this section, we shall apply our technique to minimize the weighted average response time with the following definition:

$$\bar{R} \stackrel{\text{def}}{=} \lim_{f \to \infty} \frac{1}{f} \sum_{i=1}^{n} \sum_{j=1}^{f} w_i \cdot R_{i,j} \tag{5}$$

We shall first simplify equation 5. Since the tasks are periodic, we know that the schedule itself is periodic with a period $H \stackrel{\text{def}}{=} \text{lcm}\{T_i | i = 1, \ldots, n\}$. Consequently, the sum of the response times during the interval $[0, k \cdot H)$ ($k \geq 0$) can be defined as follows:

$$\sum_{j=1}^{k \cdot H/T_i} R_{i,j} = k \cdot \sum_{j=1}^{H/T_i} R_{i,j} \tag{6}$$

Consequently, for $\bar{R}_i$ (the average response time for the jobs of task τ_i) we get

$$\bar{R}_i = \lim_{k \to \infty} \frac{T_i}{k \cdot H} \sum_{j=1}^{k \cdot H/T_i} R_{i,j} = \frac{T_i}{H} \sum_{j=1}^{H/T_i} R_{i,j} \tag{7}$$

Hence, our optimization criterion is:

$$\bar{R} = \sum_{i=1}^{n} \left(\frac{w_i \cdot T_i}{H} \sum_{j=1}^{H/T_i} R_{i,j} \right) \tag{8}$$

3.2 The priority assignment

Our algorithm assigns priorities from the higher to the lowest priority. A node in our search tree corresponds to a task, its level corresponds to its priority. For each node (leaf excepted) we evaluate a lower bound for our criterion (with a pseudo-polynomial algorithm). For the leaves, we compute the exact value of our criterion (e.g., by simulation).

3.3 The upper bound

We consider synchronous and constrained deadline task sets for which the deadline monotonic algorithm is an optimal priority assignment regarding the feasibility of the system. But such a priority ordering is of course not optimal while optimizing an arbitrary objective function.
Nevertheless, this priority ordering leads to a feasible schedule (if the system is feasible), and can be used to compute the upper bound.
A better approach is to use the priority ordering defined in [1]. This priority ordering always leads to a better solution in comparison to the Deadline Monotonic schedule. We assign priorities in reverse order. Such backward priority ordering is based on an interesting property of fixed-priority schedules.

Lemma 1 *[1] if a task τ_i is* lowest-priority viable *(i.e., τ_i is assigned the lowest priority, the remaining tasks are assigned higher priorities in an any arbitrary order and jobs generated by τ_i may not miss any deadline) then there is a feasible static priority assignment for τ iff there is a feasible static priority assignment for $\tau \setminus \{\tau_i\}$*

The heuristic assigns priorities to tasks from the lowest priority to the highest one, and at each step it chooses a task that increases as less as possible the objective function among all tasks that are feasible for the current priority level. At a given step, the algorithm selects the task having the smallest weight among unprioritized tasks. As a direct consequence of Lemma 1, the heuristic always finds a feasible priority assignment if one exists. In Algorithm 2, we give the pseudo-code of the heuristic.

3.4 The lower bound

In this section, we shall characterize a lower bound (and an iterative process for its evaluation) for our criterion. We have to distinguish between two kinds

Algorithm 2: Upper Bound pseudo-code

Let X be the set of tasks and n be the number of tasks ;
Let $z = 0$ be the objective function;
while $X \neq \emptyset$ **do**
    Let X' the tasks feasible at priority level n among tasks belonging to X;
    Let a be the task with the lowest weighted contribution to the optimized criterion in X';
    Assign priority level n to a;
    $n = n - 1$;
    $X = X \setminus \{a\}$;
end
Compute the objective function associated to the defined priority ordering;

of tasks: tasks which have already a priority and tasks which have no priority yet.

3.4.1 Prioritized tasks

We consider the task τ_i and assume that the task τ_i has already a priority assigned.
Let us define, on the interval $[0, H)$:

- $R_i^{\max}$ the largest response time among the jobs of τ_i.
- $R_i^{\min}$ the smallest response time among the jobs of τ_i.

We consider the pessimistic case where $(\frac{H}{T_i} - 1)$ jobs of τ_i have a response time equal to $R_i^{\min}$ and a single job of τ_i has a response time equal to $R_i^{\max}$, consequently:

$$\bar{R}_i \geq \frac{T_i}{H}\left((\frac{H}{T_i} - 1) \cdot R_i^{\min} + R_i^{\max}\right)$$

Notice that since we know all highest priority tasks than τ_i, $R_i^{\max}$ is the minimal positive solution of the equation (see [6] for a proof)

$$R_i^{\max} = C_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{R_i^{\max}}{T_j} \right\rceil \cdot C_j \tag{9}$$

The computation of $R_i^{\min}$ is more complex, from [10] we define a lower bound for the best case response time (better than C_i), we denote by R_i^{lb} this lower bound, which is the largest positive solution to the equation

$$R_i^{\text{lb}} = C_i + \sum_{j \in \text{hp}(i)} \left\lceil \frac{\max\{R_i^{\text{lb}} - T_j, 0\}}{T_j} \right\rceil \cdot C_j$$

An iterative process can be defined to find the largest positive solution:

$$\begin{aligned} \mathcal{R}_0 &= \text{init} \\ \mathcal{R}_k &= C_i + \sum_{j \in \text{hp}(i)} \left\lceil \frac{\max\{\mathcal{R}_{k-1} - T_j, 0\}}{T_j} \right\rceil \cdot C_j \end{aligned}$$

In [10] no value for init is given, we shall here fill the gap.
First, observe that init must be not greater than R_i^{lb}:

$$\begin{aligned} \text{init} &\leq C_i + \sum_{j \in \text{hp}(i)} \left\lceil \frac{\max\{R_i^{\text{lb}} - T_j, 0\}}{T_j} \right\rceil \cdot C_j \\ &\leq \frac{C_i}{1 - \sum_{j \in \text{hp}(i)} \frac{C_j}{T_j}} \end{aligned}$$

Consequently we propose the following iterative process

$$\begin{aligned} \mathcal{R}_0 &= \frac{C_i}{1 - \sum_{j \in \text{hp}(i)} \frac{C_j}{T_j}} \\ \mathcal{R}_k &= C_i + \sum_{j \in \text{hp}(i)} \left\lceil \frac{\max\{\mathcal{R}_{k-1} - T_j, 0\}}{T_j} \right\rceil \cdot C_j \end{aligned}$$

3.4.2 Unprioritized tasks

We consider the case where the task τ_i has not priority already assigned.
The priority of an unprioritized tasks will be chosen in the interval $[k+1, n]$ (k is the number of prioritized tasks). It may be noticed that decreasing the priority of a task can only decrease its response time. More formally, we have:

Lemma 2 *Let $R_{i,j}^{(k)}, 1 \leq j \leq \infty$ be the response time of the k^{th} request of the task τ_i assuming that the priority level of τ_i is k then*

$$k < \ell \Rightarrow R_{i,j}^{(k)} \leq R_{i,j}^{(\ell)} \qquad 1 \leq j \leq \infty$$

Consequently, the average response time cannot increase by decreasing the priority level of a task:

Theorem 2 *Let $\bar{R}_i^{(\ell)}$ be the average response time of the requests of task τ_i assuming that the priority level of τ_i is ℓ, then*

$$k < \ell \Rightarrow \bar{R}_i^{(k)} \leq \bar{R}_i^{(\ell)}$$

It is easy to see that R_i^{lb} and $R_i^{\max}$ are also non decreasing in regard to the priorities.
Consequently, in our priority assignment we assign the priority $k + 1$ to all unprioritized tasks.
Now, we have the material to give the lower bound of our criterion for unprioritized tasks:

$$\sum_{i=1}^{n} \frac{w_i \cdot T_i}{H} \left((\frac{H}{T_i} - 1) \cdot R_i^{\text{lb}} + R_i^{\max} \right) \tag{10}$$

3.5 Experimental results

We experimented this method on randomly generated problems. For a given number of tasks, we randomly generated 25 instances. Problems have between 5 and 25 tasks (step 5) and the utilization factor of the processor is 0.5 for all instances. A uniform law has been used by the random generator and the characteristics of the generation are $C_i \in [1 \ldots 10]$, $D_i = T_i$ are computed in order to have a utilization factor close to 0.5, and $w_i \in [0 \ldots 20]$. Numerical experimentations have been performed on a Pentium IV/2 GHz personal computer. A time limit has been fixed to ten minutes for solving every instance of problems. This time is very short, but in must applications, Branch & Bound algorithms spend a lot of time to prove the optimality of the best known solution.
Figure 1 presents the average resolution time for every size of problem. For instances having less than 20 tasks, the resolution time is very short. But, for instances with 25 tasks, the time limit has been reached for every generated instances.

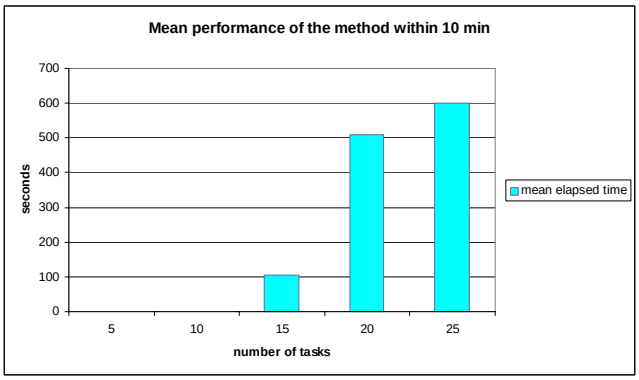

Figure 1: Resolution times over 25 instances (Time limit 600sec. per instance).

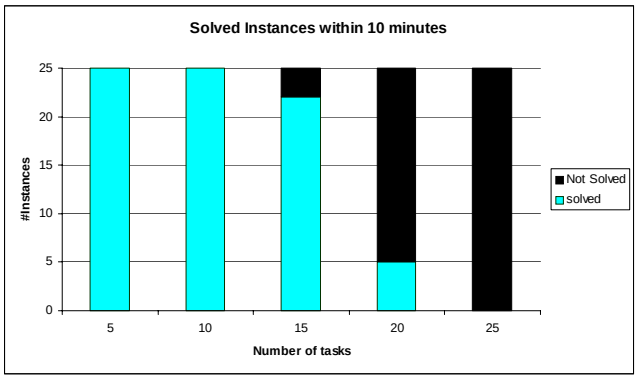

Figure 2: Solved instances versus Unsolved Instances

Figure 2 presents the number of instances solved versus of unsolved instances for each size of problems. We can see that no instance with 25 tasks has been optimally solved within the time limit.
Figures 3 indicates the average number of vertices explored while solving 25 instances of a given problem (i.e., for a fixed number of tasks). From 20 to 25 tasks, the number of explored vertices decreases. When the size of the problem increases, then the method spends more time to evaluate each separated vertex. This is a direct consequence of calculating the worst-case response times of tasks in the elimination rule (these computations are done in pseudo-polynomial time).

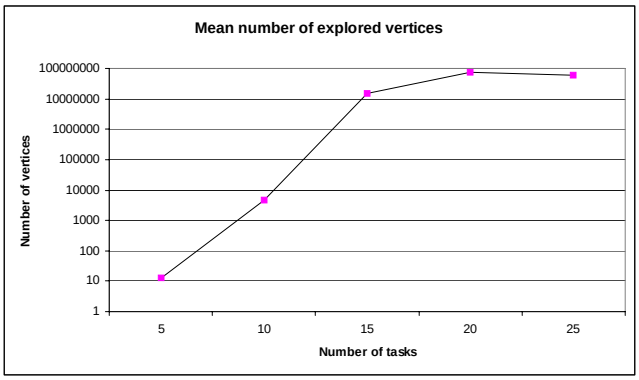

Figure 3: Average number of vertices explored over 25 instances

4 Conclusion

Many real-time systems must simultaneously handle hard real-time constraints and Quality of Service constraints. Thus, a performance criterion must be optimized while meeting all deadlines for hard real-time tasks. For that purpose, we have presented a generic Branch & Bound method dedicated to hard real-time fixed-priority schedulers. Our approach is based on a generic algorithm that used common functions to any optimization criteria. Only few functions need to be specialized to a specific criterion. We have presented such a specialization for minimizing the weighted average response time of synchronously released and constrained deadline tasks.
Perspectives of this work is to develop a tool based on two modules:

- a toolbox, that allows developers to add specific functions needed to consider new performance criteria;
- a Graphical User Interface that allows users to specify problems, to upload functions that are specialized to new criteria in the Branch & Bound procedure, to define numerical experimentations and lastly to exploit numerical results. All these interactions must be independent of performance criteria.

References

[1] AUDSLEY, N. Optimal priority assignment and feasibility of static pri-

ority assignment with arbitrary start times. *YCS 164, University of York* (1991).

[2] BAKER, K. *Introduction to sequencing and scheduling*. John Wiley and Sons, 1974.

[3] BRUCKER, P. *Scheduling algorithms*. Springer Verlag, 2001.

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

[5] JONSSON, J., AND SHIN, K. A parametrized branch and bound strategy for scheduling precedence-constrained tasks on a multiprocessor system. In *proc. Int. Conf. on Parallel Processing* (1997), pp. 158–165.

[6] JOSEPH, M., AND PANDYA, P. Finding response times in a real-time system. *The Computer Journal 29*, 5 (Oct. 1986), 390–395.

[7] LEHOCZKY, J. P. Fixed priority scheduling of periodic task sets with arbitrary deadlines. *proc. IEEE Real-Time System Symposium* (1990), 201–209.

[8] LIU, C., AND LAYLAND, J. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM 20*, 1 (1973), 46–61.

[9] PAN, Y. An improved branch and bound algorithm for single machine scheduling with deadlines to minimize total weighted completion time. *Operations Reseach Letters 31* (2003), 492–496.

[10] REDELL, O., AND SANFRIDSON, M. Exact best-case response time analysis of fixed-priority scheduled tasks. In *Euromicro Real-Time Systems* (2002).

[11] RICHARD, P. Controlling response time in real-time systems. In *Computer Performance Evaluation: Modelling Techniques and Tools, LNCS 2324, Springer Verlag* (2002), pp. 339–348.

[12] TINDELL, K. *Fixed-Priority Scheduling of Hard Real-Time Systems*. PhD thesis, University of York, UK, 1994.