Quasi-Static Scheduling for Real-Time Systems with Hard and Soft Tasks

Luis Alejandro Cortés, Petru Eles, and Zebo Peng
Embedded Systems Laboratory
Department of Computer and Information Science
Linköping University, S-581 83 Linköping, Sweden
{luico,petel,zebpe}@ida.liu.se

Technical Report
September 2003

Abstract

This report addresses the problem of scheduling for real-time systems that include both hard and soft tasks. In order to capture the relative importance of soft tasks and how the quality of results is affected when missing a soft deadline, we use utility functions associated to soft tasks. Thus the aim is to find the execution order of tasks that makes the total utility maximum and guarantees hard deadlines. We consider intervals rather than fixed execution times for tasks. Since a purely off-line solution is too pessimistic and a purely on-line approach incurs an unacceptable overhead due to the high complexity of the problem, we propose a quasi-static approach where a number of schedules are prepared at design-time and the decision of which of them to follow is taken at run-time based on the actual execution times. We propose an exact algorithm as well as different heuristics for the problem addressed in this report.

1 Introduction

There are real-time systems that require the execution of tasks with distinct types of timing constraints. Some of these real-time tasks correspond to activities that must be completed before a given deadline. These tasks are referred to as *hard* because missing one such deadline might have severe

consequences. Such systems also include tasks that are also time-constrained but more loosely and hence referred to as *soft*. A soft-task deadline miss can be tolerated though the quality of results can be degraded.

Scheduling for real-time systems composed of hard and soft tasks has previously been addressed, for example in the context of integrating multimedia into hard real-time systems [10], [1]. Most of the scheduling approaches for mixed hard/soft real-time systems consider that hard tasks are periodic while soft tasks are aperiodic. In this frame, both dynamic and fixed priority systems have been considered. In the former case, the Earliest Deadline First (EDF) algorithm is used for scheduling hard tasks and the response time of soft aperiodic tasks is minimized while guaranteeing hard deadlines [3], [14], [9]. The joint scheduling approaches for fixed priority systems also try to serve soft tasks the soonest while guaranteeing hard deadlines, but make use of the Rate Monotonic (RM) algorithm for scheduling the hard periodic tasks [6], [11], [16].

Scheduling for hard/soft systems permits dealing with tasks with different level of criticality and fits better a broader range of applications (as compared to pure hard real-time techniques), yet providing a certain number of guarantees. It is usually assumed that the sooner a soft task is served the better, without distinction among soft tasks. In many contexts, however, differentiating among soft tasks gives an additional degree of flexibility as it allows allocating the processing resources more efficiently. This is the case, for example, in videoconference applications where audio streams are considered more important than the video ones. In order to capture the relative significance of soft tasks and how the quality of results is affected when missing a soft deadline, we use utility functions. Utility functions were first suggested by Locke [12] to represent importance and criticality of tasks.

Utility-based scheduling has been addressed before [2]. For instance, imprecise computation techniques [15], [7] are examples of utility-based scheduling. Such techniques consider tasks as composed by a mandatory and an optional part. The mandatory subtask must be completed by the deadline of the task. The optional subtask can be left incomplete at the expense of the quality of results. The problem is thus finding a schedule that maximizes the total length of the executed portions of optional subtasks. For many applications, however, it is not possible to identify the mandatory and optional parts of a task. We consider tasks without optional part and, once they are started, tasks run until completion. Our utility function is expressed as a function of the completion time of the task (and not its execution time as in the case of imprecise computation). Other value-based approaches include best-effort techniques [12], the QoS-based resource allocation model [13], and Time-Value-Function scheduling [4]. The latter also uses the completion time

as argument of the utility function, but does not consider hard tasks in the system. It provides an $O(n^3)$ on-line heuristic, which might still be impractical, and assumes fixed task execution times. We, on the contrary, consider that the execution time is variable within an interval and unknown beforehand. This is actually the reason why a single static schedule can be too pessimistic.

We take into consideration the fact that the actual execution time of a task is rarely its worst case execution time (WCET). We thus use the expected or mean duration of tasks when evaluating the utility functions associated to soft tasks. However, we consider the maximum duration of tasks for ensuring that all hard time constraints are met in every possible scenario. Earlier work generally uses only the WCET for scheduling which leads to an excessive degree of pessimism (Abeni and Buttazzo [1] do use mean values for serving soft tasks and WCET for guaranteeing hard deadlines though).

In the context of the systems we are addressing in this report, *off-line* scheduling refers to finding at design-time one schedule that maximizes the sum of individual utilities by soft tasks and at the same time guarantees hard deadlines (we have addressed such a problem in an earlier report [5]). *On-line* scheduling refers to computing at run-time, every time a task finishes, a new schedule for the remaining tasks such that it makes the total utility maximum and also guarantees no hard deadline miss, but taking into consideration the actual execution times of tasks already completed. On the one hand, having one single schedule obtained off-line might be too pessimistic as information about actual execution times is not exploited. On the other hand, due to the complexity of the problem, the overhead of computing schedules on-line is unacceptable. In order to overcome these drawbacks, we propose to analyze the system and compute a set of schedules at design-time, and let the decision of which of them is to be followed be taken at run-time. Thus the problem we address in this report is that of *quasi-static* scheduling for hard/soft real-time systems.

The rest of this report is structured as follows. Section 2 introduces definitions and notations used along the report. We present an example in Section 3 motivating the addressed in this report. In Section 4 we precisely formulate the problem we are solving. We present an exact method (Section 5) as well as a number of heuristics (Section 6) for solving the problem. A system corresponding to a real-life application is studied in Section 7. Finally, some conclusions are drawn in Section 8.

2 Preliminaries

We consider that the system is represented by a directed acyclic graph $G = (T, E)$ where nodes correspond to tasks and data dependencies are captured by the graph edges. Throughout this paper we assume that all the tasks of the system are mapped onto a single processor.

The actual execution time of a task t at a certain activation of the system, denoted $|t|$, lies in the interval bounded by the minimum duration $l(t)$ and the maximum duration $m(t)$ of the task, i.e. $l(t) \leq |t| \leq m(t)$. The expected duration $e(t)$ of a task t is the mean value of the possible execution times of the task. In the simple case that the execution time is uniformly distributed over the interval $[l(t), m(t)]$, we have $e(t) = (l(t) + m(t))/2$. For an arbitrary continuous execution time probability distribution $f(\tau)$, the expected duration is given by $e(t) = \int_{l(t)}^{m(t)} \tau f(\tau) d\tau$.

We define a *schedule* as the execution order for the tasks in the system. We assume a single-rate semantics, that is, each task is executed exactly once for every activation of the system. Thus a schedule is a bijection $\sigma : T \rightarrow \{1, 2, \ldots, |T|\}$. We use the notation $\sigma = t_1 t_2 \ldots t_n$ as shorthand for $\sigma(t_1) = 1, \sigma(t_2) = 2, \ldots, \sigma(t_n) = |T|$. We assume that the system is activated periodically[1].

In this context, a schedule does not provide the starting time for tasks, only their execution sequence. Thus, for the schedule $\sigma = t_1 t_2 \ldots t_n$, task t_1 will start when the system is activated and task t_{i+1}, $1 \leq i < n$, will start executing as soon as task t_i has finished. For a given schedule, the completion time of a task t_i is denoted τ_i. In the sequel, the starting and completion times that we use are relative to the system activation instant. For example, according to the schedule $\sigma = t_1 t_2 \ldots t_n$, t_1 starts executing at time 0 and its completion time is $\tau_1 = |t_1|$, the completion time of t_2 is $\tau_2 = \tau_1 + |t_2|$, and so forth.

The tasks that make up a system can be classified as non-real-time, hard, or soft. Non-real-time tasks are neither hard nor soft, and have no timing constraints, though they may influence other hard or soft tasks through precedence constraints as defined by the task graph $G = (T, E)$. Both hard and soft tasks have deadlines. A hard deadline $d(h)$ is the time by which a hard task $h \in T$ *must* be completed, otherwise the integrity of the system is jeopardized. A soft deadline $d(s)$ is the time by which a soft task $s \in T$ *should* be completed. Lateness of soft tasks is acceptable though it decreases

[1]Handling tasks with different periods is possible by generating several instances of the tasks and building a graph that corresponds to a set of tasks as they occur within a time period that is equal the least common multiple of the periods of the involved tasks. In this case, the release time of certain tasks must be taken into account.

the quality of results. In order to capture the relative importance among soft tasks and how the quality of results is affected when missing a soft deadline, we use a non-increasing utility function $u_j(\tau_j)$ for each soft task s_j. Typical utility functions are depicted in Figure 1.

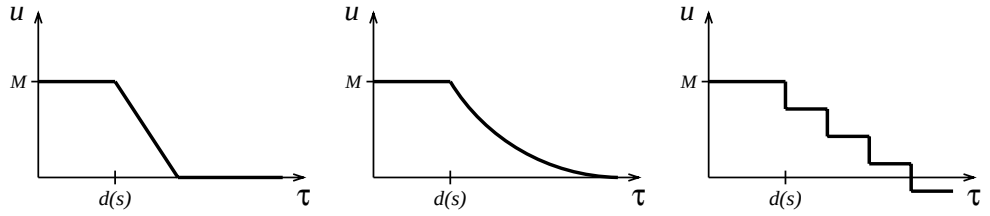


Figure 1: Typical utility functions for soft tasks

In the problem of quasi-static scheduling, we aim to find off-line a set of schedules and the relation among them, that is, the conditions under which the scheduler decides on-line to switch from one schedule to another. There might be one or more *switching points* associated to a schedule, which define when to switch to another schedule. A switching point is characterized by a task and a time interval. For example, the switching point $\langle t_i; (a, b]\rangle$ associated to σ indicates that when the task t_i in σ finishes and its completion time is $a < \tau_i \leq b$, another schedule σ' must be followed as execution order for the remaining tasks.

3 Motivational Example

Let us consider a system that has five tasks t_1, t_2, t_3, t_4, and t_5, with data dependencies as shown in Figure 2. The minimum and maximum duration of every task are given in Figure 2 in the form $[l(t), m(t)]$. We assume in this example that the execution time of every task t is uniformly distributed over the interval $[l(t), m(t)]$. The only hard task in the system is t_4 and its deadline is $d(t_4) = 30$. Tasks t_2 and t_3 are soft, their deadlines are $d(t_2) = 9$ and $d(t_3) = 18$, and their utility functions are given, respectively, by:

$$u_2(\tau_2) = \begin{cases} 3 & \text{if } \tau_2 \leq 9, \\ \dfrac{9}{2} - \dfrac{\tau_2}{6} & \text{if } 9 \leq \tau_2 \leq 27, \\ 0 & \text{if } \tau_2 \geq 27. \end{cases} \qquad u_3(\tau_3) = \begin{cases} 2 & \text{if } \tau_3 \leq 18, \\ 8 - \dfrac{\tau_3}{3} & \text{if } 18 \leq \tau_3 \leq 24, \\ 0 & \text{if } \tau_3 \geq 24. \end{cases}$$

We initially want to find the schedule that, among all schedules that respect the hard constraints in the worst case, maximizes the total utility (sum of individual contributions evaluated at each soft task's completion time) in the case when tasks last their expected duration. The problem of

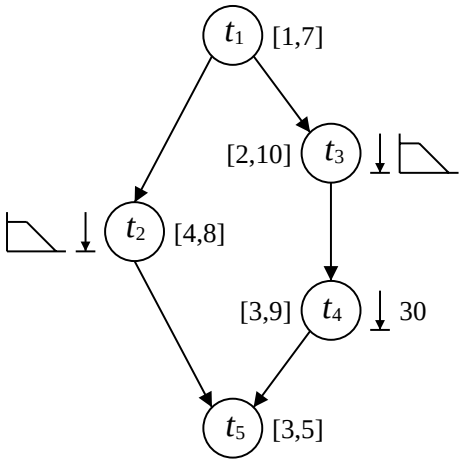


Figure 2: Motivational example

computing one such optimal schedule has been proved **NP**-complete [5]. For the system of Figure 2, such a schedule is $\sigma = t_1 t_3 t_4 t_2 t_5$.

Let us now consider the situation in which all tasks but t_3 take their maximum duration, that is $|t_i| = m(t_i)$ for $i = 1, 2, 4, 5$, and the duration of task t_3 is $|t_3| = 3$. The schedule that, under these particular conditions, maximizes the total utility is $\sigma^{\times} = t_1 t_2 t_3 t_4 t_5$. Such a total utility is $U^{\times} = u_2(15) + u_3(18) = 2 + 2 = 4$. Compare with the total utility $U = u_2(27) + u_3(10) = 0+2 = 2$ obtained when using the schedule $\sigma = t_1 t_3 t_4 t_2 t_5$. Although $\sigma^{\times}$ maximizes the total utility when $|t_1| = 7$, $|t_2| = 8$, $|t_3| = 3$, $|t_4| = 9$, and $|t_5| = 5$, it does not guarantee that hard deadlines are met in all situations. If t_3 took 7 time units instead of 3, the completion time of task t_4 would be $\tau_4 = 31$ and therefore its hard deadline would be missed. Since we cannot predict the execution time for t_3 (we only know it is bounded in the interval $[2, 10]$), selecting $\sigma^{\times} = t_1 t_2 t_3 t_4 t_5$ as the schedule for our system implies potential hard deadline misses.

Nonetheless we can do much better than just selecting $\sigma = t_1 t_3 t_4 t_2 t_5$ as the unique schedule for our system. Though σ ensures that hard constraints are always satisfied, it can still be pessimistic as the actual execution times might be far off from those used when computing σ. Suppose we initially choose σ as schedule. Assume that task t_1 executes and its duration is $|t_1| = 7$, and then, following the order in σ, task t_3 executes and its duration is $|t_3| = 3$. At this point we know that the completion time of task t_3 is $\tau_3 = 10$. Taking advantage of the fact that $\tau_3 = 10$, we can compute the schedule (that has $t_1 t_3$ as prefix) which maximizes the total utility (considering the actual execution times of the tasks t_1 and t_3—already completed—and the

expected duration for t_2, t_4, and t_5—remaining tasks—when evaluating the utility functions of soft tasks) and guarantees no hard deadline miss. Such a schedule is $\sigma' = t_1 t_3 t_2 t_4 t_5$. In the situation $|t_1| = 7$, $|t_2| = 8$, $|t_3| = 3$, $|t_4| = 9$, and $|t_5| = 5$, σ' yields a total utility $U' = u_2(18) + u_3(10) = 1.5 + 2 = 3.5$, which is greater than the one given when using $\sigma = t_1 t_3 t_4 t_2 t_5$ ($U = 2$) and less than the one given when using $\sigma^\times = t_1 t_2 t_3 t_4 t_5$ ($U^\times = 4$). However, as opposed to $\sigma^\times$, σ' does guarantee completion of every hard task before or at its deadline, because the decision to follow σ' is taken after t_3 has been executed and its completion time is thus known.

Following the idea that we have just described, one could think of starting off the execution of tasks in the system as given by an initial schedule. Then every time a task finishes we measure its completion time and compute a new schedule, i.e. the execution order for the remaining tasks, that optimizes the total utility for the new conditions while guaranteeing that hard deadlines are met. This approach would give the best results in terms of total utility. However, since it requires the on-line computation of an optimal schedule, a problem which is **NP**-complete [5], its overhead is unacceptable.

A better approach is to compute *off-line* a number of schedules and schedule-switching points. Then one of the precomputed schedules is selected *on-line* based on the actual execution times. Hence the only overhead at run-time is the selection of a schedule, which is very cheap because it requires a simple comparison between the time given by a switching point and the actual completion time. The most relevant question in such a *quasi-static* approach is thus how to compute at design-time the schedules and switching points such that they deliver the highest quality (utility).

For the system shown in Figure 2, for instance, we can define a switching point $\langle t_3; [3, 13] \rangle$ associated to $\sigma = t_1 t_3 t_4 t_2 t_5$, that together with the schedule $\sigma' = t_1 t_3 t_2 t_4 t_5$, works as follows: the system starts executing according to the schedule σ, i.e. task t_1 runs followed by t_3; when t_3 finishes, the completion time τ_3 is compared to that of the switching point and if $3 \leq \tau_3 \leq 13$ the remaining tasks execute following the schedule σ', else the execution order continues according to σ.

The set of schedules $\{\sigma, \sigma'\}$ as explained above outperforms the solution of a single schedule σ: while the scheme $\{\sigma, \sigma'\}$ guarantees satisfaction of all hard deadlines, it yields a total utility which is greater than the one given by σ in 83.3% of the cases (this figure can be obtained analytically by considering the execution time probability distributions).

4 Problem Formulation

A system is defined by: a set T of tasks; a directed acyclic graph $G = (T, E)$ defining precedence constraints for the tasks; a minimum duration $l(t)$ for each task $t \in T$; a maximum duration $m(t)$ for each task $t \in T$; an expected duration[2] $e(t)$ for each task $t \in T$ ($l(t) \leq e(t) \leq m(t)$); a subset $H \subseteq T$ of hard tasks; a deadline $d(h)$ for each hard task $h \in H$; a subset $S \subseteq T$ of soft tasks ($S \cap H = \emptyset$); a non-increasing utility function $u_j(\tau_j)$ for each soft task $s_j \in S$ (τ_j is the completion time of s_j).

On-Line Scheduler: The following is the problem that the on-line scheduler would solve before the activation of the system and every time a task completes, in order to give the best results in terms of total utility (in the sequel, this problem will be referred to as the *one-schedule problem*):

> Find a schedule σ (a bijection $\sigma : T \to \{1, 2, \ldots, |T|\}$) that maximizes $U = \sum_{s_j \in S} u_j(\tau_j^e)$, where τ_j^e is the expected completion time[3] of task s_j, subject to: $\tau_i^m \leq d(h_i)$ for all $h_i \in H$, where τ_i^m is the maximum completion time[4] of task h_i; $\sigma(t) < \sigma(t')$ for all $\langle t, t' \rangle \in E$; σ has σ_x as prefix, where σ_x is the order of the already executed tasks.

Considering an ideal case, in which the on-line scheduler executes in zero time, for any possible set of execution times $|t_1|, |t_2|, \ldots, |t_n|$ (which are not known beforehand), the utility $U_{\{|t_i|\}}$ produced by the on-line scheduler is maximal and will be denoted $U^{max}_{\{|t_i|\}}$.

Due to the complexity of the problem that the on-line scheduler solves after completing every task, such a solution is infeasible in practice. We therefore propose to compute a number of schedules and switching points at design-time aiming to match the total utility produced by the ideal on-line scheduler. This leaves for run-time only the decision of selecting one of the precomputed schedules, which is done by the so-called *quasi-static scheduler*. The problem we concentrate on in the rest of this report is formulated as follows:

---

[2] Mean value of the possible execution times.

[3] τ_j^e is given by

$$\tau_j^e = \begin{cases} \mathsf{e}_j & \text{if } \sigma(t_j) = 1, \\ \tau_k^e + \mathsf{e}_j & \text{if } \sigma(t_j) = \sigma(t_k) + 1. \end{cases}$$

where $\mathsf{e}_j = |t_j|$ if t_j has already been executed, else $\mathsf{e}_j = e(t_j)$.

[4] τ_i^m is given by

$$\tau_i^m = \begin{cases} \mathsf{m}_i & \text{if } \sigma(t_i) = 1, \\ \tau_k^m + \mathsf{m}_i & \text{if } \sigma(t_i) = \sigma(t_k) + 1. \end{cases}$$

where $\mathsf{m}_i = |t_i|$ if t_i has already been executed, else $\mathsf{m}_i = m(t_i)$.

MULTIPLE-SCHEDULES PROBLEM: Find a set of schedules and switching points such that, for any combination of execution times $|t_1|, |t_2|, \ldots, |t_n|$, the quasi-static scheduler yields a total utility $U_{\{|t_i|\}}$ that is equal to the one produced by the ideal on-line scheduler $U^{max}_{\{|t_i|\}}$ and, at the same time, guarantees satisfaction of hard deadlines.

5 Optimal Set of Schedules and Switching Points

In this section we propose a systematic method for finding the optimal set of schedules and switching points as required by the multiple-schedules problem.

We start by taking the *basis* schedule σ, i.e. the one that maximizes $\sum_{s_j \in S} u_j(\tau^e_j)$ considering that no task has yet been executed. Let us assume that $\sigma(t_1) = 1$, i.e. t_1 is the first task of σ. For each one of the schedules σ_i that start with t_1 and satisfy the precedence constraints, we express the total utility $U_i(\tau_1)$ as a function of the completion time τ_1 of task t_1 for $l(t_1) \leq \tau_1 \leq m(t_1)$. When computing U_i we consider $|t| = e(t)$ for all $t \in T \setminus \{t_1\}$ (expected duration for the remaining tasks). Then, for each possible σ_i, we analyze the schedulability of the system, that is, which values of the completion time τ_1 imply potential hard deadline misses when σ_i is followed. For this analysis we consider $|t| = m(t)$ for all $t \in T \setminus \{t_1\}$ (maximum duration for the remaining tasks). We introduce the auxiliary function $\hat{U}_i$ such that $\hat{U}_i(\tau_1) = -\infty$ if following σ_i, after t_1 has completed at τ_1, does not guarantee the hard deadlines, else $\hat{U}_i(\tau_1) = U_i(\tau_1)$.

Once we have computed all the functions $\hat{U}_i(\tau_1)$, we may determine which σ_i yields the maximum total utility at which instants in the interval $[l(t_1), m(t_1)]$. We get thus the interval $[l(t_1), m(t_1)]$ partitioned into subintervals and, for each one of these, the schedule that maximizes the total utility and guarantees satisfaction of hard deadlines. In this way we obtain the schedules to be followed after finishing t_1 depending on the completion time τ_1.

For each one of the obtained schedules, we repeat the process, this time computing $\hat{U}_j$'s as a function of the completion time of the second task in the schedule and for the interval in which this second task may finish. Then the process is similarly repeated for the third element of the new schedules, and then for the fourth, and so on. In this manner we obtain the best *tree* of schedules and switching points.

Let us consider the example discussed in Section 3 and shown in Figure 2. The basis schedule is in this case $\sigma = t_1 t_3 t_4 t_2 t_5$. Due to the data dependencies, there are three possible schedules that start with t_1, namely

$\sigma_a = t_1t_2t_3t_4t_5$, $\sigma_b = t_1t_3t_2t_4t_5$, and $\sigma_c = t_1t_3t_4t_2t_5$. We want to compute the corresponding functions $U_a(\tau_1)$, $U_b(\tau_1)$, and $U_c(\tau_1)$, $1 \leq \tau_1 \leq 7$, considering the expected duration for t_2, t_3, t_4, and t_5. For example, $U_b(\tau_1) = u_2(\tau_1 + e(t_3) + e(t_2)) + u_3(\tau_1 + e(t_3)) = u_2(\tau_1 + 12) + u_3(\tau_1 + 6)$. We get the following functions:

$$U_a(\tau_1) = \begin{cases} 5 & \text{if } 1 \leq \tau_1 \leq 3, \\ \frac{11}{2} - \frac{\tau_1}{6} & \text{if } 3 \leq \tau_1 \leq 6, \\ \frac{15}{2} - \frac{\tau_1}{2} & \text{if } 6 \leq \tau_1 \leq 7. \end{cases} \qquad U_b(\tau_1) = \frac{9}{2} - \frac{\tau_1}{6} \quad \text{if } 1 \leq \tau_1 \leq 7.$$

$$U_c(\tau_1) = \frac{7}{2} - \frac{\tau_1}{6} \quad \text{if } 1 \leq \tau_1 \leq 7.$$

The functions $U_a(\tau_1)$, $U_b(\tau_1)$, and $U_c(\tau_1)$, as given above, are shown in Figure 3(a). Now, for each one of the schedules σ_a, σ_b, and σ_c, we determine the latest completion time τ_1 that guarantees meeting hard deadlines when that schedule is followed. For example, if the execution order given by $\sigma_a = t_1t_2t_3t_4t_5$ is followed and the remaining tasks take their maximum duration, the hard deadline $d(t_4)$ is met only when $\tau_1 \leq 3$. This is because $\tau_4 = \tau_1 + m(t_2) + m(t_3) + m(t_4) = \tau_1 + 27$ in the worst case and therefore $\tau_4 \leq d(t_4) = 30$ iff $\tau_1 \leq 3$. A similar analysis shows that σ_b guarantees meeting the hard deadline only when $\tau_1 \leq 3$ while σ_c guarantees the hard deadline for any completion time τ_1 in the interval $[1, 7]$. Thus we get the auxiliary functions as given below and depicted in Figure 3(b):

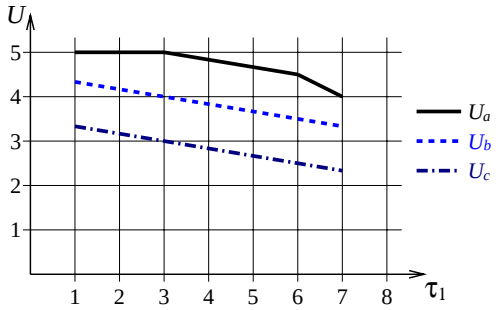
$$\hat{U}_a(\tau_1) = \begin{cases} 5 & \text{if } 1 \leq \tau_1 \leq 3, \\ -\infty & \text{if } 3 < \tau_1 \leq 7. \end{cases} \qquad \hat{U}_b(\tau_1) = \begin{cases} \frac{9}{2} - \frac{\tau_1}{6} & \text{if } 1 \leq \tau_1 \leq 3, \\ -\infty & \text{if } 3 < \tau_1 \leq 7. \end{cases}$$

$$\hat{U}_c(\tau_1) = \frac{7}{2} - \frac{\tau_1}{6} \quad \text{if } 1 \leq \tau_1 \leq 7.$$

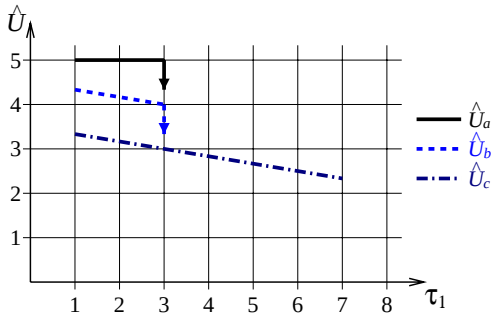
From the graphic of Figure 3(b) we conclude that $\sigma_a = t_1t_2t_3t_4t_5$ yields the highest total utility when t_1 completes in the subinterval $[1, 3]$ still guaranteeing the hard deadline, and that $\sigma_c = t_1t_3t_4t_2t_5$ yields the highest total utility when t_1 completes in the subinterval $(3, 7]$ also guaranteeing the hard deadline in this case.

A similar procedure is followed, first for σ_a and then for σ_c, considering the completion time of the second task in these schedules. Let us take $\sigma_a = t_1t_2t_3t_4t_5$. We must analyze the legal schedules that start with t_1t_2. However, since there is only one such schedule, there is no need to continue along the branch originated from σ_a.

Let us take $\sigma_c = t_1t_3t_4t_2t_5$. We make an analysis of the possible schedules σ_j that start with t_1t_3 ($\sigma_d = t_1t_3t_2t_4t_5$ and $\sigma_e = t_1t_3t_4t_2t_5$) and for each



(a) $U_i(\tau_1)$



(b) $\hat{U}_i(\tau_1)$

Figure 3: $U_i(\tau_1)$ and $\hat{U}_i(\tau_1)$, $1 \leq \tau_1 \leq 7$, for the example of Figure 2

of these we obtain $U_j(\tau_3)$, $5 < \tau_3 \leq 17$ (recall that: σ_c is followed when $3 < \tau_1 \leq 7$; $2 \leq |t_3| \leq 10$). The corresponding functions, when considering expected duration for t_2, t_4, and t_5, are:

$$U_d(\tau_3) = \frac{7}{2} - \frac{\tau_3}{6} \quad \text{if } 5 < \tau_3 \leq 17. \qquad U_e(\tau_3) = \begin{cases} \frac{5}{2} - \frac{\tau_3}{6} & \text{if } 5 < \tau_3 \leq 15, \\ 0 & \text{if } 15 \leq \tau_3 \leq 17. \end{cases}$$

The above $U_d(\tau_3)$ and $U_e(\tau_3)$ are shown in Figure 4(a). Note that these utility functions do not include the contribution by the soft task t_3 because this contribution $u_3(\tau_3)$ is the same for both σ_d and σ_e and therefore it is not relevant when differentiating between $\hat{U}_d(\tau_3)$ and $\hat{U}_e(\tau_3)$. After the hard deadlines analysis, the auxiliary utility functions under consideration become:

$$\hat{U}_d(\tau_3) = \begin{cases} \frac{7}{2} - \frac{\tau_3}{6} & \text{if } 5 < \tau_3 \leq 13, \\ -\infty & \text{if } 13 < \tau_3 \leq 17. \end{cases} \qquad \hat{U}_e(\tau_3) = \begin{cases} \frac{5}{2} - \frac{\tau_3}{6} & \text{if } 5 < \tau_3 \leq 15, \\ 0 & \text{if } 15 \leq \tau_3 \leq 17. \end{cases}$$

From the Figure 4(b) we conclude that if task t_3 completes in the interval $(5, 13]$, the schedule $\sigma_d = t_1 t_3 t_2 t_4 t_5$ should be followed while if t_3 completes in the interval $(13, 17]$, $\sigma_e = t_1 t_3 t_4 t_2 t_5$ should be followed. The process terminates at this point since there is no other scheduling alternative after completing the third task of either σ_d or σ_e.

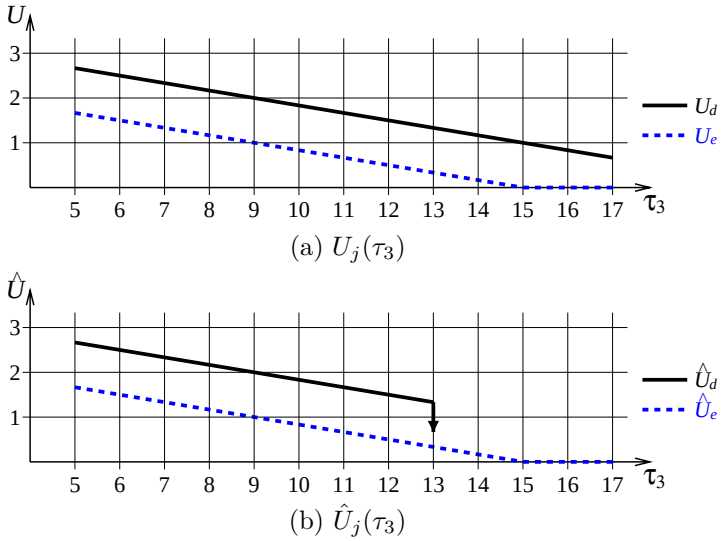


Figure 4: $U_j(\tau_3)$ and $\hat{U}_j(\tau_3)$, $5 < \tau_3 \leq 17$, for the example of Figure 2

At the end, renaming σ_a and σ_d, we get the set of schedules $\{\sigma = t_1t_3t_4t_2t_5, \sigma' = t_1t_2t_3t_4t_5, \sigma'' = t_1t_3t_2t_4t_5\}$ that works as follows (see Figure 5): once the system is activated, it starts following the schedule σ; when t_1 is finished, its completion time τ_1 is read, and if $\tau_1 \leq 3$ the schedule is switched to σ' for the remaining tasks, else the execution order continues according to σ; when t_3 finishes, while σ is the followed schedule, its completion time τ_3 is compared with the time point 13: if $\tau_3 \leq 13$ the remaining tasks are executed according to σ'', else the schedule σ is followed.

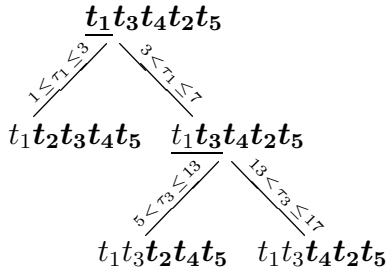


Figure 5: Optimal tree of schedules and switching points

It is not difficult to show that the procedure we have described finds a set of schedules and switching points that produces the same utility as the on-line scheduler defined in Section 4. Both the on-line scheduler and the set computed by the above method start off the system following the same schedule (the basis schedule). Upon completion of every task, the on-line scheduler

computes a new schedule that maximizes the total utility when taking into account the actual execution times for the already completed tasks and the expected durations for the tasks yet to be executed. Our procedure analyzes off-line, beginning with the first task in the basis schedule, the sum of utilities by soft tasks as a function of the completion time of such first task, for each one of the possible schedules starting with that task. For computing the utility as a function of the completion time, our procedure considers expected duration for the remaining tasks. In this way, the procedure determines the schedule that maximizes the total utility at every possible completion time. The process is likewise repeated for the second element of the new schedules, and then the third, and so forth. Thus our procedure solves *symbolically* the optimization problem for a set of completion times, one of which corresponds to the particular instance solved by the on-line scheduler. Thus, having the tree of schedules and switching points computed in this way, the schedule selected at run-time by the quasi-static scheduler produces a total utility that is equal to that of the ideal on-line scheduler, for any set of execution times.

In the previous discussion about the method for finding the optimal set of schedules and switching points, for instance when t_1 is the first task in the basis schedule, we mentioned that we considered each one of the potentially $|T \setminus \{t_1\}|!$ schedules σ_i that start with t_1 in order to obtain the utilities U_i as a function of the completion time τ_1. This can actually be done more efficiently by considering $|H \cup S \setminus \{t_1\}|!$ schedules σ_i instead of $|T \setminus \{t_1\}|!$, that is, by considering the permutations of hard and soft tasks instead of the permutations of all tasks. The rationale is that the best schedule, for a given permutation HS of hard and soft tasks, is obtained when the hard and soft tasks are set in the schedule as early as possible respecting the order given by HS.

A simple proof of the fact that by setting hard and soft tasks as early as possible according to the order given by HS we get the best schedule for HS is as follows: let σ be the schedule that respects the order of hard and soft tasks given by HS (that is, $1 \leq i < j \leq |\mathsf{HS}| \Rightarrow \sigma(\mathsf{HS}_{[i]}) < \sigma(\mathsf{HS}_{[j]})$)[5] and such that hard and soft tasks are set as early as possible (that is, for every schedule σ', different from σ, that obeys the order of hard and soft tasks given by HS, $\sigma'(\mathsf{HS}_{[i]}) > \sigma(\mathsf{HS}_{[i]})$ for some $1 \leq i \leq |\mathsf{HS}|$). Take one such σ'. For at least one task $hs_j \in H \cup S$ it holds $\sigma'(hs_j) > \sigma(hs_j)$. We study two situations: a) $hs_j \in S$: in this case ${\tau'}_j^e > \tau_j^e$ (${\tau'}_j^e$ is the completion time of hs_j when we use σ' as schedule while τ_j^e is the completion time of hs_j when σ is used as schedule, considering in both cases expected duration for the remaining tasks). Thus $u_j({\tau'}_j^e) \leq u_j(\tau_j^e)$ because utility functions for

[5] $\mathsf{HS}_{[i]}$ denotes the i-th task in the array representing the permutation HS.

soft tasks are non-increasing. Consequently $\hat{U}'(\tau) \leq \hat{U}(\tau)$ for every possible completion time τ, where $\hat{U}'(\tau)$ and $\hat{U}(\tau)$ correspond, respectively, to σ' and σ. b) $hs_j \in H$: in this case $\tau'^m_j > \tau^m_j$ (τ'^m_j is the completion time of hs_j when we use σ' as schedule while τ^m_j is the completion time of hs_j when σ is used as schedule, considering in both cases maximum duration for the remaining tasks). Thus there exists some $\tau^\times$ for which σ guarantees meeting hard deadlines whereas σ' does not. Recall that we include the information about potential hard deadline misses in the form $\hat{U}'(\tau) = -\infty$ if following σ', after completing the current task at τ, implies potential hard deadline violations. Accordingly $\hat{U}'(\tau) \leq \hat{U}(\tau)$ for every possible completion time τ. Hence we conclude that every schedule σ', which respects the order for hard and soft tasks given by HS, yields a function $\hat{U}'(\tau)$ such that $\hat{U}'(\tau) \leq \hat{U}(\tau)$ for every τ, and therefore σ is the best schedule for the given permutation HS.

The pseudocode of the algorithm for finding the optimal set of schedules and switching points is presented in Figures 6 and 7. First of all, if there is no basis schedule that guarantees satisfaction of all hard deadlines, the system is not schedulable and therefore the multiple-schedules problem has no solution. IsValidPerm(HS) is used in order to examine if a permutation HS defines a feasible schedule: if there exists a path from the task $\mathsf{HS}_{[j]}$ to the task $\mathsf{HS}_{[i]}$, $j > i$, HS is not valid. The procedure BestSchedule(HS, σ, n) returns the schedule that agrees with σ up to the n-th position and for which the remaining hard and soft tasks are set as early as possible obeying the order given by HS. The method AddSubtree(Ψ^k, σ, n, I^k_n) adds Ψ^k to Ψ, i.e. the root σ^k of Ψ^k is a child of σ, in such a way that upon completing the n-th task of σ in the time interval I^k_n, the schedule σ^k is followed for the remaining tasks. Observe that the tree of schedules obtained using the algorithm OptimalTree might contain nodes σ with a single child σ_k (when the whole interval I^n is covered by a single schedule σ_k). In these cases, since $\sigma = \sigma_k$, the node σ can be removed, so that σ_k becomes a child of the parent of σ, without changing the behavior.

Algorithm OptimalTree()
output: The best tree Ψ of schedules and switching points

begin
  $\sigma :=$ *basis* schedule
  $\Psi :=$ OptimalTree($\sigma, 1$)
end

Figure 6: Algorithm OptimalTree()

Algorithm OPTIMALTREE(σ, n)
input: A schedule σ and a positive integer n
output: The best tree Ψ of schedules to follow after completing the n-th task in σ

```
begin
  set σ as root of Ψ
  A := {t ∈ T | σ(t) ≤ n}
  if |H ∪ S \ A| > 1 then
    for i ← 1, 2, ..., |H ∪ S \ A|! do
      if IsValidPerm(HS_i) then
        σ_i := BestSchedule(HS_i, σ, n)
        compute Û_i(τ_n)
      end if
    end for
    partition the interval I^n of possible τ_n into subintervals
    I^n_1, I^n_2, ..., I^n_M s.t. σ_k makes Û_k(τ_n) maximal in I^n_k
    for k ← 1, 2, ..., M do
      Ψ_k := OptimalTree(σ_k, n + 1)
      AddSubtree(Ψ_k, σ, n, I^n_k)
    end for
  end if
end
```

Figure 7: Algorithm OPTIMALTREE(σ, n)

In the procedure for finding the optimal set of schedules and switching points, we partition the interval of possible completion times τ_i for a task t_i into subintervals which define the switching points and schedules to follow after executing t_i. The interval-partitioning step can be done in $O((|H| + |S|)!)$ time. Though a time complexity $O((|H| + |S|)!)$ in this step is a great improvement with respect to $O(|T|!)$, the multiple-schedules problem is intractable. Moreover, the inherent nature of the problem (finding a tree of schedules) makes it such that it requires exponential-time and exponential-memory solutions, even when using a polynomial-time heuristic in the interval-partitioning step. An additional problem is that, even if we can afford the time and memory budget for computing the optimal tree of schedules (as this is done off-line), the memory constraints of the target system still impose a limit on the size of the tree, therefore a suboptimal set of schedules must be chosen to fit in the system memory. These issues are addressed in Section 6.

Nonetheless, despite its complexity, the procedure to compute the optimal

set of schedules and switching points as outlined above has also theoretical relevance: it shows that an infinite space of execution times (the execution time of task t can be any value in the interval $[l(t), m(t)]$) might be covered optimally by a finite number of schedules, albeit it may be a very large number. This is so because, when partitioning an interval I^n of possible τ_n, the number of subintervals I^n_k is finite.

The set of schedules is stored in memory as an ordered tree. Upon completing a task, the cost of selecting at run-time the schedule for the remaining tasks is at most $O(logN)$ where N is the maximum number of children that a node has in the tree of schedules. Such cost can be included by augmenting accordingly the maximum duration of tasks.

6 Heuristic Methods and Experimental Evaluation

In this section we propose several heuristics that address different complexity dimensions of the multiple-schedules problem, namely the interval-partitioning step and the exponential growth of the tree size.

6.1 Interval Partitioning

In this section we discuss methods to avoid the computation, in the interval-partitioning step, of $\hat{U}_i(\tau_n)$ for all permutations of the remaining tasks that define possible schedules (loop **for** $i \leftarrow 1, 2, \ldots, |H \cup S \setminus A|!$ **do** in Figure 7).

The first heuristic starts similar to the algorithm of Figure 6, but instead of OptimalTree$(\sigma, 1)$, it calls LimTree$(\sigma, 1)$ as presented in Figure 8. We obtain solutions σ_{L} and σ_{U} to the *one-schedule problem* (see Section 4), respectively, for the lower and upper limits τ_{L} and τ_{U} of the interval I^n of possible completion times τ_n. OptSchedule(σ, n, τ) returns the schedule that agrees with σ up to the n-th position and maximizes the total utility, considering that the n-th task completes precisely at τ. Then we compute $\hat{U}_{\mathsf{L}}(\tau_n)$ and $\hat{U}_{\mathsf{U}}(\tau_n)$ and partition I^n considering only these two (avoiding thus computing $\hat{U}_i(\tau_n)$ corresponding to the possible schedules σ_i defined by permutations of the remaining tasks). For the example discussed in Sections 3 and 5, when partitioning the interval $I^1 = [1, 7]$ of possible completion times of the first task in the basis schedule, LimTree solves the one-schedule problem for $\tau_{\mathsf{L}} = 1$ and $\tau_{\mathsf{U}} = 7$, whose solutions are $\sigma_{\mathsf{L}} = t_1 t_2 t_3 t_4 t_5$ and $\sigma_{\mathsf{U}} = t_1 t_3 t_4 t_2 t_5$ respectively. Then $\hat{U}_{\mathsf{L}}(\tau_1)$ and $\hat{U}_{\mathsf{U}}(\tau_1)$ are computed as described in Section 5 and only these two are used for partitioning the interval. Referring to Figure 3(b), $\hat{U}_{\mathsf{L}} = \hat{U}_a$ and $\hat{U}_{\mathsf{U}} = \hat{U}_c$, and in this case LimTree gives the

same result as the optimal algorithm. The rest of the procedure is repeated in a similar way as explained in Section 5.

Algorithm LimTree(σ, n)
input: A schedule σ and a positive integer n
output: The tree Ψ of schedules to follow after completing the n-th task in σ

```
begin
  set σ as root of Ψ
  A := {t ∈ T | σ(t) ≤ n}
  if |H ∪ S \ A| > 1 then
    τ_L := lower limit of the interval I^n of possible τ_n
    σ_L := OptSchedule(σ, n, τ_L)
    compute Û_L(τ_n)
    τ_U := upper limit of the interval I^n of possible τ_n
    σ_U := OptSchedule(σ, n, τ_U)
    compute Û_U(τ_n)
    partition I^n into subintervals I^n_1, I^n_2, ..., I^n_M s.t. σ_k makes
    Û_k(τ_n) maximal in I^n_k
    for k ← 1, 2, ..., M do
      Ψ_k := LimTree(σ_k, n + 1)
      AddSubtree(Ψ_k, σ, n, I^n_k)
    end for
  end if
end
```

Figure 8: Algorithm LimTree(σ, n)

The second of the proposed heuristics, named LimCmpTree (Figure 9), is based on the same ideas as LimTree, that is, computing only $\hat{U}_{\mathsf{L}}(\tau_n)$ and $\hat{U}_{\mathsf{U}}(\tau_n)$ that correspond to schedules σ_{L} and σ_{U} which are in turn solutions to the one-schedule problem for τ_{L} and τ_{U}, respectively. The difference lies in that, while constructing the tree of schedules to follow after completing the n-th task in σ, if the n+1-th task of the schedule σ_k (the one that yields the highest utility in the subinterval I^n_k) is the same as the $n+1$-th task of the current schedule σ (**if** $\sigma_k^{-1}(n+1) = \sigma^{-1}(n+1)$ in Figure 9), the schedule σ continues being followed instead of adding σ_k to the tree. This leads to a tree with fewer nodes.

In both LimTree and LimCmpTree we must solve the one-schedule problem through OptSchedule(σ, n, τ). The one-schedule problem is **NP**-complete and we have proposed an optimal algorithm as well as different heuristics for it [5]. In the experimental evaluation of the heuristics proposed

Algorithm LIMCMPTREE(σ, n)
input: A schedule σ and a positive integer n
output: The tree Ψ of schedules to follow after completing the n-th task in σ

```
begin
  set σ as root of Ψ
  A := {t ∈ T | σ(t) ≤ n}
  if |H ∪ S \ A| > 1 then
    τ_L := lower limit of the interval I^n of possible τ_n
    σ_L := OPTSCHEDULE(σ, n, τ_L)
    compute Û_L(τ_n)
    τ_U := upper limit of the interval I^n of possible τ_n
    σ_U := OPTSCHEDULE(σ, n, τ_U)
    compute Û_U(τ_n)
    partition I^n into subintervals I^n_1, I^n_2, ..., I^n_M s.t. σ_k makes
    Û_k(τ_n) maximal in I^n_k
    for k ← 1, 2, ..., M do
      if σ_k^{-1}(n+1) = σ^{-1}(n+1) then
        Ψ_k := LIMCMPTREE(σ, n+1)
      else
        Ψ_k := LIMCMPTREE(σ_k, n+1)
      end if
      ADDSUBTREE(Ψ_k, σ, n, I^n_k)
    end for
  end if
end
```

Figure 9: Algorithm LIMCMPTREE(σ, n)

here, we use both the optimal algorithm as well as a heuristic when solving the one-schedule problem. Thus we get four different heuristics for the multiple-schedules problem, namely LIMTREE$_A$, LIMTREE$_B$, LIMCMPTREE$_A$, and LIMCMPTREE$_B$. The first and third make use of an exact algorithm when solving the one-schedule problem while the other two make use of a heuristic presented in [5].

In order to evaluate the heuristics discussed above, we have generated a large number of synthetic examples. We considered systems with 50 tasks among which from 3 up to 25 hard and soft tasks. We generated 100 graphs for each graph dimension. All the experiments were run on a Sun Ultra 10 workstation.

Figure 10 shows the average size of the tree of schedules as a function of

the number of hard and soft tasks, for the optimal algorithm as well as for the heuristics. Note the exponential growth even in the heuristic cases which is inherent to the problem of computing a tree of schedules.

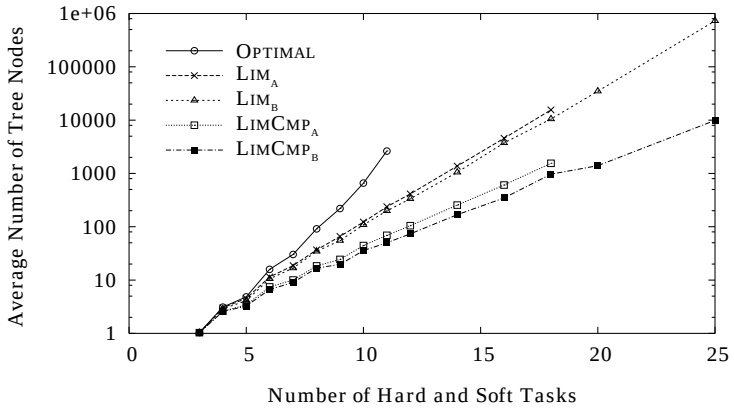


Figure 10: Average size of the tree of schedules

The average execution time of the algorithms is shown in Figure 11. The rapid growth rate of execution time for the optimal algorithm makes it feasible to obtain the optimal tree only in the case of small numbers of hard and soft tasks. Observe also that LimTree$_A$ takes much longer than LimTree$_B$, even though they yield trees with a similar number of nodes. A similar situation is noted for LimCmpTree$_A$ and LimCmpTree$_B$. This is due to the long execution time of the optimal algorithm for the one-schedule problem as compared to the heuristic.

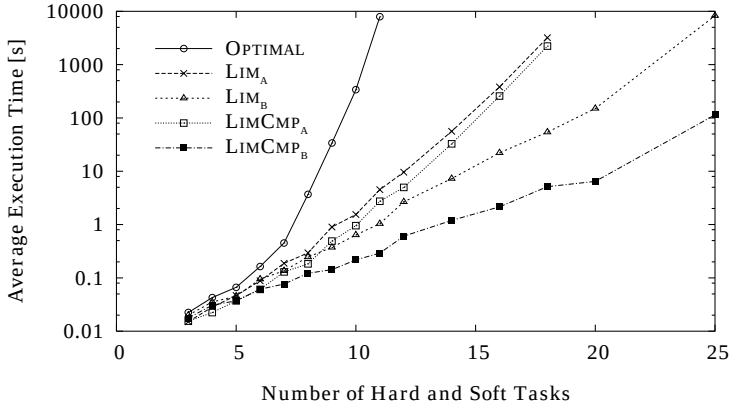


Figure 11: Average execution time when finding the tree of schedules

We have evaluated the quality of the trees of schedules as given by the

different algorithms with respect to the optimal tree. For each one of the randomly generated examples, we profiled the system for a large number of cases. We generated execution times for each task according to its probability distribution and, for each particular set of execution times, computed the total utility as given by a certain tree of schedules. For each case, we obtained the total utility yielded by a given tree and normalized with respect to the one produced by the optimal tree:

$$\|U_{alg}\| = U_{alg}/U_{opt}$$

The results are plotted in Figure 12. We have included in this plot the case of a purely off-line solution where only one schedule is used regardless of the actual execution times (SingleSch). This plot shows LimTree$_A$ and LimCmpTree$_A$ as the best of the heuristics discussed above, in terms of the total utility yielded by the trees they produce. LimTree$_B$ and LimCmpTree$_B$ produce still good results, not very far from the optimal, at a significantly lower computational cost. Observe that having one single static schedule leads to a significant quality loss.

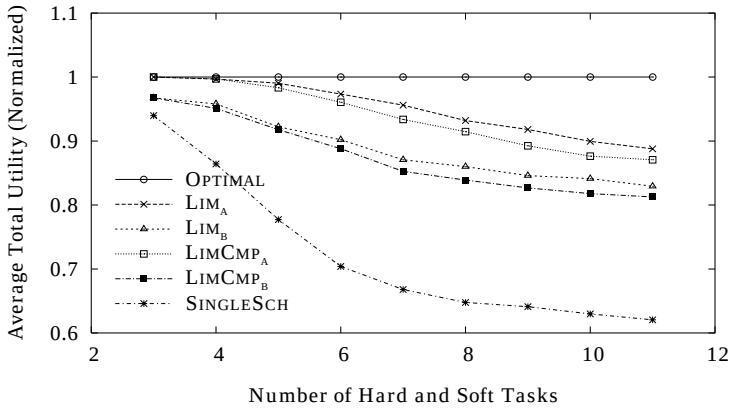


Figure 12: Average normalized total utility

6.2 Limiting the Tree Size

Even if we could afford to compute the optimal tree of schedules, the tree might be too large to fit in the available target memory. Hence we must drop some nodes of the tree at the expense of the solution quality (recall that we use the total utility as quality criterion). The heuristics presented in Section 6.1 reduce considerably both the time and memory needed to construct a tree as compared to the optimal algorithm, but still require exponential memory and time. In this section, on top of the above heuristics, we propose methods that construct a tree considering its size limit (imposed by the

memory constraints of the target system) in such a way that we can handle both the time and memory complexity.

Given a memory limit, only a certain number of schedules can be stored, so that the maximum tree size is M. Thus the question is how to generate a tree of at most M nodes which still delivers a good quality. We explore several strategies which fall under the umbrella of a generic framework with the following characteristics: (a) the algorithm goes on until no more nodes may be generated, due to the size limit M; (b) the tree is generated in a depth-first fashion; (c) in order to guarantee that hard deadlines are still satisfied when constructing a tree, either all children σ_k of a node σ (schedules σ_k to be followed after completing a task in σ) or none are added to the tree. The pseudocode for the generic algorithm is presented in Figure 13. The schedules to follow after σ correspond to those obtained in the interval-partitioning step as described in Sections 5 and 6.1. The difference among the approaches discussed in this section lies in the order in which the available memory budget is assigned to trees derived from the nodes σ_i (SORT$(\sigma_1, \sigma_2, \ldots, \sigma_c)$ in Figure 13).

```
Algorithm ConstructTree(σ, max)
input: A schedule σ and a positive integer max
output: A tree Ψ limited to max nodes whose root is σ
begin
  set σ as root of Ψ
  m := max − 1
  c := number of schedules to follow after σ
  if 1 < c ≤ m then
    add σ_1, σ_2, ..., σ_c as children of σ
    m := m − c
    Sort(σ_1, σ_2, ..., σ_c)
    for i ← 1, 2, ..., c do
      Ψ_i := ConstructTree(σ_i, m + 1)
      n_i := size of Ψ_i
      m := m − n_i + 1
    end for
  end if
end
```

Figure 13: Algorithm ConstructTree(σ, max)

For illustrative purposes, we will make use of the example in Figure 14. It represents a tree of schedules and we assume that it is the optimal tree

for a certain system. The intervals in the figure are the time intervals corresponding to switching points.

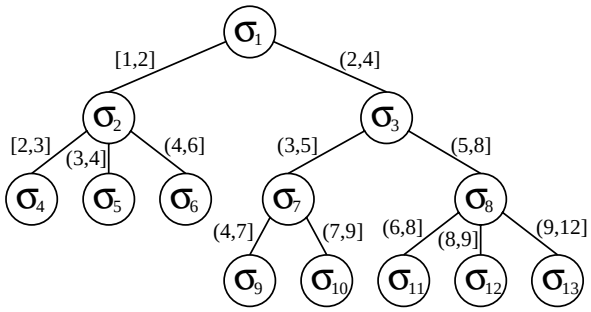


Figure 14: A complete tree of schedules

Initially we have studied two simple heuristics for constructing a tree, given a maximum size M. The first one, called EARLY, gives priority to subtrees derived from early-completion-time nodes (e.g. left-most subtrees in Figure 14). If, for instance, we are constructing a tree with a size limit $M = 10$ for the system whose optimal tree is the one given in Figure 14, we find out that σ_2 and σ_3 are the schedules to follow after σ_1 and we add them to the tree. Then, when using EARLY, the size budget is assigned first to the subtrees derived from σ_2 and the process continues until we obtain the tree shown in Figure 15. The second approach, LATE, gives priority to nodes that correspond to late completion times. The tree obtained when using LATE and having a size limit $M = 10$ is shown in Figure 16. Experimental data (see Figure 17) shows that in average LATE outperforms significantly EARLY. A simple explanation is that the system is more stressed in the case of late completion times and therefore the decisions (changes of schedule) taken under these conditions have a greater impact.

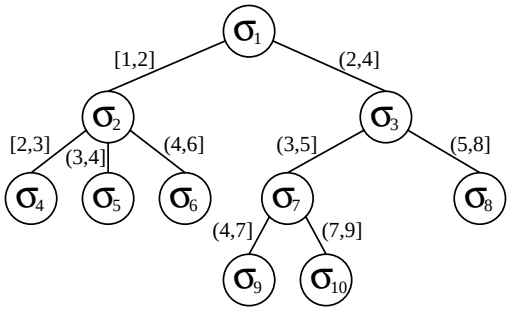


Figure 15: Tree constructed using EARLY

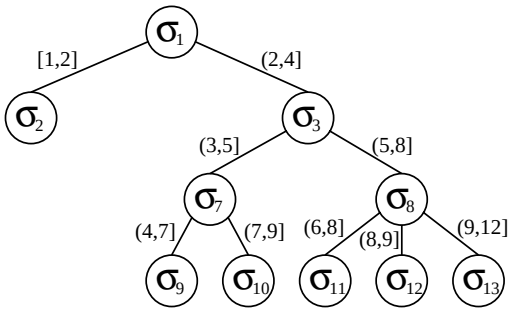


Figure 16: Tree constructed using LATE

A third, more elaborate, approach brings into the the picture the probability that a certain branch of the tree of schedules is selected during run-time. Knowing the execution time probability distribution of each individual task, we can get the probability distribution of a sequence of tasks as the convolution of the individual distributions. Thus, for a particular execution order, we may determine the probability that a certain task completes in a given interval, in particular the intervals defined by the switching points. In this way we can compute the probability for each branch of the tree and exploit this information when constructing the tree of schedules. The procedure PROB fits in the general framework of the algorithm of Figure 13, but instead of prioritizing early- or late-completion-time subtrees, it gives higher precedence to those subtrees derived from nodes that actually have higher probability of being followed at run-time.

In order to evaluate the approaches so far presented, we randomly generated 100 systems with a fix number of hard and soft tasks and for each one of them we computed the complete tree of schedules. Then we constructed the trees for the same systems using the algorithms presented in this section, for different size limits. For each of the examples we profiled the system for a large number of execution times, and for each of these we obtained the total utility yielded by a limited tree and normalized it with respect to the utility given by the complete tree (non-limited):

$$\|U_{lim}\| = U_{lim}/U_{non-lim}$$

The plot in Figure 17 shows that PROB is the algorithm that gives the best results in average.

We have further investigated the combination of PROB and LATE through a weighted function that assigns values to the tree nodes. Such values correspond to the priority given to nodes while constructing the tree. Each children of a certain node in the tree is assigned a value given by $w_1 p + (1-w_1)b$, where p is the probability of that node (schedule) being selected among its

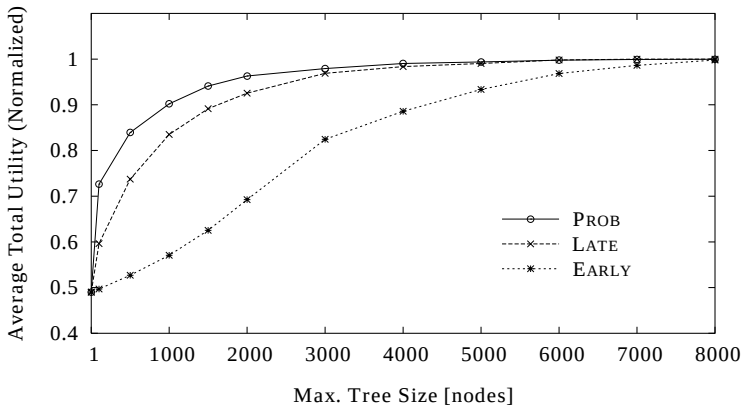


Figure 17: Evaluation of the construction algorithms

siblings and b is a quantity that captures how early/late are the completion times of that node relative to its siblings. More precisely, if there are n children, the earliest-completion-time child has $b_1 = 1/B$, the second earliest-completion-time child has $b_2 = 2/B$, etc., and the latest-completion-time child has $b_n = n/B$, where $B = \sum_{i=1}^{n} i$. Note that the particular cases $w_1 = 1$ and $w_1 = 0$ correspond to PROB and LATE respectively. The results of the weighted approach for different values of w_1 are illustrated in Figure 18. It is interesting to note that we can get even better results than PROB for certain weights, with $w_1 = 0.9$ being the one that performs the best.

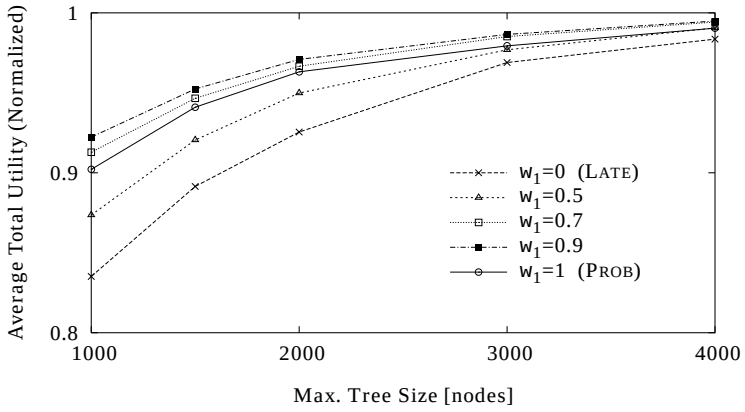


Figure 18: Construction algorithms using a weighted approach

7 Cruise Control with Collision Avoidance

Modern vehicles are equipped with sophisticated electronic aids aimed to assist the driver, increase the efficiency, and enhance the on-board comfort. One such system is the Cruise Control with Collision Avoidance (CCCA) [8] which assists the driver especially on highways in maintaining the speed and keeping safe distances to other vehicles. The CCCA allows the driver to set a particular speed. The system maintains that speed until the driver changes the reference speed, presses the break pedal, switches the system off, or the vehicle gets too close to an obstacle. The vehicle may travel faster than the set speed by overriding the control using the accelerator, but once it is released the cruise control will stabilize the speed to the set level. When another vehicle is detected in the same in front of the car, the CCCA will adjust the speed by applying limited braking to maintain a driver-selected distance to the vehicle ahead.

The CCCA is composed of four main subsystems, namely Braking Control (BC), Engine Control (EC), Collision Avoidance (CA), and Display Control (DC), each one of them having its own average period: $\mathsf{T}_{BC} = 100$ ms, $\mathsf{T}_{EC} = 200$ ms, $\mathsf{T}_{CA} = 125$ ms, $\mathsf{T}_{DC} = 500$ ms. We have modeled each subsystem as a task graph. Each subsystem has one hard deadline that equals its average period. We identified a number of soft tasks in the EC and DC subsystems. The soft tasks in the engine control part are related to the adjustment of the throttle valve for improving the fuel consumption efficiency. Thus their utility functions capture how such efficiency varies as a function of the completion time of the activities that calculate the best fuel injection rate for the actual conditions and accordingly control the throttle. For the display control part, the utility of soft tasks is a measure of the time-accuracy of the displayed data, that is, how soon the information on the dashboard is updated.

We generated several instances of the task graphs of the four subsystems mentioned above in order to construct a graph with a period $\mathsf{T} = 1000$ ms (least common multiple of the average periods of the involved tasks). The resulting graph contains 172 tasks, out of which 13 are soft and 25 are hard.

Since the CCCA is to be mapped on a vehicle ECU (Electronic Control Unit) which typically has 256 kB of memory, and assuming that we may use 40% of it for storing the tree of schedules and that one such schedule takes 100 B, we have an upper limit of 1000 nodes in the tree. We have constructed the tree of schedules using the weighted approach discussed in Section 6.2 combined with one of the heuristics presented in Section 6.1 ($\textsc{Lim}_B$). This construction takes 1591 s on a Sun Ultra 10 workstation.

Since it is infeasible to construct the optimal tree of schedules, we have

instead compared the tree with the static, off-line solution of a single schedule. We profiled the model of the CCCA and obtained an average total utility of 38.32 when using the tree of schedules, while the average total utility when using a single schedule was 29.05, that is, our approach gives in this case an average gain of 31.9%.

8 Conclusions

We have presented an approach to the problem of scheduling for real-time systems with periodic hard and soft tasks. We made use of non-increasing utility functions to capture the relevance of soft tasks and how the quality of results is diminished when missing a soft deadline. The problem we addressed is that of finding the execution order of tasks such that the total utility is maximum and hard deadlines are guaranteed.

Due to the pessimism of a purely off-line solution and the high overhead of a purely on-line approach, we proposed a quasi-static approach, where a tree of schedules and switching points is computed at design-time and the selection of schedule is done at run-time based on the actual execution times.

We have proposed an exact procedure that finds the optimal tree of schedules, in the sense that an ideal on-line scheduler is matched by a quasi-static scheduler using this tree. Also we have presented a number of heuristics for solving the problem efficiently.

References

[1] L. Abeni and G. Buttazzo. Integrating Multimedia Applications in Hard Real-Time Systems. In *Proc. Real-Time Systems Symposium*, pages 4–13, 1998.

[2] A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. A. Stankovic, and L. Strigini. The Meaning and Role of Value in Scheduling Flexible Real-Time Systems. *Journal of Systems Architecture*, 46(4):305–325, Jan. 2000.

[3] G. Buttazzo and F. Sensini. Optimal Deadline Assignment for Scheduling Soft Aperiodic Tasks in Hard Real-Time Environments. *IEEE. Trans. on Computers*, 48(10):1035–1052, Oct. 1999.

[4] K. Chen and P. Muhlethaler. A Scheduling Algorithm for Tasks described by Time Value Function. *Real-Time Systems*, 10(3):293–312, May 1996.

[5] L. A. Cortés, P. Eles, and Z. Peng. Static Scheduling of Monoprocessor Real-Time Systems composed of Hard and Soft Tasks. Technical report, Embedded Systems Lab, Dept. of Computer and Information Science, Linköping University, Linköping, Sweden, Apr. 2003.

[6] R. I. Davis, K. W. Tindell, and A. Burns. Scheduling Slack Time in Fixed Priority Pre-emptive Systems. In *Proc. Real-Time Systems Symposium*, pages 222–231, 1993.

[7] W.-C. Feng. *Applications and Extensions of the Imprecise-Computation Model*. PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, Dec. 1996.

[8] A. R. Girard, J. Borges de Sousa, J. A. Misener, and J. K. Hedrick. A Control Architecture for Integrated Cooperative Cruise Control with Collision Warning Systems. In *Proc. Conference on Decision and Control*, volume 2, pages 1491–1496, 2001.

[9] N. Homayoun and P. Ramanathan. Dynamic Priority Scheduling of Periodic and Aperiodic Tasks in Hard Real-Time Systems. *Real-Time Systems*, 6(2):207–232, Mar. 1994.

[10] H. Kaneko, J. A. Stankovic, S. Sen, and K. Ramamritham. Integrated Scheduling of Multimedia and Hard Real-Time Tasks. In *Proc. Real-Time Systems Symposium*, pages 206–217, 1996.

[11] J. P. Lehoczky and S. Ramos-Thuel. An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems. In *Proc. Real-Time Systems Symposium*, pages 110–123, 1992.

[12] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, May 1986.

[13] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A Resource Allocation Model for QoS Management. In *Proc. Real-Time Systems Symposium*, pages 298–307, 1997.

[14] I. Ripoll, A. Crespo, and A. García-Fornes. An Optimal Algorithm for Scheduling Soft Aperiodic Tasks in Dynamic-Priority Preemptive Systems. *IEEE. Trans. on Software Engineering*, 23(6):388–400, Oct. 1997.

[15] W.-K. Shih, J. W. S. Liu, and J.-Y. Chung. Fast Algorithms for Scheduling Imprecise Computations. In *Proc. Real-Time Systems Symposium*, pages 12–19, 1989.

[16] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. *IEEE. Trans. on Computers*, 44(1):73–91, Jan. 1995.