in *Proc. Design, Automation and Test in Europe Conference*, 2004, pp. 1176-1181.

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

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

Abstract

This paper addresses the problem of scheduling for real-time systems that include both hard and soft tasks. The relative importance of soft tasks and how the quality of results is affected when missing a soft deadline are captured by 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 time 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 paper.

1. Introduction

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 [1]. Both dynamic [3] and fixed priority systems [7], have been considered, minimizing in both cases the response time of soft aperiodic tasks.

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. 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 [9] in order to represent importance and criticality of tasks.

Utility-based scheduling [2] has been addressed before, for instance, in the context of imprecise computation techniques [11]. These consider tasks as composed by a mandatory and an optional part and the problem is 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 utility-based approaches include the QoS-based resource allocation model [10] 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). However, we consider the maximum duration of tasks for ensuring that all hard time constraints are met in every possible scenario.

In the context of the systems we are addressing in this paper, *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. *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 energy and time 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 paper is that of *quasi-static* scheduling for hard/soft real-time systems.

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.

We assume that tasks are non-preemptable. 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].

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

[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.

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. H and S denote the subsets of hard and soft tasks respectively. Non-real-time tasks have no timing constraints, though they may influence other hard or soft tasks through precedence constraints. A hard deadline $d(t)$ is the time by which a hard task $t \in H$ *must* be completed, otherwise the integrity of the system is jeopardized. A soft deadline $d(t)$ is the time by which a soft task $t \in S$ *should* be completed. Lateness of soft tasks is acceptable though it decreases 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 $t_j \in S$. Typical utility functions are depicted in Fig. 1.

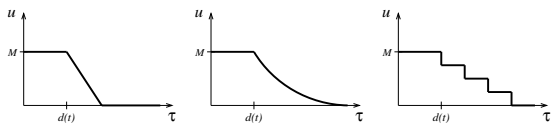
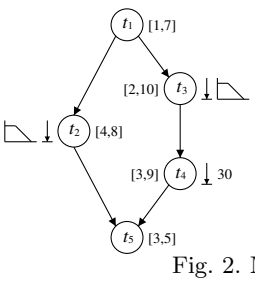


Fig. 1. Typical utility functions for soft tasks

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. A *switching point* defines when to switch from one to another schedule. A switching point is characterized by a task and a time interval, as well as the involved schedules. For example, the switching point $\sigma \xrightarrow{t_i;(a,b]} \sigma'$ 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 the system shown in Fig. 2. The minimum and maximum duration of every task are given in Fig. 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 in Fig. 2.



$$u_2(\tau_2) = \begin{cases} 3 & \text{if } \tau_2 \leq 9, \\ \frac{9}{2} - \frac{\tau_2}{6} & \text{if } 9 \leq \tau_2 \leq 27, \\ 0 & \text{if } \tau_2 \geq 27. \end{cases}$$

$$u_3(\tau_3) = \begin{cases} 2 & \text{if } \tau_3 \leq 18, \\ 8 - \frac{\tau_3}{3} & \text{if } 18 \leq \tau_3 \leq 24, \\ 0 & \text{if } \tau_3 \geq 24. \end{cases}$$

Fig. 2. Motivational example

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. We have proved that the problem of computing one such optimal schedule is **NP**-complete [6]. For the system of Fig. 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 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 $U^\times = u_2(15) + u_3(18) = 4$. Compare with the total utility $U = u_2(27) + u_3(10) = 2$ 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 meeting the hard deadlines. 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.

Though $\sigma = t_1 t_3 t_4 t_2 t_5$ ensures that hard constraints are always satisfied, it can 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 $U' = u_2(18) + u_3(10) = 3.5$, 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 the hard 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 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 [6], 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 Fig. 2, for instance, we can define a switching point $\sigma \xrightarrow{t_3;[3,13]} \sigma'$, where $\sigma = t_1 t_3 t_4 t_2 t_5$ and $\sigma' = t_1 t_3 t_2 t_4 t_5$, so that the system starts executing according to 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 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 $l(t)$, a maximum duration $m(t)$, and an expected duration $e(t)$ ($l(t) \leq e(t) \leq m(t)$) for each task $t \in T$; a subset $H \subseteq T$ of hard tasks; a deadline $d(t)$ for each hard task $t \in H$; 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 $t_j \in S$ (τ_j is the completion time of t_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 $\sigma : T \rightarrow \{1, 2, \ldots, |T|\}$) that maximizes $U = \sum_{t_j \in S} u_j(\tau_j^e)$, where τ_j^e is the expected completion time[2] of task t_j, subject to: $\tau_i^m \leq d(t_i)$ for all $t_i \in H$, where τ_i^m is the maximum completion time[3] of task t_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 paper is formulated as follows:

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 guarantees hard deadlines and maximizes $\sum_{t_j \in S} u_j(\tau_j^e)$ 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 $t \in T \setminus \{t_1\}$. 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 $t \in T \setminus \{t_1\}$. 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 $\hat{U}_i(\tau_1) = U_i(\tau_1)$.

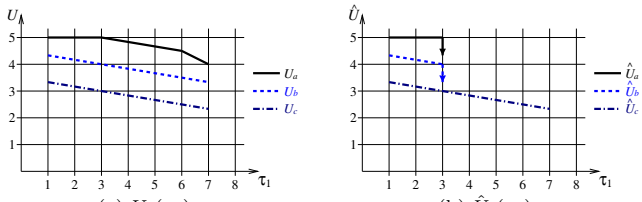
Once we have computed all the functions $\hat{U}_i(\tau_1)$, we may determine which σ_i yields the maximum total utility at which instants in the interval $[l(t_1), m(t_1)]$. We get thus the interval $[l(t_1), m(t_1)]$ partitioned into subintervals and, for each one of these, the schedule that maximizes 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 so on. In this manner we obtain the optimal *tree* of schedules and switching points.

Let us consider the example shown in Fig. 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 $\sigma_a = t_1 t_2 t_3 t_4 t_5$, $\sigma_b = t_1 t_3 t_2 t_4 t_5$, and $\sigma_c = t_1 t_3 t_4 t_2 t_5$. We 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:

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

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

The functions $U_a(\tau_1)$, $U_b(\tau_1)$, and $U_c(\tau_1)$, as given above, are shown in Fig. 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 $\sigma_a = t_1 t_2 t_3 t_4 t_5$ is followed and the remaining tasks take their maximum duration, the hard deadline $d(t_4)$ is met only when $\tau_1 \leq 3$. 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 functions $\hat{U}_i(\tau_1)$ as depicted in Fig. 3(b), from where we conclude that $\sigma_a = t_1 t_2 t_3 t_4 t_5$ yields the highest total utility when t_1 completes in the subinterval $[1, 3]$ and that $\sigma_c = t_1 t_3 t_4 t_2 t_5$ yields the highest total utility when t_1 completes in the subinterval $(3, 7]$, guaranteeing the hard deadline in both cases.



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

Fig. 3. $U_i(\tau_1)$ and $\hat{U}_i(\tau_1)$, $1 \leq \tau_1 \leq 7$

A similar procedure is followed, first for σ_a and then for

[2] 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 $\mathsf{e}_j = e(t_j)$.

[3] 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 $\mathsf{m}_i = m(t_i)$.

σ_c, considering the completion time of the second task in these schedules. At the end, 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 Fig. 4): 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.

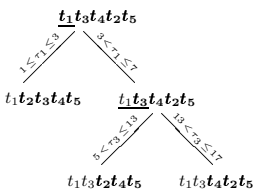


Fig. 4. Optimal tree of schedules and switching points

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. The details as well as the pseudocode and particularities of the algorithm for finding such an optimal tree can be found in [5].

When computing 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. As the interval-partitioning step requires $O((|H|+|S|)!)$ time in the worst case [5], 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, and hence a suboptimal set of schedules must be chosen to fit in the system memory. These issues are addressed in Section 6.

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. In the first heuristic, named LIM, 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. 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, LIM solves the one-schedule problem for $\tau_{\mathsf{L}} = 1$ and $\tau_{\mathsf{U}} = 7$, whose solutions are $\sigma_{\mathsf{L}} = t_1t_2t_3t_4t_5$ and $\sigma_{\mathsf{U}} = t_1t_3t_4t_2t_5$ 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 Fig. 3(b), $\hat{U}_{\mathsf{L}} = \hat{U}_a$ and $\hat{U}_{\mathsf{U}} = \hat{U}_c$, and in this case LIM gives the same result as the optimal algorithm. The rest of the procedure is repeated in a similar way as explained in Section 5.

The second of the proposed heuristics, named LIMCMP, is based on the same ideas as LIM, 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_k^n) is the same as the n+1-th task of the current schedule σ, the schedule σ continues being followed instead of adding σ_k to the tree. This leads to a tree with fewer nodes.

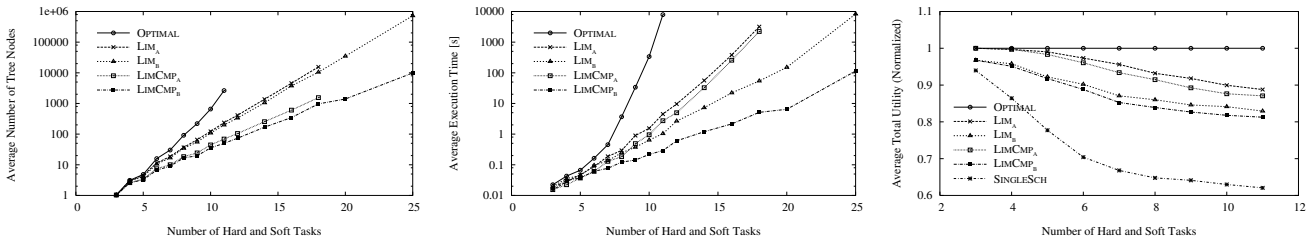
In both LIM and LIMCMP we must solve the one-schedule problem. The problem is **NP**-complete and we have proposed an optimal algorithm as well as different heuristics for it [6]. In the experimental evaluation of the heuristics proposed here, we use both the optimal algorithm as well as a heuristic when solving the one-schedule problem. Thus we get four different heuristics for the multiple-schedules problem, namely LIM_A, LIM_B, LIMCMP_A, and LIMCMP_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 [6].

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.

Fig. 5(a) 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.

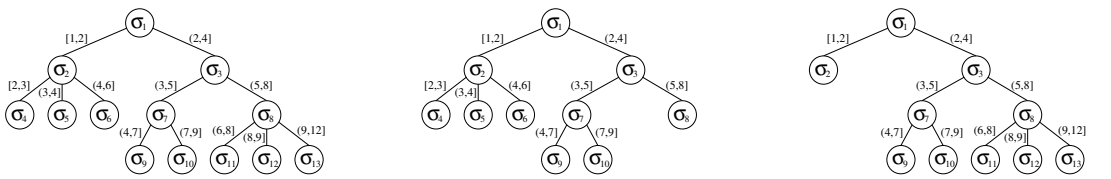
The average execution time of the algorithms is shown in Fig. 5(b). The rapid growth rate of execution time for the optimal algorithm makes it feasible to obtain the optimal tree only in the case of small numbers of hard and soft tasks. Observe also that LIM_A takes much longer than LIM_B, even though they yield trees with a similar number of nodes. A similar situation is noted for LIMCMP_A and LIMCMP_B. This is due to the long execution time of the optimal algorithm for the one-schedule problem as compared to the heuristic.

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 Fig. 5(c). 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 LIM_A and LIMCMP_A as the best of the heuristics discussed above, in terms of the total utility yielded by the trees they produce. LIM_B and LIMCMP_B produce still good results, not very far from the optimal, at a significantly



(a) Tree size (b) Execution time (c) Normalized total utility

Fig. 5. Evaluation of different algorithms for computing a tree of schedules (systems with 50 tasks)



(a) Complete tree (b) Using Early (max. size $M=10$) (c) Using Late (max. size $M=10$)

Fig. 6. Trees of schedules

lower computational cost. Observe that having one single static schedule leads to a significant quality loss.

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.

For illustrative purposes, we use the example in Fig. 6(a). 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.

Initially we studied two simple heuristics to 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 Fig. 6(a)). 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 Fig. 6(a), 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 Fig. 6(b). 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 Fig. 6(c). Experimental data (see Fig. 7) 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.

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 gives higher precedence to those subtrees derived from nodes that actually have higher probability of being followed at run-time.

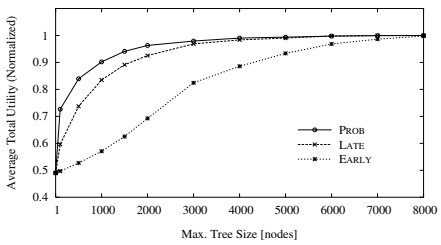


Fig. 7. Evaluation of the construction algorithms

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 Fig. 7 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 child 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 siblings and b is a quantity that captures how early/late are the completion times of that node relative to its siblings. 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 Fig. 8. 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.

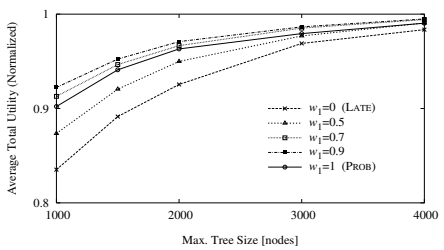


Fig. 8. Construction algorithms using a weighted approach

7. Cruise Control with Collision Avoidance

Modern vehicles can be 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 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. When another vehicle is detected in front of the car, the CCCA will adjust the speed by applying limited braking to maintain the 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 (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 around 30%.

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 utility functions to capture the relative relevance of soft tasks. 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*, pp. 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. Quasi-Static Scheduling for Real-Time Systems with Hard and Soft Tasks. Technical report, Embedded Systems Lab, Dept. of Computer Science, Linköping University, Linköping, Sweden, Sept. 2003.

[6] L. A. Cortés, P. Eles, and Z. Peng. Static Scheduling of Monoprocessor Real-Time Systems composed of Hard and Soft Tasks. In *Proc. Intl. Workshop on Electronic Design, Test and Applications*, pp. 115–120, 2004.

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

[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. Conf. on Decision and Control*, vol. 2, pp. 1491–1496, 2001.

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

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

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