Combining Static and Dynamic Scheduling for Real-Time Systems

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

We address in this paper the combination of static and dynamic scheduling into an approach called quasi-static scheduling, in the context of real-time systems composed of hard and soft tasks. For the particular problem discussed in this paper, a single static schedule is too pessimistic while a purely dynamic scheduling approach causes a very high on-line overhead. In the proposed quasi-static solution we compute at design-time a set of schedules, and leave for run-time only the selection of a particular schedule based on the actual execution times. We propose an exact algorithm as well as heuristics that tackle the time and memory complexity of the problem. The approach is evaluated through synthetic examples.

1. INTRODUCTION

Digital computer-based systems have become ubiquitous. These systems have many applications including automotive and aircraft controllers, cellular phones, household appliances, network switches, medical devices, and consumer electronics.

The remarkable development of computer systems is partly due to the advances in semiconductor technology. But also, to a great extent, new paradigms and design methodologies have made it possible to deploy devices with extraordinary computation capabilities. Innovative design frameworks have thus exploited the rapid technological progress in order to create more powerful computer systems at lower cost.

The design of modern computer systems is a difficult task as they must not only implement the desired functionality but also must satisfy diverse constraints (power consumption, performance, correctness, size, cost, flexibility) that typically compete with each other [10]. Moreover, the ever increasing complexity of computer systems combined with small time-to-market windows poses interesting challenges for the designers.

The designer must thus explore several design alternatives and trade off among the different design objectives. One such trade-off dimension is the choice of *static* solutions versus *dynamic* solutions. Let us take as example the problem of *mapping* in which the tasks or processes must be assigned to the processing elements (processors and buses) of the system. A static solution would correspond to deciding at *design-time* which tasks are mapped onto which processing elements, such that the very same mapping is kept along the operational life of the system. A dynamic solution would, on the other hand, allow the change of mapping during *run-time* (a typical case is process migration). Dynamic solutions exploit relevant information while the system executes but their overhead is usually quite significant. Static solutions cause no on-line overhead but have to make assumptions that in many cases turn out to be very pessimistic.

In practice, the execution time of tasks in a system is not fixed. It depends on the input data, the system state, and the target architecture, among others. Thus the execution time of a certain task may differ from one activation of the system to another. Therefore, task execution times are best modeled by time intervals. In this context, real-time systems leave open design alternatives that range from purely static solutions (where fixed execution times, typically worst-case execution times, are assumed) to dynamic solutions (where the actual execution time is taken into account at run-time).

We explore the combination of static and dynamic scheduling, into an approach we call *quasi-static scheduling*, for real-time systems where task execution times are given by intervals. In this paper we focus on a particular problem, namely scheduling for real-time systems composed of hard and soft tasks. Note, however, that the concept of quasi-static scheduling, and in general the idea of combining static and dynamic techniques might well be applied to other design problems. For example, slack management in the context of energy and power minimization has recently received renewed attention.

Many real-time systems are composed of tasks which are characterized by distinct types of timing constraints. Hard tasks have critical deadlines that must be met in every possible scenario. Soft tasks have looser timing constraints and soft deadline misses can be tolerated at the expense of the quality of results.

Scheduling for hard/soft real-time systems has been addressed, for example, in the context of integrating multimedia and hard real-time tasks [6], [1]. Most of the previous work on scheduling for hard/soft real-time systems

*This work has been partially supported by CICYT TIC 2001-2705, by Generalitat Valenciana CTIDIA/2002/205 and GRUPOS03/025, by EU-India Economic Cross Cultural Prog. ALA/95/23/2003/077-054, and by MCYT under grants HA2001-59 and HU2003-003.

assumes that hard tasks are periodic whereas soft tasks are aperiodic. The problem is to find a schedule such that all hard periodic tasks meet their deadlines and the response time of soft aperiodic tasks is minimized. The problem has been considered under both dynamic [2], [9] and fixed priority assignments [5], [7]. It is usually assumed that the sooner a soft task is served the better, but no distinction is made among soft tasks. However, by differentiating among soft tasks, processing resources can be allocated more efficiently. This is the case, for instance, in videoconference applications where audio streams are deemed more important than the video ones. We make use of utility functions in order to capture the relative importance of soft tasks and how the quality of results is influenced upon missing a soft deadline. Value or utility functions were first suggested by Locke [8] for representing significance and criticality of tasks.

We consider monoprocessor systems where both hard and soft tasks are periodic and there might exist data dependencies among tasks. We aim at finding an execution sequence (actually a set of execution sequences as explained later) such that the sum of individual utilities by soft tasks is maximal and, at the same time, satisfaction of all hard deadlines is guaranteed. Since the actual execution times usually do not coincide with parameters like expected durations or worst-case execution times (WCET), it is possible to exploit information regarding execution time intervals in order to obtain schedules that yield higher utilities, that is, improve the quality of results.

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). We take into consideration the fact that the actual execution time of a task is rarely its WCET. We use instead the expected or mean duration of tasks when evaluating the utility functions associated to soft tasks. Nevertheless, we do consider the worst-case duration of tasks for ensuring that hard time constraints are always met.

In the frame of the problem discussed in this paper, *off-line* scheduling refers to obtaining at design-time one single task execution order that makes the total utility maximal and guarantees the hard constraints. *On-line* scheduling refers to finding at run-time, every time a task completes, a new task execution order such that the total utility is maximized, yet guaranteeing that hard deadlines are met, but considering the actual execution times of those tasks which already completed. On the one hand, off-line scheduling causes no overhead at run-time but, by producing one static schedule, it can be too pessimistic since the actual execution times might be far off from the time values used to compute the schedule. On the other hand, on-line scheduling exploits the information about actual execution times and computes at run-time new schedules that improve the quality of results. But, due to the high complexity of the problem, the time and energy overhead needed for computing on-line the dynamic schedules is unacceptable. In order to exploit the benefits of off-line and on-line scheduling, and at the same time overcome their drawbacks, we propose an approach in which the scheduling problem is solved in two steps: first, we compute a number of schedules at design-time; second, we leave for run-time only the decision regarding which of the precomputed schedules to follow. Thus the problem we address in this paper is that of *quasi-static* scheduling for monoprocessor hard/soft real-time systems.

We propose a method for computing at design-time a set of schedules such that an ideal on-line scheduler (Section 4) is matched by a quasi-static scheduler operating on this set of schedules (Section 5). Since this problem is intractable, we present heuristics that deal with the time and memory complexity and produce suboptimal good-quality solutions (Section 6).

2. PRELIMINARIES

We consider that the functionality of the system is represented by a directed acyclic graph $G = (\mathbf{T}, \mathbf{E})$ where the nodes $\mathbf{T}$ correspond to tasks and data dependencies are captured by the graph edges $\mathbf{E}$.

The tasks that make up a system can be classified as non-real-time, hard, or soft. $\mathbf{H}$ and $\mathbf{S}$ denote, respectively, the subsets of hard and soft tasks. 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 $G = (\mathbf{T}, \mathbf{E})$. Both hard and soft tasks have deadlines. A hard deadline d_i is the time by which a hard task $T_i \in \mathbf{H}$ must be completed. A soft deadline d_i is the time by which a soft task $T_i \in \mathbf{S}$ 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 $u_i(t_i)$ for each soft task $T_i \in \mathbf{S}$, where t_i is the completion time of T_i. Typical utility functions are depicted in Figure 1. We consider that the delivered value or utility by a soft task decreases after its deadline (for example, in an engine controller, lateness of the task that computes the best fuel injection rate, and accordingly adjusts the throttle, implies a reduced fuel consumption efficiency), hence the use of non-increasing functions. The total utility, denoted U, is given by the expression $U = \sum_{T_i \in \mathbf{S}} u_i(t_i)$.

The actual execution time of a task T_i at a certain activation of the system, denoted τ_i, lies in the interval bounded by the best-case duration τ_i^{bc} and the worst-case duration τ_i^{wc} of the task, that is $\tau_i^{\mathrm{bc}} \leq \tau_i \leq \tau_i^{\mathrm{wc}}$ (dense-time semantics). The expected duration τ_i^{e} of a task T_i is the mean value of the possible execution times of the task.

We consider that tasks are non-preemptable. A schedule gives 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 σ is bijection $\sigma : \mathbf{T} \rightarrow \{1, 2, \ldots, |\mathbf{T}|\}$ where $|\mathbf{T}|$ denotes the cardinality of $\mathbf{T}$. We use the notation $\sigma = T_1 T_2 \ldots T_n$ as shorthand for $\sigma(T_1) = 1, \sigma(T_2) = 2, \ldots, \sigma(T_n) = |\mathbf{T}|$. We assume that the system is activated periodically and that there exists an implicit hard deadline equal to the period. This is easily modeled by adding a hard task, that is successor of all other tasks, which consumes no time and no resources. 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 their hyperperiod (least common multiple of the periods of the involved tasks).

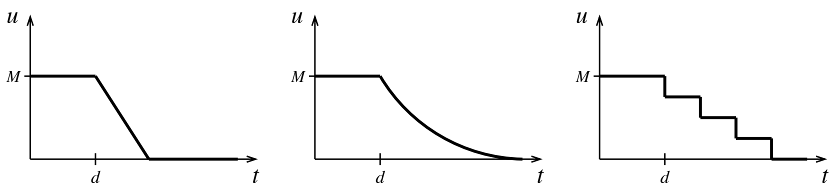


Figure 1: Typical utility functions for soft tasks

For a given schedule, the starting and completion times of a task T_i are denoted s_i and t_i respectively, with $t_i = s_i + \tau_i$. 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 $s_1 = 0$ and completes at $t_1 = \tau_1$, T_2 starts at $s_2 = t_1$ and completes at $t_2 = \tau_1 + \tau_2$, and so forth.

We aim at finding off-line a set of schedules and conditions under which the quasi-static 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, while σ is the current schedule, when the task T_i finishes and its completion time is $a \leq t_i \leq b$, another schedule σ' must be followed as execution order for the remaining tasks.

3. MOTIVATIONAL EXAMPLE

Let us consider the system shown in Figure 2. The best-case and worst-case duration of every task are shown in Figure 2 in the form $[\tau^{\text{bc}}, \tau^{\text{wc}}]$. In this example we assume that the execution time of every task T_i is uniformly distributed over its interval $[\tau_i^{\text{bc}}, \tau_i^{\text{wc}}]$. The only hard task in the system is T_4 and its deadline is $d_4 = 30$. Tasks T_2 and T_3 are soft and their utility functions are given in Figure 2.

The best static schedule, that can be calculated off-line, corresponds to the task execution order which, among all the schedules that satisfy the hard constraints in the worst case, maximizes the sum of individual contributions by soft tasks when each utility function is evaluated at the task's expected completion time (completion time considering the particular situation in which each task in the system lasts its expected duration). For the system in Figure 2 such a schedule is $\sigma = T_1 T_3 T_4 T_2 T_5$. We have proved that the problem of computing one such optimal schedule is **NP**-hard [3].

Although $\sigma = T_1 T_3 T_4 T_2 T_5$ is optimal in the static sense discussed above, it is still pessimistic because the actual execution times, which are unknown beforehand, might be far off from the ones used to compute the static schedule. This point is illustrated by the following situation. The system starts execution according to σ, that is T_1 starts at $s_1 = 0$. Assume that T_1 completes at $t_1 = 7$ and then T_3 executes ($\tau_3 = 3$) and completes at $t_3 = 10$. At this point, taking advantage of the fact that we know the completion time t_3, we can compute the schedule that is consistent with the tasks already executed, maximizes the total utility (considering the actual execution times of T_1 and T_3—already executed—and expected duration for T_2, T_4, T_5—remaining tasks), and also guarantees all hard deadlines (even if all remaining tasks execute with their worst-case duration). Such a schedule is $\sigma' = T_1 T_3 T_2 T_4 T_5$. In the case $\tau_1 = 7$, $\tau_3 = 3$, and $\tau_i = \tau_i^{\text{e}}$ for $i = 2, 4, 5$, σ' yields a total utility $U' = u_2(16) + u_3(10) = 3.83$ which is higher than the one given by the static schedule σ ($U = u_2(22) + u_3(10) = 2.83$). Since the decision to follow σ' is taken after T_3 completes and knowing its completion time, meeting the hard deadlines is also guaranteed.

A purely on-line scheduler would compute, every time a task completes, a new execution order for the tasks not yet started such that the total utility is maximized for the new conditions while guaranteeing that hard deadlines are met. However, the complexity of the problem is so high that the on-line computation of one such schedule is prohibitively expensive. In our quasi-static solution, we compute at design-time a number of schedules and switching points, leaving only for run-time the decision to choose a particular schedule based on the actual execution times. Thus the on-line overhead by the quasi-static scheduler is very low because it only compares the actual completion time of a task with that of a predefined switching point and selects accordingly the already computed execution order for the remaining tasks.

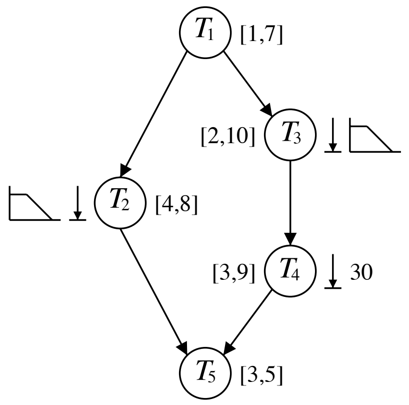
We can define, for instance, a switching point $\sigma \xrightarrow{T_3;[3,13]} \sigma'$ for the example given in Figure 2, with $\sigma = T_1 T_3 T_4 T_2 T_5$ and $\sigma' = T_1 T_3 T_2 T_4 T_5$, such that the system starts executing according to the schedule σ; when T_3 completes, if $3 \leq t_3 \leq 13$ the tasks not yet started execute in the order given by σ', else the execution order continues according to σ. While the solution $\{\sigma, \sigma'\}$, as explained above, guarantees meeting the hard deadlines, it provides a total utility which is greater than the one given by the static schedule σ in 83.3% of the cases, at a very low on-line overhead. Thus the most important question in the quasi-static approach discussed in this paper is how to compute, at design-time, the set of schedules and switching points such that they deliver the highest quality (utility). The rest of the paper addresses this question and different issues that arise when solving the problem.

4. IDEAL ON-LINE SCHEDULER AND PROBLEM FORMULATION

4.1 Ideal On-Line Scheduler

In this paper we use a purely on-line scheduler as reference point in our quasi-static approach. This means that, when computing a number of schedules and switching points as outlined in the previous section, our aim is to match an ideal on-line scheduler in terms of the yielded total utility. The formulation of this on-line scheduler is as follows:

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 the sequel



$$u_2(t_2) = \begin{cases} 3 & \text{if } t_2 \leq 9, \\ \frac{9}{2} - \frac{t_2}{6} & \text{if } 9 \leq t_2 \leq 27, \\ 0 & \text{if } t_2 \geq 27. \end{cases}$$

$$u_3(t_3) = \begin{cases} 2 & \text{if } t_3 \leq 18, \\ 8 - \frac{t_3}{3} & \text{if } 18 \leq t_3 \leq 24, \\ 0 & \text{if } t_3 \geq 24. \end{cases}$$

Figure 2: Motivational example

we will refer to this problem as the *one-schedule problem*):

> Find a schedule σ (a bijection $\sigma : \mathbf{T} \rightarrow \{1, 2, \dots, |\mathbf{T}|\}$) that maximizes $U = \sum_{T_i \in \mathbf{S}} u_i(t_i^{\mathrm{e}})$ where t_i^{e} is the expected completion time[1] of task T_i, subject to: $t_i^{\mathrm{wc}} \leq d_i$ for all $T_i \in \mathbf{H}$, where t_i^{wc} is the worst-case completion time[2] of task T_i; $\sigma(T) < \sigma(T')$ for all $(T, T') \in \mathbf{E}$; σ has a prefix σ_x, with σ_x being the order of the tasks already executed.

Ideal On-Line Scheduler: In an ideal case, where the on-line scheduler solves the one-schedule problem in zero time, for any set of execution times $\tau_1, \tau_2, \dots, \tau_n$ (each known only when the corresponding task completes), the total utility yielded by the on-line scheduler is denoted $U^{ideal}_{\{\tau_i\}}$.

The total utility delivered by the ideal on-line scheduler, as defined above, represents an upper bound on the utility that can practically be produced without knowing in advance the actual execution times and without accepting risks regarding hard deadline violations. This is due to the fact that the defined scheduler optimally solves the one-schedule problem in zero time, it is aware of the actual execution times of all completed tasks, and optimizes the total utility assuming that the remaining tasks will run for their expected (which is the most likely) execution time. We note again that, although the optimization goal is the total utility assuming expected duration for the remaining tasks, this optimization is performed under the constraint that hard deadlines are satisfied even in the situation of worst-case duration for the remaining tasks.

4.2 Problem Formulation

Due to the **NP**-hardness of the one-schedule problem [3], which the on-line scheduler must solve every time a task completes, such an on-line scheduler causes an unacceptable overhead. We propose instead to prepare at design-time schedules and switching points, where the selection of the actual schedule is done at run-time, at a low cost, by the so-called *quasi-static scheduler*. The aim is to match the utility delivered by an ideal on-line 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 set of execution times $\tau_1, \tau_2, \dots, \tau_n$, hard deadlines are guaranteed and the total utility $U_{\{\tau_i\}}$ yielded by the quasi-static scheduler is equal to $U^{ideal}_{\{\tau_i\}}$.

5. COMPUTING THE OPTIMAL SET OF SCHEDULES & SWITCHING POINTS

We present in this section a systematic procedure for computing the optimal set of schedules and switching points as formulated by the multiple-schedules problem. By optimal, in this context, we mean a solution which guarantees hard deadlines and produces a total utility of $U^{ideal}_{\{\tau_i\}}$. Note that the problem of obtaining such an optimal solution is intractable. Nonetheless, despite its complexity, the optimal procedure described here has also theoretical relevance: it shows that an infinite space of execution times (the execution time of task T_j can be any value in the interval $[\tau_j^{\mathrm{bc}}, \tau_j^{\mathrm{wc}}]$) might be covered optimally by a finite number of schedules, albeit it may be a very large number.

The key idea is to express the total utility, for every feasible task execution order, as a function of the completion time t_k of a particular task T_k. Since different schedules yield different utilities, the objective of the analysis is to pick out the schedule that gives the highest utility and also guarantees no hard deadline miss, depending on the completion time t_k.

We may thus determine (off-line) what is the schedule that must be followed after completing task T at a particular time t. For each schedule σ_i that satisfies the precedence constraints and is consistent with the tasks so far executed, we express the total utility $U_i(t)$ as a function of the completion time t (considering the expected duration for every task not yet started). Then, for every σ_i, we analyze the schedulability of the system, that is, which values of t imply potential hard deadline misses when σ_i is followed (for this analysis, the worst-case execution times of tasks not completed are considered). We introduce the auxiliary function $\hat{U}_i$ such that $\hat{U}_i(t) = -\infty$ if following σ_i, after T has completed at t, does not guarantee the hard deadlines,

[1] t_i^{e} is given by
$$t_i^{\mathrm{e}} = \begin{cases} \mathsf{e}_i & \text{if } \sigma(T_i) = 1, \\ t_k^{\mathrm{e}} + \mathsf{e}_i & \text{if } \sigma(T_i) = \sigma(T_k) + 1. \end{cases}$$
where: $\mathsf{e}_i = \tau_i$ if T_i has completed, else $\mathsf{e}_i = \tau_i^{\mathrm{e}}$.

[2] t_i^{wc} is given by
$$t_i^{\mathrm{wc}} = \begin{cases} \mathsf{wc}_i & \text{if } \sigma(T_i) = 1, \\ t_k^{\mathrm{wc}} + \mathsf{wc}_i & \text{if } \sigma(T_i) = \sigma(T_k) + 1. \end{cases}$$
where: $\mathsf{wc}_i = \tau_i$ if T_i has completed, else $\mathsf{wc}_i = \tau_i^{\mathrm{wc}}$.

else $\hat{U}_i(t) = U_i(t)$. Based on the functions $\hat{U}_i(t)$ we select the schedules that deliver the highest utility, yet guaranteeing the hard deadlines, at different completion times. The interval of possible completion times gets thus partitioned into subintervals and, for each of these, we get the corresponding execution order to follow after T. We refer to this as the *interval-partitioning step.* Observe that such subintervals define the switching points we want to compute.

For each of the newly computed schedules, the process is repeated for a task T' that completes after T, this time computing $\hat{U}_i(t')$ for the interval of possible completion times t'. Then the process is similarly repeated for the new schedules and so forth. In this way we obtain the optimal *tree* of schedules and switching points.

Let us consider the example shown in Figure 2. The static schedule is in this case $\sigma = T_1T_3T_4T_2T_5$. 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 compute the corresponding functions $U_a(t_1)$, $U_b(t_1)$, and $U_c(t_1)$, $1 \leq t_1 \leq 7$, considering the expected duration for T_2, T_3, T_4, and T_5. For example, $U_b(t_1) = u_2(t_1+\tau_3^e+\tau_2^e)+u_3(t_1+\tau_3^e) = u_2(t_1+12)+u_3(t_1+6)$. We get the following functions:

$$U_a(t_1) = \begin{cases} 5 & \text{if } 1 \leq t_1 \leq 3, \\ \dfrac{11}{2} - \dfrac{t_1}{6} & \text{if } 3 \leq t_1 \leq 6, \\ \dfrac{15}{2} - \dfrac{t_1}{2} & \text{if } 6 \leq t_1 \leq 7. \end{cases}$$

$$U_b(t_1) = \frac{9}{2} - \frac{t_1}{6} \quad \text{if } 1 \leq t_1 \leq 7.$$

$$U_c(t_1) = 7/2 - t_1/6 \quad \text{if } 1 \leq t_1 \leq 7.$$

The functions $U_a(t_1)$, $U_b(t_1)$, and $U_c(t_1)$, as given above, are shown in Figure 4(a). Now, for each one of the schedules σ_a, σ_b, and σ_c, we determine the latest completion time t_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_4 is met only when $t_1 \leq 3$. A similar analysis shows that σ_b guarantees meeting the hard deadline only when $t_1 \leq 3$ while σ_c guarantees the hard deadline for any completion time t_1 in the interval $[1, 7]$. Thus we get the functions $\hat{U}_i(t_1)$ as depicted in Figure 4(b), from where we conclude that $\sigma_a = T_1T_2T_3T_4T_5$ yields the highest total utility when T_1 completes in the subinterval $[1, 3]$ and that $\sigma_c = T_1T_3T_4T_2T_5$ yields the highest total utility when T_1 completes in the subinterval $(3, 7]$, guaranteeing the hard deadline in both cases.

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. 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 Figure 3): once the system is activated, it starts following the schedule σ; when T_1 is finished, its completion time t_1 is read, and if $t_1 \leq 3$ 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 t_3 is compared with the time point 13: if $t_3 \leq 13$ the remaining tasks are executed according to σ'', else the schedule σ is followed.

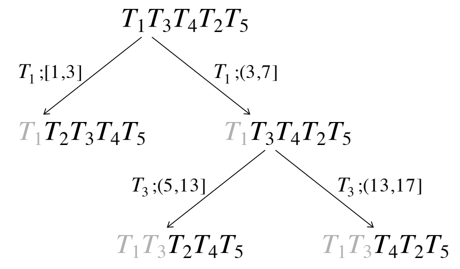


Figure 3: Optimal tree of schedules and switching points

When computing the optimal set of schedules and switching points, we partition the interval of possible completion times t_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 requires $\mathcal{O}((|\mathbf{H}| + |\mathbf{S}|)!)$ time in the worst case, therefore 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, 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.

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 $\mathcal{O}(logN)$ 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 worst-case 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 subsection we discuss methods to avoid the computation, in the interval-partitioning step, of $\hat{U}_i(t_n)$ 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 t_{L} and t_{U} of the interval I^n of possible completion times t_n. Then we compute $\hat{U}_{\mathsf{L}}(t_n)$ and $\hat{U}_{\mathsf{U}}(t_n)$ and partition I^n considering only these two (avoiding thus computing $\hat{U}_i(t_n)$ corresponding to the possible schedules σ_i defined by permutations of the remaining tasks). For completion times $t_n \in I^n$ different from t_{L}, σ_{L} is rather optimistic but it might happen that it does not guarantee hard deadlines.

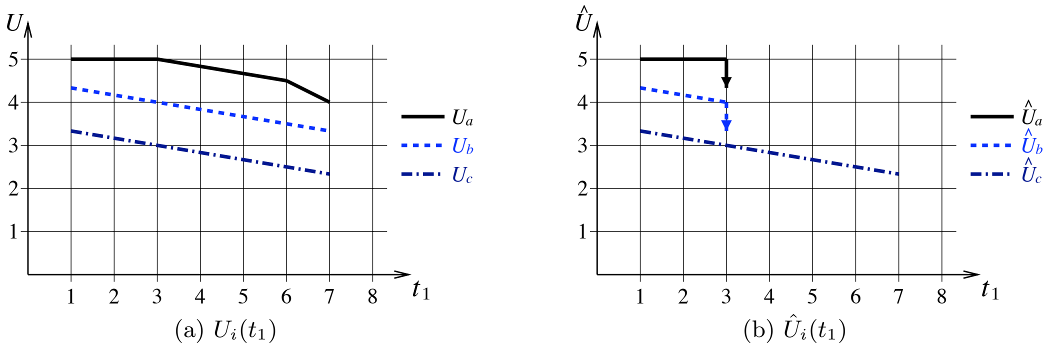


Figure 4: $U_i(t_1)$ **and** $\hat{U}_i(t_1)$**,** $1 \le t_1 \le 7$

On the other hand, σ_{U} can be pessimistic but does guarantee hard deadlines for all $t_n \in I^n$. Thus, by combining the optimism of σ_{L} with the guarantees provided by σ_{U}, good quality solutions can be obtained.

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 $t_{\mathsf{L}} = 1$ and $t_{\mathsf{U}} = 7$, whose solutions are $\sigma_{\mathsf{L}} = T_1T_2T_3T_4T_5$ and $\sigma_{\mathsf{U}} = T_1T_3T_4T_2T_5$ respectively. Then $\hat{U}_{\mathsf{L}}(t_1)$ and $\hat{U}_{\mathsf{U}}(t_1)$ are computed as described in Section 5 and only these two are used for partitioning the interval. Referring to Figure 4(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 $\hat{U}_{\mathsf{L}}(t_n)$ and $\hat{U}_{\mathsf{U}}(t_n)$ that correspond to schedules σ_{L} and σ_{U} which are in turn solutions to the one-schedule problem for t_{L} and t_{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**-hard and we have proposed an optimal algorithm as well as different heuristics for it [4]. 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 [4].

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.

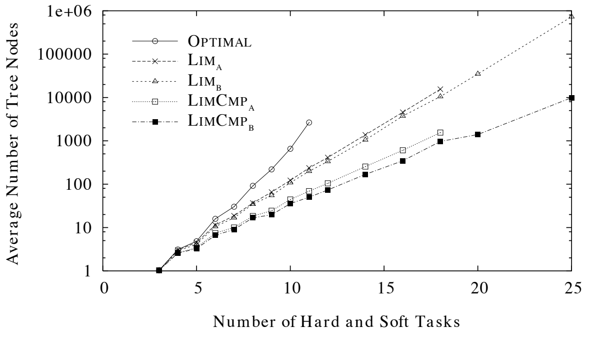
Figure 5(a) shows the average size of the tree of schedules as a function of the number of hard and soft tasks, for the optimal algorithm as well as for the heuristics. Note the exponential growth even in the heuristic cases which is inherent to the problem of computing a tree of schedules.

The average execution time of the algorithms is shown in Figure 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.

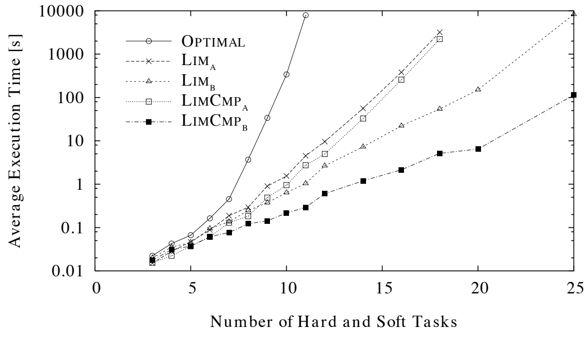
We have evaluated the quality of the trees of schedules as given by the different algorithms with respect to the optimal tree. For each one of the randomly generated examples, we profiled the system for a large number of cases. We generated execution times for each task according to its probability distribution 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 Figure 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 lower computational cost. Observe that having one single static schedule leads to a significant quality loss.

6.2 Tree Size Restriction

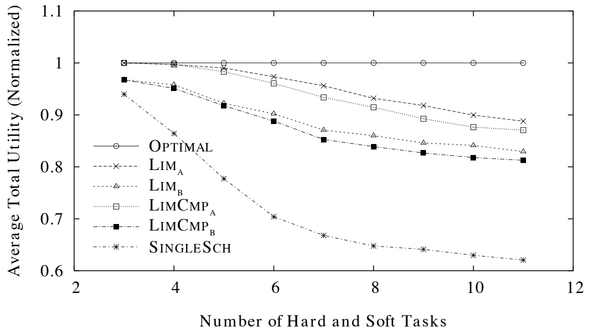
Even if we could afford to compute the optimal tree of schedules, the tree might be too large to fit in the available target memory. Hence we must drop some nodes of the tree at the expense of the solution quality (recall that we use the total utility as quality criterion). The heuristics presented in Subsection 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



(a) Tree size



(b) Execution time



(c) Normalized total utility

Figure 5: Evaluation of different algorithms for computing a tree of schedules

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.

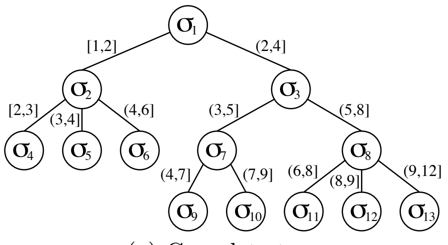
Given a memory limit, only a certain number of schedules can be stored, so that the maximum tree size is M. Thus the question is how to generate a tree of at most M nodes which still delivers a good quality. We explore several strategies which fall under the umbrella of a generic framework with the following 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 Figure 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.

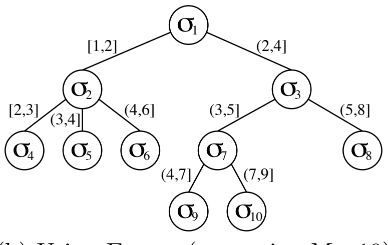
Initially we studied two simple heuristics to constructing a tree, given a maximum size M. The first one, called EARLY, gives priority to subtrees derived from early-completion-time nodes (e.g. left-most subtrees in Figure 6(a)). If, for instance, we are constructing a tree with a size limit $M = 10$ for the system whose optimal tree is the one given in Figure 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 Figure 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 Figure 6(c). Experimental data (see Figure 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.

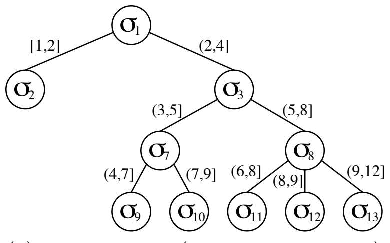
In order to evaluate the approaches so far presented, we randomly generated 100 systems with a fix number of hard and soft tasks and for each one of them we computed the complete tree of schedules. Then we constructed the trees for the same systems using the algorithms presented in this section, for 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



(a) Complete tree



(b) Using Early (max. size $M = 10$)



(c) Using Late (max. size $M = 10$)

Figure 6: Trees of schedules

to the utility given by the complete tree (non-limited): $\|U_{lim}\| = U_{lim}/U_{non-lim}$. The plot in Figure 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 Figure 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.

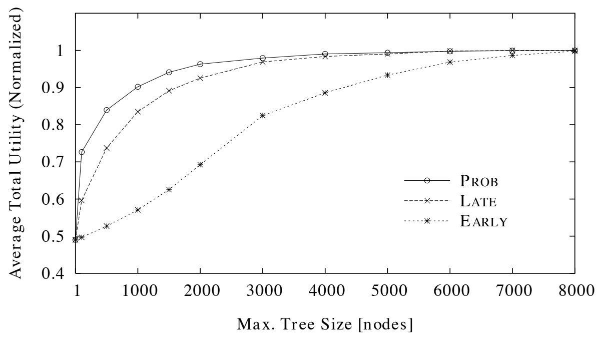


Figure 7: Evaluation of the construction algorithms

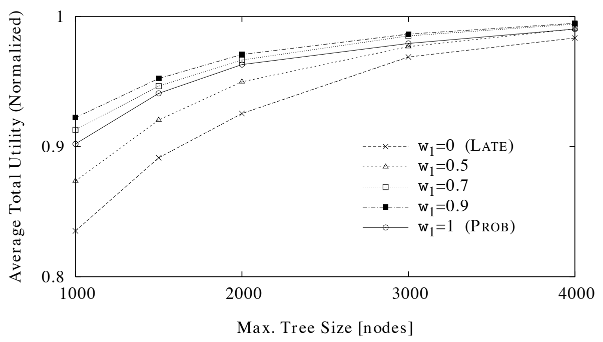


Figure 8: Construction algorithms using a weighted approach

7. CONCLUSIONS

One important dimension in the design space of computer systems is the issue of selecting solutions that may or may not vary during the operation of the system, that is, static versus dynamic solutions. In many cases, quasi-static solutions provide a good trade-off, exploiting information known only at run-time and achieving low on-line overhead.

We have presented an approach to the problem of scheduling for monoprocessor real-time systems with periodic soft and hard tasks. In order to distinguish among soft tasks, we made use of utility functions, which capture both the relative importance of soft tasks and how the quality of results is affected when a soft deadline is missed. We aimed at finding task execution orders that produce maximal total utility and, at the same time, guarantee hard deadlines.

Since a single static schedule computed off-line is rather pessimistic and a purely on-line solution entails a high overhead, we have therefore proposed a quasi-static approach where a number of schedules and switching points are prepared at design-time, so that at run-time the quasi-static scheduler only has to select, depending on the actual execution times, one of the precomputed schedules.

8. REFERENCES

[1] Luca Abeni and Giorgio Buttazzo. Integrating Multimedia Applications in Hard Real-Time

Systems. In *Proc. Real-Time Systems Symposium*, pages 4–13, 1998.

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

[3] Luis Alejandro Cortés, Petru Eles, and Zebo 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, April 2003. Available from `http://www.ida.liu.se/∼luico`.

[4] Luis Alejandro Cortés, Petru Eles, and Zebo Peng. Static Scheduling of Monoprocessor Real-Time Systems composed of Hard and Soft Tasks. In *Proc. Intl. Workshop on Electronic Design, Test and Applications*, pages 115–120, 2004.

[5] 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.

[6] Hiroyuki Kaneko, John A. Stankovic, Subhabrata Sen, and Krithi Ramamritham. Integrated Scheduling of Multimedia and Hard Real-Time Tasks. In *Proc. Real-Time Systems Symposium*, pages 206–217, 1996.

[7] John P. Lehoczky and Sandra 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.

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

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

[10] Frank Vahid and Tony Givargis. *Embedded Systems Design: A Unified Hardware/Software Introduction.* John Wiley & Sons, New York, NY, 2002.