Quasi-Static Scheduling for Multiprocessor 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
December 2003

Abstract

We address in this report the problem of scheduling for multiprocessor real-time systems comprised of hard and soft tasks. We use utility functions associated to soft tasks that capture their relative importance and how the quality of results is influenced when a soft deadline is missed. The problem is thus finding a task execution order that maximizes the total utility and guarantees meeting the hard deadlines. We consider time intervals rather than fixed execution times for tasks. On the one hand, a single static schedule computed off-line is too pessimistic. On the other hand, a purely on-line approach, which computes a new schedule every time a task completes considering the actual conditions, incurs an overhead that is unacceptable due to the high complexity of the problem. We propose a quasi-static solution where a number of schedules are computed at design-time, letting only for run-time 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. We evaluate our approach through synthetic examples and a realistic application.

1 Introduction

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.

As compared to pure hard real-time techniques, scheduling for hard/soft systems permits dealing with a broader range of applications. Scheduling for hard/soft systems has been addressed, for example, in the context of integrating multimedia and hard real-time tasks [12], [1]. Most of the previous work on scheduling for hard/soft real-time systems considers that hard tasks are periodic whereas soft tasks are aperiodic. The problem is thus finding 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 [3], [17], [11] and fixed priority assignments [8], [13], [21]. 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 [14] for representing significance and criticality of tasks.

In this report we consider multiprocessor systems where both hard and soft tasks are periodic and there might exist data dependencies among tasks. We aim to find 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. An important contribution of our approach is that we consider intervals rather than fixed execution times for tasks. 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 such information in order to obtain schedules that yield higher utilities, that is, improve the quality of results.

Utility-based scheduling [2], [15] has been addressed before, for instance, in the frame of imprecise computation techniques [20], [9]. These assume tasks as composed of a mandatory and an optional part: the mandatory part must be completed by its deadline and the optional one can be left incomplete at the expense of the quality of results. The problem to be solved is thus

finding a schedule that maximizes the total execution time of the optional subtasks. There are many systems, however, where it is not possible to identify the mandatory and optional parts of tasks. We consider in our approach that tasks have no optional part and tasks run until completion, once they have started executing. Our utility functions for soft tasks are expressed as function of the task completion time (and not its execution time as in the case of imprecise computation). Other utility-based approaches include best-effort techniques [14], the QoS-based resource allocation model [16], and Time-Value-Function scheduling [4]. The latter also uses the completion time as argument of the utility function, though it considers independent tasks running on a single processor and assumes fixed execution times. It proposes an $O(n^3)$ on-line heuristic whose overhead might still be too large for realistic systems.

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 maximum duration of tasks for ensuring that all hard time constraints are always met.

In the frame of the problem we discuss in this report, *off-line* scheduling refers to obtaining at design-time one 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 tasks already completed. On the one hand, off-line scheduling causes no overhead at run-time but having one static schedule 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 is totally unacceptable. In order to exploit the benefits of off-line and on-line scheduling, at the same time overcome their drawbacks, we combine them in a solution where we compute a number of schedules at design-time and leave only at run-time the decision of which of them is to be followed. Thus the problem we address in this report is that of *quasi-static* scheduling for multiprocessor hard/soft real-time systems.

Quasi-static scheduling has been studied previously, but mostly in the context of formal synthesis and without considering an explicit notion of

time, only the partial order of events [18], [22], [5]. Recently, in the context of real-time systems, Shih *et al.* have proposed a template-based approach that combines off-line and on-line scheduling for phase array radar systems [19], where templates for schedules are computed off-line considering performance constraints, and tasks are scheduled on-line such that they fit in the templates. The on-line overhead, though, can be significant when the system workload is high. In a previous work we have discussed quasi-static scheduling for hard/soft systems in the particular case of monoprocessor systems [7], a problem whose analysis complexity is significantly lower than when considering multiple processors.

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 problem addressed in this report. In Section 4 we give a precise formulation of the problem we are solving. An overview of different solutions for solving the off-line problem is given in Section 5. We present an exact method (Section 6) as well as a number of heuristics (Section 7) for solving the problem of quasi-static scheduling. A system corresponding to a real-life application is studied in Section 8. Finally, some conclusions are drawn in Section 9.

2 Preliminaries

We consider that the functionality of 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.

The mapping of tasks is defined by a function $M : T \rightarrow P$ where P is the set of processing elements. Thus $M(t)$ denotes the processing element on which task t executes. Inter-processor communication is captured by considering the buses as processing elements and the communication activities as tasks. If $t \in C$ then $M(t) \in B$, where $C \subset T$ is the set of communication tasks and $B \subset P$ is the set of buses.

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, that is $l(t) \leq |t| \leq m(t)$. 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 use ${}^\circ t$ to denote set of direct predecessors of task t, that is, ${}^\circ t = \{t' \in$

$T \mid \langle t', t\rangle \in E\}$. Similarly, $t^\circ = \{t' \in T \mid \langle t, t'\rangle \in E\}$ denotes the set of direct successors of task t.

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 Ω in a system with $|P|$ processing elements is a set of $|P|$ bijections $\{\sigma^{(1)} : T^{(1)} \rightarrow \{1, 2, \ldots, |T^{(1)}|\}, \sigma^{(2)} : T^{(2)} \rightarrow \{1, 2, \ldots, |T^{(2)}|\}, \ldots, \sigma^{(|P|)} : T^{(|P|)} \rightarrow \{1, 2, \ldots, |T^{(|P|)}|\}\}$ where $T^{(i)} = \{t \in T \mid M(t) = p_i\}$ is the set of tasks mapped onto the processing element p_i. We use the notation $\sigma^{(i)} = t_1 t_2 \ldots t_n$ as shorthand for $\sigma^{(i)}(t_1) = 1, \sigma^{(i)}(t_2) = 2, \ldots, \sigma^{(i)}(t_n) = |T^{(i)}|$. 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 a time period that is equal the least common multiple of the periods of the involved tasks.

A schedule does not provide the starting time for tasks, only their execution sequence. For a given schedule, the starting and completion times of a task t_i are denoted ϱ_i and τ_i respectively, with $\tau_i = \varrho_i + |t_i|$. Thus, for $\sigma^{(k)} = t_1 t_2 \ldots t_n$, task t_1 will start executing at $\varrho_1 = max_{\,t_j \in {}^\circ t_1}\{\tau_j\}$ and task t_i, $1 < i \leq n$, will start executing at $\varrho_i = max(max_{\,t_j \in {}^\circ t_i}\{\tau_j\}, \tau_{i-1})$. In the sequel, the starting and completion times that we use are relative to the system activation instant. Thus a task t with no predecessor such that $\sigma^{(k)}(t) = 1$ has starting time $\varrho = 0$. For example, in a monoprocessor system, according to the schedule $\Omega = \{\sigma^{(1)} = t_1 t_2 \ldots t_n\}$, t_1 starts executing at time $\varrho_1 = 0$ and completes at $\tau_1 = |t_1|$, 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, 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 $G = (T, E)$. Both hard and soft tasks have deadlines. 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 $u_i(\tau_i)$ for each soft task $t_i \in S$, where

τ_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 $U = \sum_{t_i \in S} u_i(\tau_i)$.

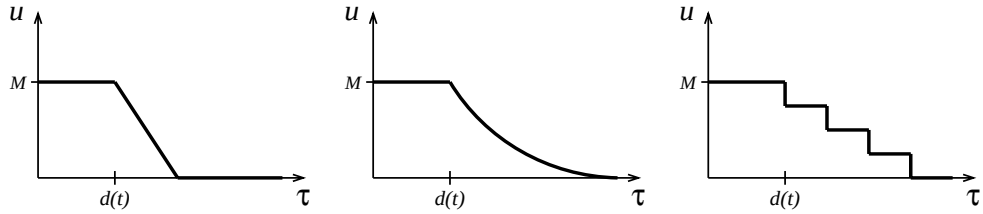


Figure 1: Typical utility functions for soft tasks

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 $\Omega \xrightarrow{t_i;(a,b]} \Omega'$ indicates that, while Ω is the current schedule, when the task t_i finishes and its completion time is $a < \tau_i \leq b$, another schedule Ω' must be followed as execution order for the remaining tasks.

We assume that the system has a dedicated shared memory for storing the set of schedules, which all processing elements can access. There is an exclusion mechanism that grants access to one processing element at a time. The worst case blocking time on this memory is considered in our analysis by increasing correspondingly the maximum duration of tasks. Upon finishing a task running on a certain processor, a new schedule can be selected (according to the set of schedules and switching points prepared off-line) which will then be followed by the tasks on all processing elements. Our analysis takes care that the execution sequence of tasks already executed or still under execution is consistent with the new schedule.

3 Motivational Example

Let us consider the system shown in Figure 2. Tasks t_1, t_3, t_5 are mapped on processor p_1 and tasks t_2, t_4, t_6, t_7 are mapped on p_2. For the sake of simplicity, we have ignored inter-processor communication. The minimum and

maximum duration of every task are given in Figure 2 in the form $[l(t), m(t)]$. In this example we assume that the execution time of every task t is uniformly distributed over its interval $[l(t), m(t)]$. Tasks t_3 and t_6 are hard and their deadlines are $d(t_3) = 16$ and $d(t_6) = 22$ respectively. Tasks t_5 and t_7 are soft and their utility functions are given, respectively, by:

$$u_5(\tau_5) = \begin{cases} 2 & \text{if } \tau_5 \leq 5, \\ 3 - \dfrac{\tau_5}{5} & \text{if } 5 \leq \tau_5 \leq 15, \\ 0 & \text{if } \tau_5 \geq 15. \end{cases} \qquad u_7(\tau_7) = \begin{cases} 3 & \text{if } \tau_7 \leq 3, \\ \dfrac{18}{5} - \dfrac{\tau_7}{5} & \text{if } 3 \leq \tau_7 \leq 18, \\ 0 & \text{if } \tau_7 \geq 18. \end{cases}$$

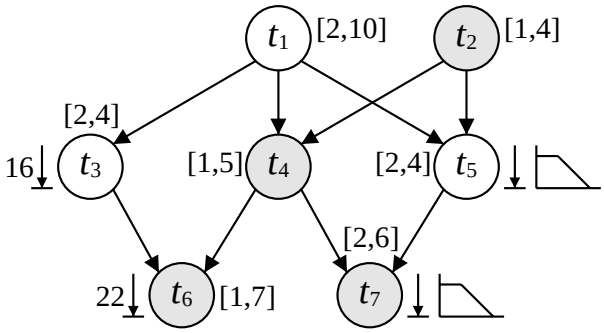


Figure 2: Motivational example

The static schedule corresponds to the task execution order that, 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 that each task in the system lasts its expected duration). For the system of Figure 2 such a schedule is $\Omega = \{\sigma^{(1)} = t_1 t_3 t_5, \sigma^{(2)} = t_2 t_4 t_6 t_7\}$ (in the rest of this section we will use the simplified notation $\Omega = \{t_1 t_3 t_5, t_2 t_4 t_6 t_7\}$). The problem of computing one such optimal schedule has been proved **NP**-hard even in the monoprocessor case [6].

Although $\Omega = \{t_1 t_3 t_5, t_2 t_4 t_6 t_7\}$ is optimal in the sense discussed above, it is too pessimistic, as illustrated by the following situation. The system starts execution according to Ω, that is t_1 and t_2 start at $\varrho_1 = \varrho_2 = 0$. Assume that t_2 completes at $\tau_2 = 4$ and then t_1 completes at $\tau_1 = 6$. At this point, taking advantage of the fact that we know the completion times τ_1 and τ_2, we can compute the schedule that maximizes the total utility (considering the actual execution times of t_1 and t_2—already executed—and expected duration for t_3, t_4, t_5, t_6, t_7—remaining tasks) and also guarantees all hard deadlines. Such a schedule is $\Omega' = \{t_1 t_5 t_3, t_2 t_4 t_6 t_7\}$. In the situation $|t_1| = 6$, $|t_2| = 4$, and $|t_i| = e(t_i)$ for $3 \leq i \leq 7$, Ω' yields a total utility

$U' = u_5(9) + u_7(20) = 1.2$ which is higher than the one given by the static schedule Ω ($U = u_5(12) + u_7(17) = 0.8$). Since the decision to follow Ω' is taken after t_1 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 utility is maximized for the new conditions while guaranteeing that hard deadlines are met. This would give the best results in terms of total utility. However, the complexity of the problem is so high that the on-line computation of one such schedule is prohibitively expensive.

We propose to compute at design-time a number of schedules and switching points, leaving only for run-time the decision of a particular schedule based on the actual execution times. Thus the on-line overhead is very low because it is simply comparing the actual completion time of a task with that of a predefined switching point and selecting accordingly the already computed execution order for the remaining tasks.

We can define, for instance, a switching point $\Omega \xrightarrow{t_1;[2,6]} \Omega'$ for the example given in Figure 2, with $\Omega = \{t_1t_3t_5, t_2t_4t_6t_7\}$ and $\Omega' = \{t_1t_5t_3, t_2t_4t_6t_7\}$, such that the system starts executing according to the schedule Ω; when t_1 completes, if $2 \leq \tau_1 \leq 6$ the tasks not yet started execute in the order given by Ω', else the execution order continues according to Ω. While the solution $\{\Omega, \Omega'\}$ as explained above guarantees meeting the hard deadlines, it provides a total utility which is greater than the one by the static schedule Ω in 43% of the cases at a very low on-line overhead. Also, by profiling the system (generating a large number of execution times for tasks according to their probability distributions and, for each particular set of execution times, computing the total utility) for each of the above two solutions, we find that the static schedule Ω yields an average total utility 0.89 while the quasi-static solution $\{\Omega, \Omega'\}$ gives an average total utility 1.04.

Another quasi-static solution, similar to the one discussed above, is $\{\Omega, \Omega'\}$ but with $\Omega \xrightarrow{t_1;[2,7]} \Omega'$ which actually gives better results (it outperforms the static schedule Ω in 56 % of the cases and yields an average total utility 1.1, yet guaranteeing no hard deadline miss). Thus the most important question in the quasi-static approach discussed in this report 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 report addresses this question and different issues that arise when solving the problem.

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 set P of processing elements and a function $M : T \rightarrow P$ defining the mapping of tasks; a minimum duration $l(t)$, an expected duration $e(t)$, and a maximum duration $m(t)$ for each task $t \in T$ ($l(t) \leq e(t) \leq m(t)$); a subset $H \subseteq T$ of hard tasks; a deadline $d(t)$ for each hard task $t \in H$; a subset $S \subseteq T$ of soft tasks ($S \cap H = \emptyset$); a non-increasing utility function $u_i(\tau_i)$ for each soft task $t_i \in S$ (τ_i is the completion time of t_i).

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

> Find a multiprocessor schedule Ω (a set of bijections $\{\sigma^{(1)} : T^{(1)} \rightarrow \{1, 2, \ldots, |T^{(1)}|\}, \ldots, \sigma^{(|P|)} : T^{(|P|)} \rightarrow \{1, 2, \ldots, |T^{(|P|)}|\}\}$ with $T^{(q)}$ being the set of tasks mapped on the processing element p_q) that maximizes $U = \sum_{t_i \in S} u_i(\tau_i^e)$ where τ_i^e is the expected completion time[1] of task t_i, subject to: no deadlock[2] is introduced by Ω; $\tau_i^m \leq d(t_i)$ for all $t_i \in H$, where τ_i^m is the maximum completion time[3] of task t_i; each $\sigma^{(q)}$ has a prefix $\sigma_x^{(q)}$, being $\sigma_x^{(q)}$ the order of the tasks already executed or under execution on processor p_q.

In an ideal case, where the on-line scheduler solves the one-schedule problem in zero time, for any set of execution times $|t_1|, |t_2|, \ldots, |t_n|$ (each known only when the corresponding task completes), the total utility $U_{\{|t_i|\}}$ yielded by the on-line scheduler is maximal and therefore denoted $U^{max}_{\{|t_i|\}}$.

Due to the **NP**-hardness of the one-schedule problem [6], which the on-line scheduler must solve every time a task completes, such an on-line sched-

---

[1] τ_i^e is given by

$$\tau_i^e = \begin{cases} max_{\,t_j \in {}^\circ t_i}\{\tau_j^e\} + \mathsf{e}_i & \text{if } \sigma^{(q)}(t_i) = 1, \\ max(max_{\,t_j \in {}^\circ t_i}\{\tau_j^e\}, \tau_k^e) + \mathsf{e}_i & \text{if } \sigma^{(q)}(t_i) = \sigma^{(q)}(t_k) + 1. \end{cases}$$

where: $M(t_i) = p_q$; $max_{\,t_j \in {}^\circ t_i}\{\tau_j^e\} = 0$ if ${}^\circ t_i = \emptyset$; $\mathsf{e}_i = |t_i|$ if t_i has been completed, $\mathsf{e}_i = m(t_i)$ if t_i is executing, else $\mathsf{e}_i = e(t_i)$.

[2] When considering a task graph with its original edges together with additional edges defined by the schedule, the resulting task graph must be acyclic.

[3] τ_i^m is given by

$$\tau_i^m = \begin{cases} max_{\,t_j \in {}^\circ t_i}\{\tau_j^m\} + \mathsf{m}_i & \text{if } \sigma^{(q)}(t_i) = 1, \\ max(max_{\,t_j \in {}^\circ t_i}\{\tau_j^m\}, \tau_k^m) + \mathsf{m}_i & \text{if } \sigma^{(q)}(t_i) = \sigma^{(q)}(t_k) + 1. \end{cases}$$

where: $M(t_i) = p_q$; $max_{\,t_j \in {}^\circ t_i}\{\tau_j^e\} = 0$ if ${}^\circ t_i = \emptyset$; $\mathsf{m}_i = |t_i|$ if t_i has been completed, else $\mathsf{m}_i = m(t_i)$.

uler causes an unacceptable overhead. We propose instead to prepare at design-time schedules and switching points aiming to match the utility delivered by an ideal on-line scheduler, where the selection of the actual schedule is done at run-time by the so-called *quasi-static scheduler* at a low cost. The problem we concentrate on in the rest of this paper is formulated as follows:

MULTIPLE-SCHEDULES PROBLEM: Find a set of multiprocessor schedules and switching points such that, for any set of execution times $|t_1|, |t_2|, \ldots, |t_n|$, the total utility $U_{\{|t_i|\}}$ yielded by the quasi-static scheduler is equal to $U^{max}_{\{|t_i|\}}$, yet guaranteeing the hard deadlines.

5 The One-Schedule Problem

The one-schedule problem refers to finding one schedule that maximizes the total utility for *one particular set of execution times*. Such a problem is **NP**-hard even in the case of a single processor [6]. The algorithm that solves exactly this problem is shown in Figure 3. A valid solution is found if the system is schedulable in first place.

Algorithm OPTIMALSCHEDULE($\Omega_x, \{|t_1|, |t_2|, \ldots, |t_n|\}$)
input: A schedule prefix Ω_x and a set of execution times $|t_1|, |t_2|, \ldots, |t_n|$
output: The optimal schedule Ω

```
begin
  Ω := ϵ
  U := −∞
  A := {t ∈ T | t is in Ω_x}
  for i ← 1, 2, ... , |T \ A|! do
    if Ω_i is valid and guarantees hard deadlines then
      U_i = Σ_{t_j ∈ S} u_j(τ_j^e)
      if U_i > U then
        Ω := Ω_i
        U := U_i
      end if
    end if
  end for
end
```

Figure 3: Algorithm OPTIMALSCHEDULE

We propose a couple of heuristics for solving the one-schedule problem. They progressively construct the schedule Ω by concatenating tasks to the strings $\sigma^{(k)}$, where $\sigma^{(k)}$ is the order of tasks mapped on processing element p_k.

In the sequel, when we say concatenate t to Ω it will really mean concatenate t to $\sigma^{(k)}$, considering that $M(t) = p_k$. The algorithms make use of a list *Ready* of available tasks (after the tasks in the schedule prefix Ω_x have finished) at every step. The difference among the heuristics presented in this section lies in how the next task, among those in *Ready*, is selected to be concatenated to one of the $\sigma^{(k)}$. The generic heuristic is presented in Figure 4.

Algorithm HEURSCHEDULE($\Omega_x, \{|t_1|, |t_2|, \ldots, |t_n|\}$)
input: A schedule prefix Ω_x and a set of execution times $|t_1|, |t_2|, \ldots, |t_n|$
output: A near-optimal schedule Ω

begin
  $\Omega := \Omega_x$
  $A := \{t \in T \mid t \text{ is in } \Omega_x\}$
  $Ready := \{t \in T \mid t \text{ is ready after finishing all } t_x \in \Omega_x\}$
  while $Ready \neq \emptyset$ **do**
    $B := \{t \in Ready \mid$ by concatenating t to Ω, hard deadlines are still guaranteed$\}$
    if $S \setminus A = \emptyset$ **then**
      select $\bar{t} \in B$
    else
      $\mathsf{SP} :=$ PRIORITY(Ω)
      select $t_j \in S \setminus A$ such that $\mathsf{SP}_{[j]} \geq \mathsf{SP}_{[i]}$ for all $t_i \in S \setminus A$
      $C := \{t \in B \mid (t, t_j) \in \mathcal{P}\}$
      if $C \neq \emptyset$ **then**
        select $\bar{t} \in C$
      else
        select $\bar{t} \in B$
      end if
    end if
    concatenate t to Ω
    $A := A \cup \{\bar{t}\}$
    $Ready := Ready \setminus \{\bar{t}\} \cup \{t \in \bar{t}^{\circ} \mid$ all $q \in {}^{\circ}t$ are in $\Omega\}$
  end while
end

Figure 4: Algorithm HEURSCHEDULE

PRIORITY(Ω) is a method for computing priorities for soft tasks that guide the construction of Ω when using the list-scheduling-based algorithm of Figure 4. The first of the proposed heuristics makes use of the generic algorithm of Figure 4 together with the algorithm of Figure 5.

A second algorithm for computing priorities for soft tasks is shown in Figure 6. It defines our second heuristic within the frame of the algorithm of Figure 4.

Algorithm PRIORITY$_C(\Omega)$
input: A schedule prefix Ω
output: A vector SP containing the priority for soft tasks

```
begin
  A := {t ∈ T | t is in Ω}
  for i ← 1, 2, ... , |S| do
    if t_i ∈ A then
      SP_[i] := −∞
    else
      construct Ω_i that has Ω as prefix and such that t_i is set the earliest
      compute τ_i^e according to Ω_i
      SP_[i] := u_i(τ_i^e)
    end if
  end for
end
```

Figure 5: Algorithm PRIORITY$_C$

6 Computing the Optimal Set of Schedules and 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.

The key idea is to express the total utility, for every feasible task execution order, as a function of the completion time τ of a particular task t. 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 τ.

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

Algorithm PRIORITY$_B(\Omega)$
input: A schedule prefix Ω
output: A vector SP containing the priority for soft tasks

begin
  $A := \{t \in T \mid t \text{ is in } \Omega\}$
  for $i \leftarrow 1, 2, \ldots, |S|$ **do**
    if $t_i \in A$ **then**
      $\mathsf{SP}_{[i]} := -\infty$
    else
      construct Ω_i that has Ω as prefix and such that t_i is set the earliest
      compute τ_j^e for all $t_j \in S$ according to Ω_i
      $\mathsf{SP}_{[i]} := \sum_{t_j \in S} u_j(\tau_j^e)$
    end if
  end for
end

Figure 6: Algorithm PRIORITY$_B$

and, for each of these, the corresponding execution order to follow after t. We refer to this as the *interval-partitioning step*. Note that such subintervals define the switching points we want to compute.

For each of the obtained schedules, the process is repeated for a task t' that completes after t, this time computing $\hat{U}_i(\tau')$ as a function of τ' for the interval of possible completion times of 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.

An important aspect to bear in mind while constructing the optimal tree of schedules is that tasks mapped on different processors may be running in parallel at a certain moment. Therefore the "next task to complete" may not necessarily be unique. For example, if tasks t_1, t_2, t_3 execute concurrently and their completion time intervals overlap, either of them can complete first. In our analysis we consider separately each of these situations. For each situation the interval of possible completion times can easily be computed and then it can be partitioned (getting the schedule(s) to follow after completing the task in that particular interval) as explained above. In other words, the tree includes the interleaving of possible finishing orders for concurrent tasks.

In order to illustrate this procedure, we make use of the example shown in Figure 2. In this case the initial schedule is $\Omega = \{t_1 t_3 t_5, t_2 t_4 t_6 t_7\}$, that is, t_1 and t_2 start executing at time zero and their completion time intervals are $[2, 10]$ and $[1, 4]$ respectively. We initially consider two situations: t_1 completes before t_2 ($2 \leq \tau_1 \leq 4$); t_2 completes before t_1 ($1 \leq \tau_2 \leq 4$). For

the first one, we compute $\hat{U}_i(\tau_1)$, $2 \leq \tau_1 \leq 4$, for each one of the Ω_i that satisfy the precedence constraints, and we find that $\Omega'' = \{t_1t_5t_3, t_2t_4t_7t_6\}$ is the schedule to follow after t_1 completes before t_2 at $\tau_1 \in [2, 4]$. For the second situation, in a similar manner, we find that when t_2 completes before t_1 in the interval $[1, 4]$, $\Omega = \{t_1t_3t_5, t_2t_4t_6t_7\}$ is the schedule to follow. Details of the interval-partitioning step are illustrated next.

Let us continue with the branch after t_2 completes in $[1, 4]$. Under this conditions t_1 is the only running task and its interval of possible completion times is $[2, 10]$. Due to the data dependencies, there are four feasible schedules $\Omega_a = \{t_1t_3t_5, t_2t_4t_6t_7\}$, $\Omega_b = \{t_1t_3t_5, t_2t_4t_7t_6\}$, $\Omega_c = \{t_1t_5t_3, t_2t_4t_6t_7\}$, and $\Omega_d = \{t_1t_5t_3, t_2t_4t_7t_6\}$, and for each of these we compute the corresponding functions $U_a(\tau_1)$, $U_b(\tau_1)$, $U_c(\tau_1)$, and $U_d(\tau_1)$, $2 \leq \tau_1 \leq 10$, considering expected duration for t_3, t_4, t_5, t_6, t_7. For example, $U_d(\tau_1) = u_5(\tau_1 + e(t_5)) + u_7(\tau_1 + max(e(t_4), e(t_5)) + e(t_7)) = u_5(\tau_1 + 3) + u_7(\tau_1 + 7)$. We get the functions shown in Figure 7(a) and given by:

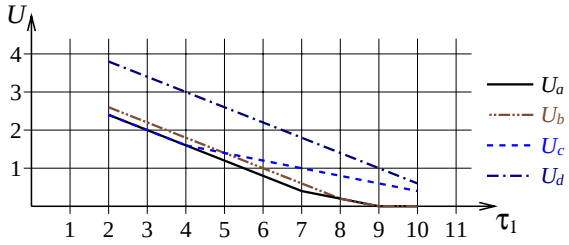
$$U_a(\tau_1) = \begin{cases} \frac{16}{5} - \frac{2\tau_1}{5} & \text{if } 2 \leq \tau_1 \leq 7, \\ \frac{9}{5} - \frac{\tau_1}{5} & \text{if } 7 \leq \tau_1 \leq 9, \\ 0 & \text{if } 9 \leq \tau_1 \leq 10. \end{cases} \qquad U_b(\tau_1) = \begin{cases} \frac{17}{5} - \frac{2\tau_1}{5} & \text{if } 2 \leq \tau_1 \leq 8, \\ \frac{9}{5} - \frac{\tau_1}{5} & \text{if } 8 \leq \tau_1 \leq 9, \\ 0 & \text{if } 9 \leq \tau_1 \leq 10. \end{cases}$$

$$U_c(\tau_1) = \begin{cases} \frac{16}{5} - \frac{2\tau_1}{5} & \text{if } 2 \leq \tau_1 \leq 4, \\ \frac{12}{5} - \frac{\tau_1}{5} & \text{if } 4 \leq \tau_1 \leq 10. \end{cases} \qquad U_d(\tau_1) = \frac{23}{5} - \frac{2\tau_1}{5} \quad \text{if } 2 \leq \tau_1 \leq 10.$$

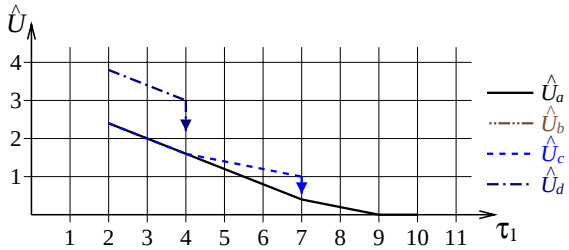
Now, for Ω_a, Ω_b, Ω_c, and Ω_d, we compute the latest completion time τ_1 that guarantees satisfaction of the hard deadlines when that particular schedule is followed. For example, when the execution order is $\Omega_c = \{t_1t_5t_3, t_2t_4t_6t_7\}$, in the worst case $\tau_3 = \tau_1 + m(t_5) + m(t_3) = \tau_1 + 8$ and $\tau_6 = max(\tau_3, \tau_1 + m(t_4)) + m(t_6) = max(\tau_1 + 8, \tau_1 + 5) + 7 = \tau_1 + 15$. Since the hard deadlines for this system are $d(t_3) = 16$ and $d(t_6) = 22$, when Ω_c is followed, $\tau_3 \leq 16$ and $\tau_6 \leq 22$ iff $\tau_1 \leq 7$. A similar analysis shows: Ω_a guarantees the hard deadlines for any completion time $\tau_1 \in [2, 10]$; Ω_b implies potential hard deadline misses for any $\tau_1 \in [2, 10]$; Ω_d guarantees the hard deadlines iff $\tau_1 \leq 4$. Thus we get auxiliary functions as shown in Figure 7(b) and given below:

$$\hat{U}_a(\tau_1) = \begin{cases} \frac{16}{5} - \frac{2\tau_1}{5} & \text{if } 2 \leq \tau_1 \leq 7, \\ \frac{9}{5} - \frac{\tau_1}{5} & \text{if } 7 \leq \tau_1 \leq 9, \\ 0 & \text{if } 9 \leq \tau_1 \leq 10. \end{cases} \qquad \hat{U}_b(\tau_1) = -\infty \quad \text{if } 2 \leq \tau_1 \leq 10.$$

$$\hat{U}_c(\tau_1) = \begin{cases} \frac{16}{5} - \frac{2\tau_1}{5} & \text{if } 2 \le \tau_1 \le 4, \\ \frac{12}{5} - \frac{\tau_1}{5} & \text{if } 4 \le \tau_1 \le 7, \\ -\infty & \text{if } 7 < \tau_1 \le 10. \end{cases} \qquad \hat{U}_d(\tau_1) = \begin{cases} \frac{23}{5} - \frac{2\tau_1}{5} & \text{if } 2 \le \tau_1 \le 4, \\ -\infty & \text{if } 4 < \tau_1 \le 10. \end{cases}$$



(a) $U_i(\tau_1)$



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

Figure 7: $U_i(\tau_1)$ and $\hat{U}_i(\tau_1)$, $2 \le \tau_1 \le 10$, for the example of Figure 2

From the graph in Figure 7(b) we conclude that upon completing t_1, in order to get the highest total utility while guaranteeing hard deadlines, the tasks not started must execute according to: $\Omega_d = \{t_1 t_5 t_3, t_2 t_4 t_7 t_6\}$ if $2 \le \tau_1 \le 4$; $\Omega_c = \{t_1 t_5 t_3, t_2 t_4 t_6 t_7\}$ if $4 < \tau_1 \le 7$; $\Omega_a = \{t_1 t_3 t_5, t_2 t_4 t_6 t_7\}$ if $7 < \tau_1 \le 10$.

The process is then repeated in a similar manner for the newly computed schedules and the possible completion times as defined by the switching points, and so forth until the full tree is constructed. The optimal tree of schedules for the system of Figure 2 is presented in Figure 8. When all the descendant schedules of a node (schedule) in the tree are equal to that node, there is no need to store those descendants because the execution order will not change. This is the case of the schedule $\{t_1 t_5 t_3, t_2 t_4 t_7 t_6\}$ followed after completing t_1 in $[2, 4]$. Also, note that for certain nodes of the tree, there is no need to store the full schedule in the memory of the target system. For example, the execution order of tasks already completed (which has been taken into account during the preparation of the set of schedules) is clearly unnecessary for the remaining tasks during run-time.

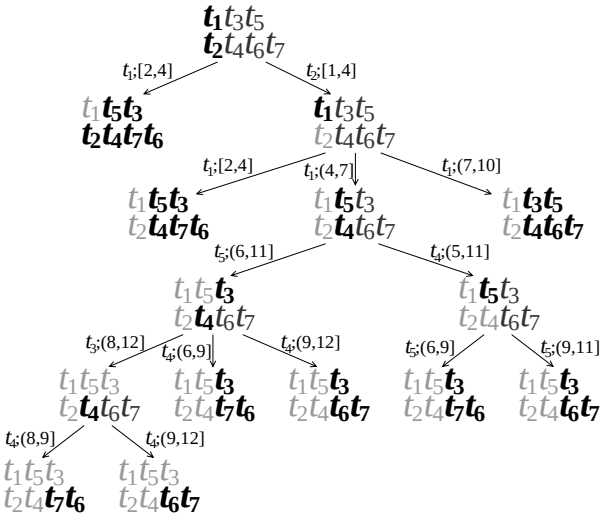


Figure 8: Optimal tree of schedules and switching points

It is not difficult to show that the method we have described finds an optimal tree of schedules, that is, a set of schedules and switching points that deliver the same total utility, for any set of of execution times, as the ideal on-line scheduler described in Section 4. By analyzing the feasible schedules after completing a task in a particular time interval, and having the same considerations about the duration of tasks not yet executed as the on-line scheduler does, our procedure solves *symbolically* the same optimization problem for a set of completion times, one of which corresponds to the particular instance solved by the on-line scheduler. Thus the quasi-static scheduler selects one of the precomputed schedules of the optimal tree, which yields a total utility that is equal to that of the ideal on-line scheduler, for any set of execution times.

The pseudocode of the algorithm for finding the optimal set of schedules and switching points is presented in Figures 9 and 10. 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.

When finding the best schedule (the one that yields the highest utility and guarantees no hard deadline miss) to follow after completing a task t in an interval of possible completion times τ, it is necessary to analyze all schedules that respect the data dependencies and are consistent with the tasks already executed. This means that the interval-partitioning step requires $O(|T|!)$ time in the worst case and therefore the multiple-schedules problem is intractable. Moreover, the inherent nature of the problem (finding

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

begin
  $\Omega :=$ *basis* schedule
  $\Psi :=$ OptimalTree$(\Omega, \emptyset, -, -)$
end

Figure 9: Algorithm OptimalTree()

a tree of schedules) makes it so that it requires exponential time and memory, even when using a polynomial-time heuristic in the interval-partitioning step. Additionally, even if we can afford to compute the optimal tree of schedules (as this is done off-line), the size of the tree might still be too large to fit in the available memory resources of the target system. Therefore a suboptimal set of schedules and switching points must be chosen such that the memory constraints imposed by the target system are satisfied. Solutions aiming to tackle different complexity issues are addressed in Section 7.

The optimal set of schedules is stored in the dedicated shared memory of the system as an ordered tree. Upon completing a task, the cost of selecting at run-time, by the quasi-static scheduler, the execution order for the remaining tasks is $O(logN)$ where N is the maximum number of children that a node has in the tree of schedules. Such cost can be included in our analysis procedure by augmenting accordingly the maximum duration of tasks.

7 Heuristics and Experimental Evaluation

This section presents several heuristic methods that tackle different complexity dimensions of the multiple-schedules problem, namely the interval-partitioning step and the exponential growth of the tree size.

7.1 Interval Partitioning

In the interval-partitioning step, when finding which schedules deliver the highest utility after completing a task t_i in an interval I^i of possible completion times, the optimal algorithm explores all the permutations of tasks not yet started that define feasible schedules Ω_j and accordingly computes $\hat{U}_j(\tau_i)$. In order to avoid computing $\hat{U}_j(\tau_i)$ for all such permutations, we propose a heuristic that instead considers only two schedules Ω_{L} and Ω_{U}, computes $\hat{U}_{\mathsf{L}}(\tau_i)$ and $\hat{U}_{\mathsf{U}}(\tau_i)$, and partitions I^i based on these two. These two schedules Ω_{L} and Ω_{U} correspond, respectively, to the solutions to the *one-schedule*

Algorithm OPTIMALTREE(Ω, A, t, I)
input: A schedule Ω, the set A of already completed tasks, the last completed task t, and the interval I of completion times for t
output: The optimal tree Ψ of schedules to follow after completing t at $\tau \in I$

```
begin
  set Ω as root of Ψ
  compute the set C of concurrent tasks
  for i ← 1, 2, ... , |C| do
    if t_i may complete before than the other t ∈ C then
      compute the interval I^i when t_i may complete first
      for j ← 1, 2, ... , |T \ A \ C|! do
        if Ω_j is valid then
          compute Û_j(τ_i)
        end if
      end for
      partition I^i into subintervals I^i_1, I^i_2, ... , I^i_K s.t. σ_k makes Û_k(τ_i)
      maximal in I^i_k
      A_i := A ∪ {t_i}
      for k ← 1, 2, ... , K do
        Ψ_k := OPTIMALTREE(Ω_k, A_i, t_i, I^i_k)
        add subtree Ψ_k s.t. Ω --t_i;I^i_k--> Ω_k
      end for
    end if
  end for
end
```

Figure 10: Algorithm OPTIMALTREE(Ω, A, t, I)

problem (see Section 4) for the lower and upper limits τ_{L} and τ_{U} of the interval I^i. For the example discussed in Sections 3 and 6, when partitioning the interval $I^1 = [2, 10]$ of possible completion times of t_1 (case when t_1 completes after t_2), the heuristic, called LIMTREE, solves the one-schedule problem for $\tau_{\mathsf{L}} = 2$ and $\tau_{\mathsf{U}} = 10$. The respective solutions are $\Omega_{\mathsf{L}} = \{t_1t_5t_3, t_2t_4t_7t_6\}$ and $\Omega_{\mathsf{U}} = \{t_1t_3t_5, t_2t_4t_6t_7\}$. Then LIMTREE computes $\hat{U}_{\mathsf{L}}(\tau_1)$ and $\hat{U}_{\mathsf{U}}(\tau_1)$ (which correspond, respectively, to $\hat{U}_a(\tau_1)$ and $\hat{U}_d(\tau_1)$ in Figure 7(b)) and partitions I^1 using only these two. In this step, the solution given by LIMTREE is, after t_1: follow Ω_{L} if $2 \leq \tau_1 \leq 4$; follow Ω_{U} if $4 < \tau_1 \leq 10$. The reader can note that in this case LIMTREE gives a suboptimal solution (see the optimal tree in Figure 8). The pseudocode of the heuristic LIMTREE is presented in Figure 11.

Along with the proposed heuristic we must solve the one-schedule prob-

Algorithm LIMTREE(Ω, A, t, I)
input: A schedule Ω, the set A of already completed tasks, the last completed task t, and the interval I of completion times for t
output: The tree Ψ of schedules to follow after completing t at $\tau \in I$

begin
  set Ω as root of Ψ
  compute the set C of concurrent tasks
  for $i \leftarrow 1, 2, \ldots, |C|$ **do**
    if t_i may complete before than the other $t \in C$ **then**
      compute the interval I^i when t_i may complete first
      τ_{L} := lower limit of I^i
      Ω_{L} := sol. *one-sch. problem* for τ_{L}
      compute $\hat{U}_{\mathsf{L}}(\tau_i)$
      τ_{U} := upper limit of I^i
      Ω_{U} := sol. *one-sch. problem* for τ_{U}
      compute $\hat{U}_{\mathsf{U}}(\tau_i)$
      partition I^i into subintervals $I^i_1, I^i_2, \ldots, I^i_K$ s.t. σ_k makes $\hat{U}_k(\tau_i)$ maximal in I^i_k
      $A_i := A \cup \{t_i\}$
      for $k \leftarrow 1, 2, \ldots, K$ **do**
        Ψ_k := LIMTREE($\Omega_k, A_i, t_i, I^i_k$)
        add subtree Ψ_k s.t. $\Omega \xrightarrow{t_i; I^i_k} \Omega_k$
      end for
    end if
  end for
end

Figure 11: Algorithm LIMTREE(Ω, A, t, I)

lem, which itself is intractable. We have proposed an exact algorithm and a number of heuristics for the one-schedule problem in Section 5. For the experimental evaluation of LIM we have used the exact algorithm and the two heuristics when solving the one-schedule problem. Hence we have three heuristics LIM$_A$, LIM$_B$, and LIM$_C$ for the multiple-schedules problem. The first uses the optimal algorithm for the one-schedule problem while the second and third make use of the two heuristics presented in Section 5.

We have generated a large number of synthetic examples in order to evaluate the quality of the heuristics. For the examples used throughout the experimental evaluation of this subsection, we have considered that, out of the n tasks of the system, $(n-2)/2$ are soft and $(n-2)/2$ are hard. The tasks are mapped on architectures consisting of between 2 and 4 processors. We generated 100 synthetic systems for each graph dimension. All the experi-

ments presented in this paper were run on a Sun Ultra 10 workstation.

The average size of the tree of schedules, when using the optimal algorithm (Section 6) as well as the above heuristics, is shown by the plot of Figure 12. Note the exponential growth even in the heuristic cases. This is inherent to the problem of computing a tree of schedules.

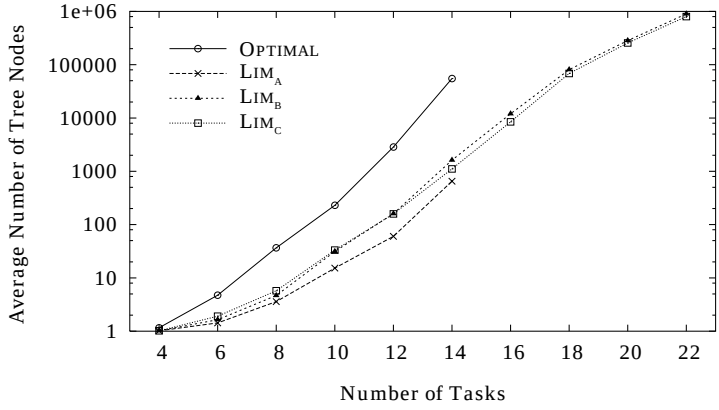


Figure 12: Average size of the tree of schedules

The average execution time for constructing the tree of schedules by the different algorithms is shown in Figure 13. 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 systems. The long execution times for LIM_A, only slightly less than the algorithm OPTIMAL, are due to the fact that, along the construction of the tree, it solves the one-schedule problem using an exact algorithm. The other heuristics, LIM_B and LIM_C, take significantly less time because of the use of polynomial-time heuristics in the interval-partitioning step when solving the one-schedule problem. However, due to the exponential growth of the tree size (see Figure 12), even LIM_B and LIM_C require exponential time.

We have evaluated the quality of the trees generated by 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. For each case, we obtained the total utility yielded by a given tree of schedules and normalized it with respect to the one produced by the optimal tree:

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

The average normalized utility, as given by trees computed using different algorithms, is shown in Figure 14. We have also plotted the case of a static solution where only one schedule is used regardless of the actual execution times (SINGLESCH). This plot shows LIMTREE_A as the best of the heuristics

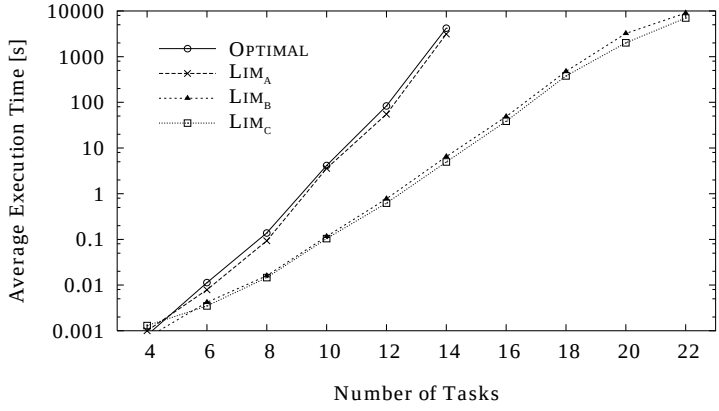


Figure 13: Average execution time

discussed above, in terms of the total utility yielded by the trees it produces. LimTree$_B$ produces 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.

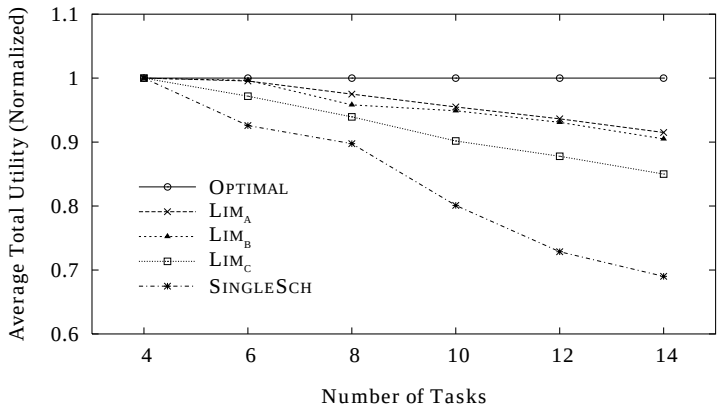


Figure 14: Average normalized total utility

7.2 Limiting the Tree Size

Even if we can afford to fully compute the optimal tree of schedules, the tree might be too large to fit in the available memory of the system under consideration. 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 7.1 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 Ω_k of a node Ω (schedules Ω_k to be followed after completing a task in Ω) or none are added to the tree. The pseudocode for the generic algorithm is presented in Figure 15. The schedules to follow after Ω correspond to those obtained in the interval-partitioning step as described in Sections 6 and 7.1. 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 Ω_i ($\textsc{Sort}(\Omega_1, \Omega_2, \ldots, \Omega_c)$) in Figure 15).

```
Algorithm ConstructTree(Ω, max)
input: A schedule Ω and a positive integer max
output: A tree Ψ limited to max nodes whose root is Ω
begin
  set Ω as root of Ψ
  m := max − 1
  c := number of schedules to follow after Ω
  if 1 < c ≤ m then
    add Ω_1, Ω_2, ..., Ω_c as children of Ω
    m := m − c
    Sort(Ω_1, Ω_2, ..., Ω_c)
    for i ← 1, 2, ..., c do
      Ψ_i :=ConstructTree(Ω_i, m + 1)
      n_i := size of Ψ_i
      m := m − n_i + 1
    end for
  end if
end
```

Figure 15: Algorithm $\textsc{ConstructTree}(\Omega, max)$

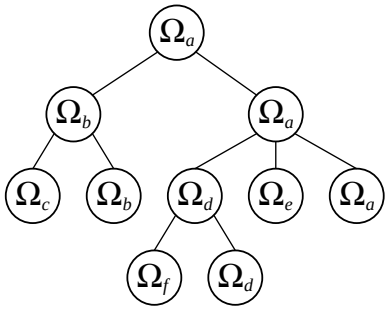


Figure 16: A complete tree of schedules

Initially we have studied two simple heuristics for constructing a tree, given a maximum size M. The first one, called DIFF, gives priority to subtrees derived from nodes whose schedules differ from their parents. We use a similarity metric, based on the concept of Hamming distance, in order to determine how similar two schedules are. If, for instance, while constructing a tree with a size limit $\mathsf{M} = 8$ for the system whose optimal tree is the one given in Figure 16, we find out that, after the initial schedule Ω_a (the root of the tree), either Ω_b must be followed or Ω_a continues as the execution order for the remaining tasks, depending on the completion time of a certain task. Therefore we add Ω_b and Ω_a to the tree. Then, when using DIFF, the size budget is assigned first to the subtrees derived from Ω_b and the process continues until we obtain the tree shown in Figure 17. The second approach, EQ, gives priority to nodes that are equal or more similar to their parents. The tree obtained when using EQ and having a size limit $\mathsf{M} = 8$ is shown in Figure 18. Experimental data (see Figure 19) shows that in average EQ outperforms DIFF. The basic idea when using EQ is that, since no change has yet been operated on the previous schedule, it is likely that several possible alternatives are potentially detected in the future. Hence it pays off to explore the possible changes of schedules derived from such branches. On the contrary, if a different schedule has been detected, it can be assumed that this one is relatively well adapted to the new situation and possible future changes are not leading to dramatic improvements.

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 may determine, for a particular execution order, 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

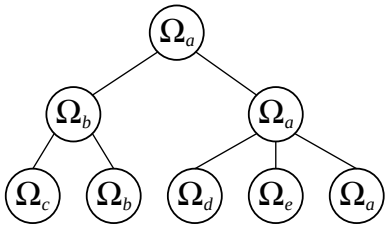


Figure 17: Tree constructed using DIFF

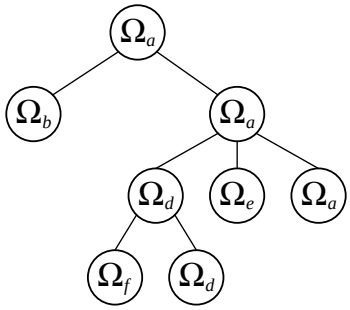


Figure 18: Tree constructed using EQ

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 proposed approaches, we have randomly generated 100 systems with a fix number of 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 shown in Figure 19 shows that PROB is the algorithm that gives the best results in average.

We have further investigated the combination of PROB and EQ 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 $wp + (1-w)s$, where p is the probability of that node (schedule) being selected among its siblings

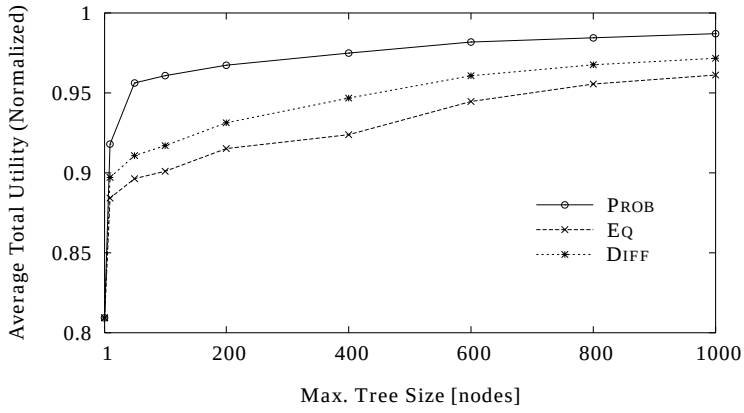


Figure 19: Evaluation of the construction algorithms

and s is a factor that captures how similar that node and its parent are. Note that the particular cases $w = 0$ and $w = 1$ correspond to EQ and PROB respectively. The results of the weighted approach for different values of w are illustrated in Figure 20. It is interesting to note that we can get even better results than PROB for certain weights, with $w = 0.9$ being the one that performs the best.

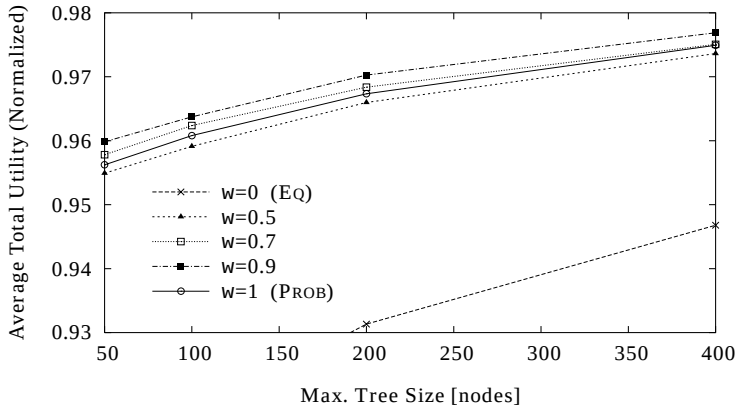


Figure 20: Construction algorithms using a weighted approach

8 Cruise Control with Collision Avoidance

Modern vehicles can be equipped with sophisticated electronic aids aiming to assist the driver, increase efficiency, and enhance on-board comfort. One such system is the Cruise Control with Collision Avoidance (CCCA) [10] 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 another vehicle or 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 lane in front of the car, the CCCA will adjust the speed by applying limited braking to maintain a given 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 period: $\mathsf{T}_{BC} = 100$ ms, $\mathsf{T}_{EC} = 250$ ms, $\mathsf{T}_{CA} = 125$ ms, and $\mathsf{T}_{DC} = 500$ ms. We have modeled each subsystem as a task graph. Each subsystem has one hard deadline that equals its 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 have considered an architecture with two processors that communicate through a bus, and assumed that the dedicated memory for storing the schedules has a capacity of 64 kB. We generated several instances of the task graphs of the four subsystems mentioned above in order to construct a graph with a period $\mathsf{T} = 500$ ms (least common multiple of the periods of the involved tasks). The resulting graph, including processing as well as communication activities, contains 126 tasks, out of which 6 are soft and 12 are hard.

Assuming that we need 100 B for storing one schedule, we have an upper limit of 640 nodes in the tree. We have constructed the tree of schedules using the approaches discussed in Section 7.2 combined with one of the heuristics presented in Section 7.1 ($\textsc{Lim}_B$).

Due to the size of the system, it is infeasible to fully construct the complete tree of schedules. Therefore, we have instead compared the tree limited to 640 nodes with the static, off-line solution of a single schedule. The results are presented in Table 1. We can achieve with our quasi-static approach, in this case of the CCCA, a gain of around 40% as compared to a single static schedule. For this example, the weighted approach does not produce further

improvements, which is explained by the fact that EQ and PROB give very similar results.

	Average Total Utility	Gain with respect to SINGLESCH
SINGLESCH	6.51	—
DIFF	7.51	11.42%
EQ	9.54	41.54%
PROB	9.6	42.43%

Table 1: Quality of different approaches for the CCCA

9 Conclusions

We have presented an approach to the problem of scheduling for multiprocessor 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. The problem we have addressed is that of finding an execution order such that the total utility is maximal and, at the same time, satisfaction of hard deadlines is guaranteed.

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 scheduler only has to select, depending on the actual execution times, one of the precomputed schedules.

We have proposed a procedure that computes the optimal tree of schedules and switching points, that is, a tree that delivers the same utility as an ideal on-line scheduler. Several heuristics, that address different complexity dimensions of the problem, have also been presented. These heuristics allow to generate good quality schedule trees for large applications, even in the context of limited resources. We have used a large number of synthetic examples and a real-life application in order to demonstrate the efficiency of our approach.

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] J. Cortadella, A. Kondratyev, L. Lavagno, and Y. Watanabe. Quasi-Static Scheduling for Concurrent Architectures. In *Proc. Intl. Conference on Application of Concurrency to System Design*, pages 29–40, 2003.

[6] 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. Available from `http://www.ida.liu.se/~luico`.

[7] L. A. Cortés, P. Eles, and Z. Peng. Quasi-Static Scheduling for Real-Time Systems with Hard and Soft Tasks. In *Proc. DATE Conference*, 2004. (to be published).

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] D. Prasad, A. Burns, and M. Atkins. The Valid Use of Utility in Adaptive Real-Time Systems. *Real-Time Systems*, 25(2-3):277–296, Sept. 2003.

[16] 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.

[17] 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.

[18] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Synthesis of Embedded Software Using Free-Choice Petri Nets. In *Proc. DAC*, pages 805–810, 1999.

[19] C.-S. Shih, S. Gopalakrishnan, P. Ganti, M. Caccamo, and L. Sha. Template-Based Real-Time Dwell Scheduling with Energy Constraints. In *Proc. Real-Time and Embedded Technology and Applications Symposium*, pages 19–27, 2003.

[20] 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.

[21] 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.

[22] F.-S. Su and P.-A. Hsiung. Extended Quasi-Static Scheduling for Formal Synthesis and Code Generation of Embedded Software. In *Proc. CODES*, pages 211–216, 2002.