in *Proc. Intl. Conference on Real-Time and Embedded Computing Systems and Applications*, 2005, pp. 422-428.

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

Luis Alejandro Cortés [1,2]
alejandro.cortes@volvo.com

Petru Eles [2]
petel@ida.liu.se

Zebo Peng [2]
zebpe@ida.liu.se

[1] Volvo Truck Corporation
Gothenburg, Sweden

[2] Linköping University
Linköping, Sweden

Abstract

We address in this paper the problem of scheduling for multiprocessor real-time systems with hard and soft tasks. Utility functions are associated to soft tasks to capture their relative importance and how the quality of results is affected when a soft deadline is missed. The problem is to find a task execution order that maximizes the total utility and guarantees the hard deadlines. In order to account for actual execution times, we consider time intervals for tasks rather than fixed execution times. A single static schedule computed off-line is pessimistic, while a purely on-line approach, which computes a new schedule every time a task completes, incurs an unacceptable overhead. We propose therefore a quasi-static solution where a number of schedules are computed at design-time, leaving 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. We evaluate our approach through synthetic examples and a realistic application.

1 Introduction

Scheduling for hard/soft real-time systems has been addressed, for example, in the context of integrating multimedia and hard real-time tasks [1]. Most of the previous work on scheduling for hard/soft real-time systems 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. This problem has been considered under both dynamic [2] and fixed priority assignments [5]. 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 use 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 [7] for representing the criticality of tasks.

In this paper we consider multiprocessor 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 all hard deadlines are guaranteed. 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, it is possible to exploit information regarding execution time intervals to obtain schedules that yield higher utilities, that is, improve the quality of results.

Utility-based scheduling [8] has been addressed before, for instance, in the frame of imprecise computation techniques [10]. These assume tasks 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 to find 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 that tasks have no optional part. 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).

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 deadlines. *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 the hard deadlines, but considering the actual execution times of those tasks 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 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 we address the problem of *quasi-static* scheduling for multiprocessor hard/soft real-time systems.

Quasi-static scheduling has been studied previously, mostly in the context of formal synthesis and without considering an explicit notion of time but only the partial order of events [9]. In a previous work we have discussed quasi-static scheduling for hard/soft systems in the particular case of monoprocessor systems [4], a problem whose analysis complexity is significantly lower than when considering multiple processors: one of the sources of additional complexity in the case of multiprocessor systems is the interleaving of possible finishing orders for concurrent tasks, which makes that the set of computed schedules grows very fast (as discussed in Section 5); another important source of complexity is that, in the case of multiprocessor systems, the off-line preparation of schedules must consider that, when a task T_i completes, tasks running on other processors may still be under execution, and therefore the analysis must take care that the schedules to be selected upon completing T_i are consistent with tasks still under execution.

2 Preliminaries

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

The mapping of tasks is defined by a function $m : \mathbf{T} \to \mathbf{PE}$ where $\mathbf{PE}$ 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 $T \in \mathbf{C}$ then $m(T) \in \mathbf{B}$, where $\mathbf{C} \subset \mathbf{T}$ is the set of communication tasks and $\mathbf{B} \subset \mathbf{PE}$ is the set of buses. The issue of assigning tasks to processing elements (processors and buses) is beyond the scope of this paper. We consider that the mapping of tasks to processing elements is already determined and given as input to the problem.

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 Fig. 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 $U = \sum_{T_i \in \mathbf{S}} u_i(t_i)$.

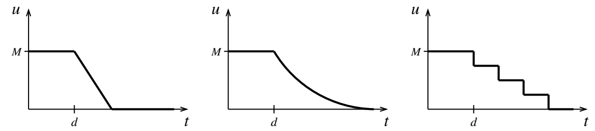


Fig. 1. Typical utility functions for soft tasks

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 $\tau_i^{\text{bc}} \leq \tau_i \leq \tau_i^{\text{wc}}$ (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 use $^\circ T$ to denote the set of direct predecessors of task T, that is, $^\circ T = \{T' \in \mathbf{T} \mid (T', T) \in \mathbf{E}\}$. Similarly, $T^\circ = \{T' \in \mathbf{T} \mid (T, T') \in \mathbf{E}\}$ is the set of direct successors of T.

We consider that tasks are periodic and non-preemptable: all the tasks in a task graph have the same period and become ready at the same time. 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 $\{\sigma^{(1)} : \mathbf{T}^{(1)} \to \{1, 2, \ldots, |\mathbf{T}^{(1)}|\}, \ldots, \sigma^{(p)} : \mathbf{T}^{(p)} \to \{1, 2, \ldots, |\mathbf{T}^{(p)}|\}\}$ where $\mathbf{T}^{(i)} = \{T \in \mathbf{T} \mid m(T) = PE_i\}$ is the set of tasks mapped onto the processing element PE_i and $|\mathbf{T}^{(i)}|$ denotes the cardinality of the set $\mathbf{T}^{(i)}$. In the particular case of monoprocessor systems, a schedule is just one bijection $\sigma : \mathbf{T} \to \{1, 2, \ldots, |\mathbf{T}|\}$. We use the notation $\sigma^{(i)} = T_{i_1} T_{i_2} \ldots T_{i_n}$ as shorthand for $\sigma^{(i)}(T_{i_1}) = 1, \sigma^{(i)}(T_{i_2}) = 2, \ldots, \sigma^{(i)}(T_{i_n}) = |\mathbf{T}^{(i)}|$. A schedule Ω as defined above captures the task execution order for each one of the processing elements.

If a system, however, contains task graphs with different periods we can still handle it by generating several instances of the task graphs and building a graph that corresponds to a set of task graphs as they occur within their hyperperiod (least common multiple of the periods of the involved tasks), in such a way that the new task graph has one period equal to the aforementioned hyperperiod.

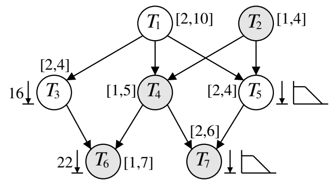
For a given schedule, the starting and completion times of a task T_i are denoted s_i and t_i respectively, with $t_i = s_i + \tau_i$. Thus, for $\sigma^{(k)} = T_1 T_2 \ldots T_n$, task T_1 will start executing at $s_1 = \max_{T_j \in {}^\circ T_1}\{t_j\}$ and task T_i, $1 < i \leq n$, will start executing at $s_i = \max(\max_{T_j \in {}^\circ T_i}\{t_j\}, t_{i-1})$. 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 in the task graph has starting time $s = 0$ if $\sigma^{(k)}(T) = 1$. For example, in a monoprocessor system, 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 the 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 schedule to another. 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 $a \leq t_i \leq b$, 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 as included in the worst-case duration of tasks. Upon finishing a task running on a certain processing element, a new schedule can be selected which will then be followed by 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 multiprocessor system shown in Fig. 2. Tasks T_1, T_3, T_5 are mapped on processor PE_1 and tasks T_2, T_4, T_6, T_7 are mapped on PE_2. For the sake of simplicity, we have ignored inter-processor communication in this example. The best-case and worst-case duration of every task, considering the given mapping, are shown in Fig. 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 $[\tau_i^{\text{bc}}, \tau_i^{\text{wc}}]$. Tasks T_3 and T_6 are hard and their deadlines are $d_3 = 16$ and $d_6 = 22$ respectively. Tasks T_5 and T_7 are soft and their utility functions are given in Fig. 2.



$$u_5(t_5) = \begin{cases} 2 & \text{if } t_5 \leq 5, \\ 3 - \dfrac{t_5}{5} & \text{if } 5 \leq t_5 \leq 15, \\ 0 & \text{if } t_5 \geq 15. \end{cases} \quad u_7(t_7) = \begin{cases} 3 & \text{if } t_7 \leq 3, \\ \dfrac{18}{5} - \dfrac{t_7}{5} & \text{if } 3 \leq t_7 \leq 18, \\ 0 & \text{if } t_7 \geq 18. \end{cases}$$

Fig. 2. Motivational example

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 Fig. 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 this section we use the simplified notation $\Omega = \{T_1 T_3 T_5, T_2 T_4 T_6 T_7\}$). We have proved that the problem of computing one such optimal schedule is **NP**-hard even in the monoprocessor case [3].

Although $\Omega = \{T_1 T_3 T_5, T_2 T_4 T_6 T_7\}$ is optimal in the static sense discussed above, it is still pessimistic because the actual execution times (not known in advance) 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 Ω, that is T_1 and T_2 start at $s_1 = s_2 = 0$. Assume that T_2 completes at $t_2 = 4$ and then T_1 completes at $t_1 = 6$. At this point, exploiting the fact that we know the completion times t_1 and t_2, 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_2—already executed—and expected duration for T_3, T_4, T_5, T_6, T_7—remaining tasks), and also guarantees the hard deadlines (even if all remaining tasks execute with their worst-case duration). Such a schedule is $\Omega' = \{T_1T_5T_3, T_2T_4T_6T_7\}$. For $\tau_1 = 6$, $\tau_2 = 4$, and $\tau_i = \tau_i^{\mathrm{e}}$ for $3 \le i \le 7$, 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, the hard deadlines are 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. 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 $\Omega \xrightarrow{T_1;[2,6]} \Omega'$ for the example given in Fig. 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 $2 \le t_1 \le 6$ 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 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 of 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 of 1.1, yet guaranteeing the hard deadlines). 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.

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

Find a multiprocessor schedule Ω (set of p bijections $\{\sigma^{(1)} : \mathbf{T}^{(1)} \to \{1, 2, \ldots, |\mathbf{T}^{(1)}|\}, \ldots, \sigma^{(p)} : \mathbf{T}^{(p)} \to \{1, 2, \ldots, |\mathbf{T}^{(p)}|\}\}$ with $\mathbf{T}^{(l)}$ being the set of tasks mapped onto the processing element PE_l and p being the number of processing elements) that maximizes $U = \sum_{T_i \in \mathbf{S}} u_i(t_i^{\mathrm{e}})$ where t_i^{e} is the expected completion time of task T_i, subject to: no deadlock[1] is introduced by Ω; $t_i^{\mathrm{wc}} \le d_i$ for all $T_i \in \mathbf{H}$, where t_i^{wc} is the worst-case completion time of task T_i; each $\sigma^{(l)}$ has a prefix $\sigma_x^{(l)}$, with $\sigma_x^{(l)}$ being the order of the tasks already executed or under execution on processing element PE_l.

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 $\tau_1, \tau_2, \ldots, \tau_n$ (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 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 multiprocessor schedules and switching points such that, for any set of execution times $\tau_1, \tau_2, \ldots, \tau_n$, 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 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 $U^{ideal}_{\{\tau_i\}}$. 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_i can be any value in $[\tau_i^{\mathrm{bc}}, \tau_i^{\mathrm{wc}}]$) 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

[1] When considering a task graph with its original edges together with additional edges defined by the partial order corresponding to the schedule, the resulting task graph must be acyclic.

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 the hard deadlines, depending on the completion time t_k.

We may thus determine (off-line) the schedule that must be followed after completing task T at a particular time t. For each schedule Ω_i that satisfies the precedence constraints and is consistent with the tasks so far executed, we express the total utility $U_i(t)$ as a function of the completion time t (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 t imply potential hard deadline misses when Ω_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 Ω_i, after T has completed at t, does not guarantee the hard deadlines, 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*. 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.

We use the example in Fig. 2 to illustrate the procedure. The initial schedule is $\Omega = \{T_1T_3T_5, T_2T_4T_6T_7\}$, that is, T_1 and T_2 start executing concurrently 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 $(2 \le t_1 \le 4)$; T_2 completes before T_1 $(1 \le t_2 \le 4)$. For the first one, we compute $\hat{U}_i(t_1)$, $2 \le t_1 \le 4$, 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 $t_1 \in [2, 4]$. 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 (see Fig. 4). Details of the interval-partitioning step are illustrated next.

Let us continue with the branch corresponding to T_2 completing first in the interval $[1, 4]$. Under these 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 $U_a(t_1)$, $U_b(t_1)$, $U_c(t_1)$, and $U_d(t_1)$, $2 \le t_1 \le 10$, considering the expected duration for T_3, T_4, T_5, T_6, T_7. For example, $U_d(t_1) = u_5(t_1 + \tau_5^{\mathrm{e}}) + u_7(t_1 + \max(\tau_4^{\mathrm{e}}, \tau_5^{\mathrm{e}}) + \tau_7^{\mathrm{e}}) = u_5(t_1 + 3) + u_7(t_1 + 7)$. We get the functions shown in Fig. 3(a)[2].

Now, for Ω_a, Ω_b, Ω_c, and Ω_d, we compute the latest completion time t_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 $t_3 = t_1 + \tau_5^{\mathrm{wc}} + \tau_3^{\mathrm{wc}} = t_1 + 8$ and $t_6 = \max(t_3, t_1 + \tau_4^{\mathrm{wc}}) + \tau_6^{\mathrm{wc}} = \max(t_1 + 8, t_1 + 5) + 7 = t_1 + 15$. Since the hard deadlines for this system are $d_3 = 16$ and $d_6 = 22$, when Ω_c is followed, $t_3 \le 16$ and $t_6 \le 22$ if and only if $t_1 \le 7$. A similar analysis shows the following: Ω_a guarantees the hard deadlines for any completion time $t_1 \in [2, 10]$; Ω_b implies potential hard deadline misses for any $t_1 \in [2, 10]$; Ω_d guarantees the hard deadlines if and only if $t_1 \le 4$. Thus we get auxiliary functions as shown in Fig. 3(b).

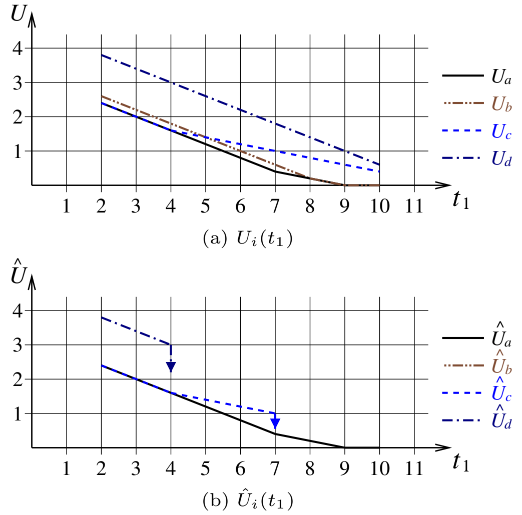


Fig. 3. $U_i(t_1)$ and $\hat{U}_i(t_1)$, $2 \le t_1 \le 10$, for the ex. of Fig. 2

From the graph in Fig. 3(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: $\Omega_d = \{T_1T_5T_3, T_2T_4T_7T_6\}$ if $2 \le t_1 \le 4$; $\Omega_c = \{T_1T_5T_3, T_2T_4T_6T_7\}$ if $4 < t_1 \le 7$; $\Omega_a = \{T_1T_3T_5, T_2T_4T_6T_7\}$ if $7 < t_1 \le 10$.

The optimal tree of schedules for the system shown in Fig. 2 is presented in Fig. 4. 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. For example, this is the case of the schedule $\{T_1T_5T_3, T_2T_4T_7T_6\}$ 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. Other regularities of the tree can be exploited in order to store it in a more compact way. We have considered these in our implementation but they are not discussed in this paper.

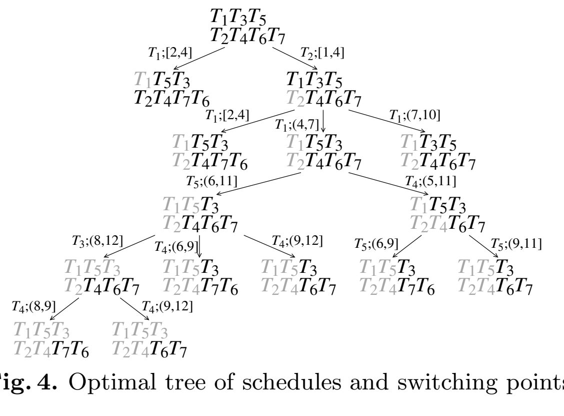


Fig. 4. Optimal tree of schedules and switching points

The schedules are 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 $\mathcal{O}(logN)$ where

[2]For instance,

$$U_c(t_1) = \begin{cases} 16/5 - 2t_1/5 & \text{if } 2 \le t_1 \le 4, \\ 12/5 - t_1/5 & \text{if } 4 \le t_1 \le 10. \end{cases}$$

N is the maximum number of children that a node has in the tree of schedules. Such cost can be included in our analysis by augmenting accordingly the worst-case duration of tasks.

6 Heuristics and Experimental Evaluation

In the optimal algorithm presented in Section 5, when finding the schedule to follow after completing a task T in an interval of possible completion times t, it is necessary to analyze all the schedules that respect the data dependencies and are consistent with the tasks already executed. Hence the interval-partitioning step requires $\mathcal{O}(|\mathbf{T}|!)$ time in the worst case. Moreover, the inherent nature of the problem (finding 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 (this is done off-line), the size of the tree might 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 tackling different complexity dimensions of the problem are addressed in this section.

6.1 Interval Partitioning

When partitioning an interval $\mathcal{I}^i$ of possible completion times t_i, the optimal algorithm explores all the permutations of tasks not yet started that define feasible schedules Ω_j and accordingly computes $\hat{U}_j(t_i)$. In order to avoid computing $\hat{U}_j(t_i)$ for all such permutations, we propose a heuristic that instead considers only two schedules Ω_L and Ω_U, computes $\hat{U}_\mathsf{L}(t_i)$ and $\hat{U}_\mathsf{U}(t_i)$, and partitions $\mathcal{I}^i$ based on these two functions. The schedules Ω_L and Ω_U correspond, respectively, to the solutions to the *one-schedule problem* (see Section 4) for the lower and upper limits t_L and t_U of the interval $\mathcal{I}^i$. For other completion times $t_i \in \mathcal{I}^i$ different from t_L, Ω_L is rather optimistic but it might happen that it does not guarantee hard deadlines. On the other hand, Ω_U can be pessimistic but does guarantee hard deadlines for all $t_i \in \mathcal{I}^i$. Thus, by combining the optimism of Ω_L with the guarantees provided by Ω_U, good quality solutions can be obtained. The pseudocode of the heuristic, called Lim, is presented in Fig. 5.

Algorithm Lim$(\Omega, \mathbf{A}, T_l, \mathcal{I}^l)$
input: A schedule Ω, the set $\mathbf{A}$ of already completed tasks, the last completed task T_l, and the interval $\mathcal{I}^l$ of completion times for T_l
output: The tree Ψ of schedules to follow after completing T_l at $t_l \in \mathcal{I}^l$

1: set Ω as root of Ψ
2: compute the set $\mathbf{C}$ of concurrent tasks
3: **for** $i \leftarrow 1, 2, \ldots, |\mathbf{C}|$ **do**
4:   **if** T_i may complete before the other $T_c \in \mathbf{C}$ **then**
5:     compute interval $\mathcal{I}^i$ when T_i may complete first
6:     t_L := lower limit of $\mathcal{I}^i$; t_U := upper limit of $\mathcal{I}^i$
7:     Ω_L := solution *one-sch. problem* for t_L; Ω_U := solution *one-sch. problem* for t_U
8:     compute $\hat{U}_\mathsf{L}(t_i)$ and $\hat{U}_\mathsf{U}(t_i)$
9:     partition $\mathcal{I}^i$ into subintervals $\mathcal{I}^i_1, \mathcal{I}^i_2, \ldots, \mathcal{I}^i_K$ s.t. Ω_k makes $\hat{U}_k(t_i)$ maximal in $\mathcal{I}^i_k$
10:     $\mathbf{A}_i := \mathbf{A} \cup \{T_i\}$
11:     **for** $k \leftarrow 1, 2, \ldots, K$ **do**
12:       Ψ_k := Lim$(\Omega_k, \mathbf{A}_i, T_i, \mathcal{I}^i_k)$
13:       add subtree Ψ_k s.t. $\Omega \xrightarrow{T_i;\mathcal{I}^i_k} \Omega_k$
14:     **end for**
15:   **end if**
16: **end for**

Fig. 5. Algorithm Lim$(\Omega, \mathbf{A}, T_l, \mathcal{I}^l)$

For the example discussed in Sections 3 and 5, when partitioning the interval $\mathcal{I}^1 = [2, 10]$ of possible completion times of T_1 (case when T_1 completes after T_2), the heuristic solves the one-schedule problem for $t_\mathsf{L} = 2$ and $t_\mathsf{U} = 10$. 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 Lim computes $\hat{U}_\mathsf{L}(t_1)$ and $\hat{U}_\mathsf{U}(t_1)$ (which correspond, respectively, to $\hat{U}_a(t_1)$ and $\hat{U}_d(t_1)$ in Fig. 3(b)) and partitions $\mathcal{I}^1$ using only these two functions. In this step, the solution given by Lim is, after T_1: follow Ω_L if $2 \leq t_1 \leq 4$; follow Ω_U if $4 < t_1 \leq 10$. The reader can note that in this case Lim gives a suboptimal solution (see Fig. 3(b) and the optimal tree in Fig. 4).

Along with the proposed heuristic we must solve the one-schedule problem (line 7 in Fig. 5), which itself is intractable. We have proposed an exact algorithm and a number of heuristics for the one-schedule problem [3]. For the experimental evaluation of Lim we have used the exact algorithm and 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 an optimal algorithm for the one-schedule problem while the second and third use two different heuristics presented in [3].

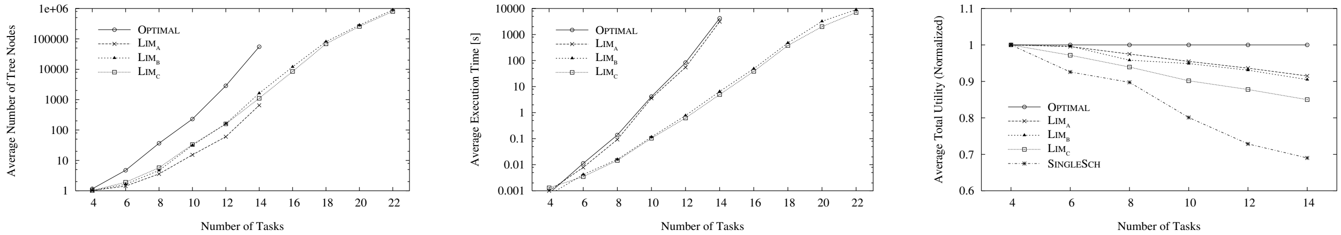
Observe that the heuristics presented in this section address only the interval-partitioning step and, in isolation, cannot handle the large complexity of the multiple-schedules problem. These heuristics are to be used in conjunction with the methods discussed in Section 6.2.

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 in 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.

The average size of the tree of schedules, when using the optimal algorithm (Section 5) as well as the above heuristics, is shown by the plot of Fig. 6(a). Note the exponential growth even in the heuristic cases. This is inherent to the problem of computing a tree of schedules.

The average execution time for constructing the tree of schedules is shown in Fig. 6(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 systems. The long execution times for Lim$_A$, slightly less than the algorithm Optimal, are due to the fact that, along the construction of the tree, it solves the one-schedule problem (itself intractable) using an exact algorithm. The other heuristics, Lim$_B$ and Lim$_C$, take significantly less time because of the use of polynomial-time heuristics for the one-schedule problem during the interval-partitioning step. However, due to the exponential growth of the tree size (see Fig. 6(a)), even Lim$_B$ and Lim$_C$ require exponential time.

We evaluated the quality of the trees generated by different algorithms with respect to the optimal tree. For each one of the 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 Fig. 6(c). We have also plotted the case of a static solution where only one schedule is used regardless of the actual execution times (SingleSch), which is the optimal solution in the static scheduling case. This plot shows Lim$_A$ as the best of the heuristics discussed above, in terms of the total utility yielded by the trees it produces. Lim$_B$ produces still good results, not very far from the optimal, at a significantly lower computational cost. Having one single static schedule leads to a considerable quality loss, even if the static solution is



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

Fig. 6. Evaluation of the algorithms for computing a tree of schedules

optimal (in the sense as being the best static solution) while the quasi-static is suboptimal (produced by a heuristic).

6.2 Tree Size Restriction

Even if we could afford to compute the optimal tree of schedules (which is not the case for large examples due to the time and memory constraints at design-time), the tree might be too large to fit in the available memory of the target system. Hence we must drop some nodes of the tree at the expense of the solution quality. The heuristics presented in Section 6.1 reduce considerably both the time and memory needed to construct a tree as compared to the optimal algorithm, but still require exponential time and memory. In this section, on top of the above heuristics, we propose methods that construct a tree considering its size limit (imposed by the designer) in such a way that we can handle both the time and memory complexity.

Given a limit for the tree size, only a certain number of schedules can be generated. 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 the generic algorithm RESTR given in Fig. 7. The schedules $\Omega_1, \Omega_2, \ldots, \Omega_K$ to follow after Ω correspond to those obtained in the interval-partitioning step as described in Sections 5 and 6.1. The difference among the approaches discussed in this section lies in the order in which the available memory budget is assigned to trees derived from the nodes Ω_k (line 7 in Fig. 7): SORT$(\Omega_1, \Omega_2, \ldots, \Omega_K)$ gives this order according to different criteria.

Algorithm RESTR(Ω, M)
input: A schedule Ω and a positive integer M
output: A tree Ψ limited to M nodes whose root is Ω

1: set Ω as root of Ψ
2: $m := M - 1$
3: find the schedules $\Omega_1, \Omega_2, \ldots, \Omega_K$ to follow after Ω (interval-partitioning step)
4: **if** $1 < K \leq m$ **then**
5: add $\Omega_1, \Omega_2, \ldots, \Omega_K$ as children of Ω
6: $m := m - K$
7: SORT$(\Omega_1, \Omega_2, \ldots, \Omega_K)$
8: **for** $k \leftarrow 1, 2, \ldots, K$ **do**
9: $\Psi_k :=$RESTR$(\Omega_k, m + 1)$
10: $n_k :=$ size of Ψ_k
11: $m := m - n_k + 1$
12: **end for**
13: **end if**

Fig. 7. Algorithm RESTR(Ω, M)

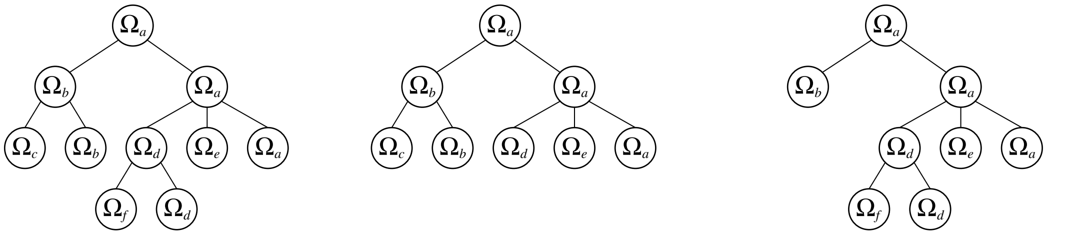
Initially we studied two simple heuristics for constructing a tree, given a maximum size M. The first one, called DIFF, gives priority to subtrees derived from nodes whose schedules differ from their parents. We use a similarity metric, based on the notion of Hamming distance, to determine how similar two schedules are. For instance, while constructing a tree with a size limit $M = 8$ for the system whose optimal tree is the one in Fig. 8(a), we find out that, after the initial schedule Ω_a (the root of the tree), either Ω_b must be followed or the same schedule 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 (which, as opposed to Ω_a, differs from its parent) and the process continues until we obtain the tree shown in Fig. 8(b). 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 $M = 8$ is shown in Fig. 8(c). Experimental data (see Fig. 9) 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 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 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 (in terms of size budget) to those subtrees derived from nodes that actually have higher probability of being followed at run-time.

For evaluation purposes, we generated 100 systems with a fixed 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 the experimental evaluation in this section we considered small graphs (16 tasks) in order to cope with complete trees: note that the complete trees for these systems have, in average, around 10.000 nodes when using LIM$_B$. 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 restricted tree and normalized it with respect to the utility given by the complete tree (non-restricted): $\|U_{restr}\| = U_{restr}/U_{non-restr}$. The plot in Fig. 9 shows that PROB is the algorithm that gives the best results in average. For example, trees limited to 200 nodes (2% of the average size of the complete tree) yield a total utility that is just 3% off from the one produced by the complete tree. Thus, good quality results and short execution times show that the proposed techniques can be applied to larger systems.

7 Cruise Control with Collision Avoidance

Modern vehicles can be equipped with sophisticated electronic aids aiming at assisting the driver, increasing efficiency, and enhancing on-board comfort. One such system is the Cruise Control with Collision Avoidance (CCCA) [6] which



(a) Complete tree (b) Using DIFF (max. size $M = 8$) (c) Using EQ (max. size $M = 8$)

Fig. 8. Trees of schedules

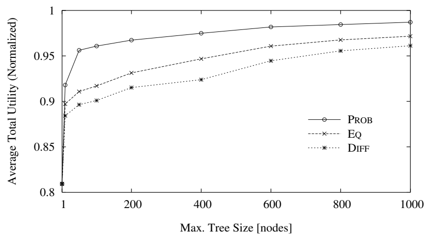


Fig. 9. Evaluation of the tree size restriction algorithms

assists the driver in maintaining the speed and keeping safe distances to other vehicles.

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. 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 considered an architecture with two processors that communicate through a bus. We generated several instances of the task graphs of the four subsystems mentioned above in order to construct a graph with a hyperperiod $\mathsf{T} = 500$ ms. The resulting graph, including processing and communication activities, contains 126 tasks, out of which 6 are soft and 12 are hard. Assuming we can store up to 640 tree nodes, we constructed the tree of schedules using the approaches discussed in Section 6.2 combined with one of the heuristics presented in Section 6.1 (LIM$_B$). Due to the size of the system, it is infeasible to fully construct the complete tree of schedules. Therefore, we compared instead the tree limited to 640 nodes with the static, off-line solution of a single schedule. The results are presented in Table 1. For the CCCA example, we can achieve with our quasi-static approach a gain of above 40% as compared to a single static schedule.

	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

8 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. 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.

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. Due to the intractability of the problem, it is feasible to compute the optimal tree of schedules only for small systems, though. Accordingly, several heuristics that address different complexity dimensions of the problem have been presented. The heuristics for the interval-partitioning step combined with the ones for tree size restriction allow to generate good-quality schedule trees for large applications, even in the context of limited resources. The solutions produced by our heuristics permit important improvements in the quality of results, as compared to a single static schedule, while keeping a very low on-line overhead. This has been demonstrated by a large number of synthetic examples and a real-life application.

References

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

[2] 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.

[3] L. A. Cortés. *Verification and Scheduling Techniques for Real-Time Embedded Systems.* PhD thesis, Dept. of Computer and Information Science, Linköping University, Linköping, Sweden, Mar. 2005.

[4] L. A. Cortés, P. Eles, and Z. Peng. Quasi-Static Scheduling for Real-Time Systems with Hard and Soft Tasks. In *Proc. DATE Conference*, pp. 1176–1181, 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*, pp. 222–231, 1993.

[6] 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*, pp. 1491–1496, 2001.

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

[8] 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.

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

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