Quality and Utility - Towards a Generalization of Deadline and Anytime Scheduling

Thomas Schwarzfischer
(schwarzf@fmi.uni-passau.de)
Chair of Computer Organization (Prof. Dr.-Ing. W. Grass)
Faculty of Mathematics and Informatics, University of Passau
Innstr. 33, 94032 Passau, Germany

Abstract

Scheduling algorithms for real-time systems can be characterized in various ways, one of the most important ones of which is the underlying task model. Many concepts of real-time scheduling relate to either properties local to a single task or constraints imposed onto a task by its context within an application. In terms of real-time applications, task-local properties usually refer to a task-local timeline, whereas context-imposed constraints can frequently be expressed by a clock global throughout the real-time application. This work introduces a paradigm which allows to separate the two aspects of time inherent in an application. Problems under this paradigm are specified in a hierarchical task model. Finally, a suggestion for a dynamic scheduling algorithm based on this specification methodology is made to demonstrate its feasibility.

Keywords: anytime planning and scheduling, planning with hierarchical task networks, dynamic scheduling, scheduling algorithms

Introduction

Specification of processor and resource scheduling problems is always based on a certain paradigm, and scheduling algorithms are being developed to cover a range of problems devised under the assumption of such a paradigm. Two concepts have been widely used in the area of real-time systems research. One of these methodologies is known as deadline scheduling, the other one is called anytime scheduling. Even though these paradigms seem to be very contrary on the first glance, they can nevertheless be unified into one common model.

Problem Statement

The application model used in this work is hierarchical and forms an *and/or* tree structure composed of tasks as nodes and composition relationships as edges. A generalized form of precedence relations defines a second directed graph structure on the same set of nodes. Task arrival times can be specified as periodic (task instances arrive at regular intervals), aperiodic (task instances arrive at irregular intervals), or sporadic (task is instantiated exactly once), and additional jitter values may indicate minor allowed deviations of the actual arrival time from the projected value. Both of the concepts of task execution times and deadlines are generalized applying a value-based approach. Value is accumulated bottom-up using different strategies at the individual nodes, so the goal of a scheduling algorithm is to maximize the value being noted at the root node of the application tree. Due to the presence of nondeterminism caused by possibly irregular and unknown task arrival times, a heuristic dynamic scheduler is proposed aiming to optimally utilize the processors of a homogeneous multiprocessor system.

Structure of the Article

This work starts with an overview of related concepts, followed by an outline of the basic idea for a model covering both deadline and anytime specification. After some definitions, the description of a suitable scheduling algorithm is given. Implementation issues and simulation results conclude the article.

Related Work

This section describes some of the categories of scheduling methodologies which had influence on this work.

Priority and Deadline Scheduling

One well-known category of scheduling algorithms is called priority schedulers, basing their task scheduling on either user-defined or system-determined priorities. Rate-Monotonic Scheduling (RMS) is a priority scheduler for periodic task sets, where the priorities are derived from the tasks' period lengths. Another important subclass of priority schedulers use the individual tasks' deadlines to make the scheduling decisions and are thus generally referred to as deadline schedulers. Some of these algorithms (e.g., Earliest-Deadline-First, EDF) use deadlines as their only source of information on the task set, others (e.g., Least-Slack-Time-First, LST) need additional data on the execution time of tasks etc. (Liu & Layland 1973) presented some fundamental work on the concept of priority and deadline scheduling, which has been investigated into thoroughly and extended in various directions in the meantime (Liu 2000).

Anytime Scheduling

Anytime scheduling algorithms, on the other hand, are based on a completely different view of tasks. Anytime tasks are assigned by the scheduler a certain amount of processor time and respond to rising allocation of cpu time with higher contribution to the reward or performance of the overall application. (Zilberstein 1993) allows a deep insight into the idea of anytime scheduling. Related concepts to anytime scheduling are flexible scheduling (Brandt & Nutt 2002), task-pair scheduling (Streich 1994), imprecise computation scheduling (Shih, Liu, & Chung 1989; Liu *et al.* 1991; Castorino & Ciccarella 2000) or quality-of-service degradable scheduling (Mittal, Manimaram, & Murthy 2000).

Clock-based Scheduling

The approach taken by clock-based scheduling algorithms is to make scheduling decisions according to the comparison of clocks associated with the individual tasks. The primary aim of clock-based scheduling was traffic control in networks (Zhang 1991). However, it has also successfully been applied to task scheduling (Duda & Cheriton 1999). Some clock-based schedulers distinguish explicitly between (usually one) global clock and task-internal clocks. Task-internal time advances only when the corresponding task is allowed to execute on a processor, in contrary to the global clock.

Value-based Scheduling

Value-based scheduling comprises all kinds of scheduling by dint of some kind of time-value functions assigned to the tasks. Value-based solutions have been applied to both (generalized) deadline (Chen & Muhlethaler 1996; Horvitz & Rutledge 1991) and anytime scheduling problems (Zilberstein 1993; Burns *et al.* 2000). However, due to the different notion of time in these two classes of scheduling algorithms, time-value functions are generally interpreted differently. (Cheng 2002) describes a value-based solution for the time-budgeting problem using Lagrange multipliers.

In the deadline-based context, time-value functions can serve as a generalization of deadlines by indicating the utility of a task when completed at a certain point on the global timeline. Figures 1a), 1b), and 1c) show examples of time-value functions describing hard, firm, or soft deadlines, respectively. Note that the arrow in figure 1a) indicates that the value associated with a task having passed its hard deadline is $-\infty$. Apart from the traditional notion of deadlines associated with the tasks of a real-time application, suggestions have been made to further generalize this concept by introducing a so-called critical time. The utility function of a task is such that it contributes to the application only if the task finishes its execution within a small interval of time. Examples can be found in multimedia applications, where an early display of a video frame is considered as bad as a late display. Figure 1d) shows the time-value function of such a task with steeply ascending or descending edges at the borders of the positive-valued interval.

For anytime applications, time-value functions describe the relationship between the cpu time assigned to the task and the resulting contribution to the application's performance. Figure 2a) shows the quality function for a task with fixed execution time, whereas figures 2b) through 2d) show other typical functions describing anytime behaviour of tasks.

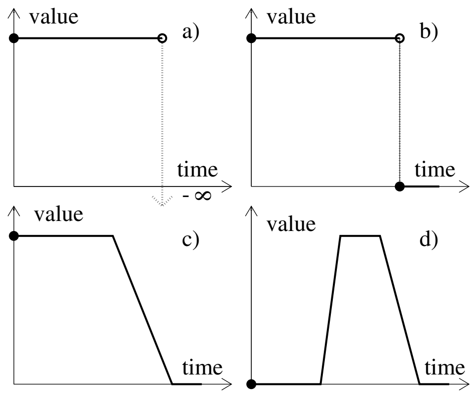

Figure 1: Typical time-value functions describing deadlines

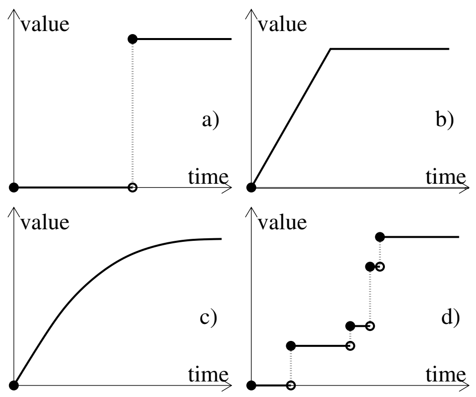

Figure 2: Typical time-value functions describing anytime behaviour

It has to be noted that time-value functions for deadline schedulers are generally defined on a global time domain, whereas time-value functions for anytime schedulers are defined on a task-local time domain. We relate the notion of global and local time to clock-based scheduling. In this respect, global attributes are not affected by the task being allowed to execute on a processor or not. Therefore, deadlines can obviously be called global attributes, even if they are specified relative to the task's release time.

Hierarchical Scheduling

Hierarchical scheduling is an expression which is used for both scheduling of a hierarchically specified application and the use of a hierarchy of schedulers to schedule an application. Hierarchically specified applications require a system of task instantiation rules to prevent inconsistent hierarchy states during the execution (Decker 1996). Scheduler hierarchies need a sophisticated set of guarantees which describe the strucure of legal scheduler hierarchies (Regehr 2001).

Adaptive Scheduling

For many applications, service requirements vary over time, and in many cases arrival of tasks happens unpredictably. Under these circumstances, schedulers must be able to maintain a minimum of service under high load or overload while not wasting resources during times of low load. Adaptive scheduling algorithms take into account these dynamic requirements and try to use information on the current load of the system to parameterize the scheduling algorithm or even apply different algorithms (Hamidzadeh, Atif, & Ramamritham 1999; West 2000). Additionally, some adaptive schedulers try to find an optimal compromise in the effort-quality tradeoff usually experienced in all planning and scheduling systems and especially important in dynamic systems. Dynamic scheduling algorithms compete with the application tasks for shared resources, like, e.g. the processing unit(s). Therefore, it is essential to limit the overhead caused by the scheduler itself (McElhone & Burns 2000). A major advance in adaptive scheduling can be seen in Feedback Control Real-Time Scheduling (Lu 2001), which allows for explicit reasoning about the influence of scheduling on the system to schedule, in contrary to traditional open loop scheduling paradigms.

Unification of Deadline and Anytime Scheduling

The question is whether it is possible to unify the two kinds of time-value functions described in the previous section. The problem arises from the usage of global time on the one hand and local time on the other hand.

This problem can be overcome by binding the local time view of the tasks to the global time domain whenever necessary. Tasks are first bound to the global time domain when they are released and again when they are selected for execution on a processor, so that the internal time advances. In other words, it is necessary to introduce time-value functions which are not functions of only one, but of two time variables. Fortunately, it will not normally be necessary to calculate these functions entirely, so that lazy evaluation mechanisms can be exploited.

The new kind of time-value functions will arise by combining traditional time-value functions for local and global time domains. Note that the following figures are mere projections onto the global time domain. We will show different phases during the "life-time" of a task instance, starting from its release time, ρ, and ending at its deadline, δ. Figure 3a) shows the combination of the time-value function of figure 2d) for an anytime task with the time-value function of figure 1b) representing a firm deadline at the instantiation time of the task, assuming exclusive availability of the processor. The aggregate function used in this case is a simple unweighted pointwise product of the underlying functions. Figure 3b) depicts the situation when Δt (global) time has passed, but no cpu time at all has been allocated to the task. On the contrary, in figure 3c) the task was able to execute during the whole interval $[\rho; \rho + \Delta t[$. Finally, in figure 3d) $2 \cdot \Delta t$ time has passed, and Δt units of cpu time have been allocated to the task. In figures 3c) and 3d) the fact that the overall allocation of cpu time to a task can be no smaller than the allocation already occured in the past is expressed by raising the function's values beyond the deadline to the level perceived at the current time.

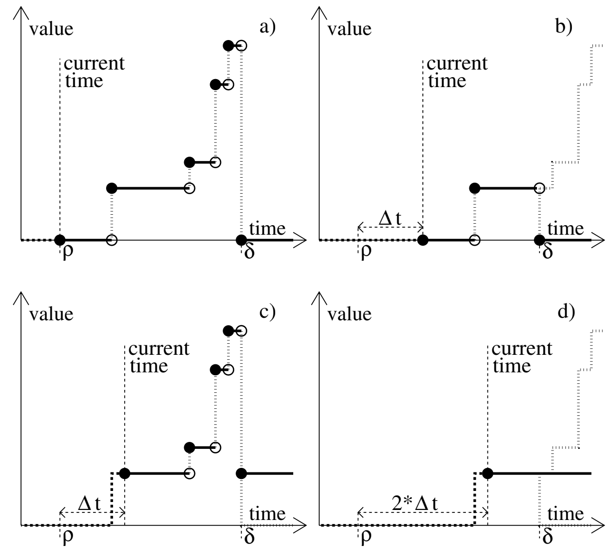

Figure 3: Aggregate function derived from a step anytime quality function and a firm-deadline utility function

Unlike *quality functions* and *utility functions* defined by the application developer, aggregate functions are derived from other time-value functions. We will call these aggregate time-value functions *value functions* throughout the rest of this article.

Definitions

Our application model consists of a set of interruptible tasks $\mathcal{T} = \{T_1, \ldots, T_{|\mathcal{T}|}\}$ arranged in a tree structure and a homogeneous set of processors $\mathcal{P} = \{P_1 \ldots, P_{|\mathcal{P}|}\}$, where $|S| = card(S)$ denotes the cardinality of a set. In order not to further add to the complexity of the model and the scheduling algorithm described later in this work, communication and context switch costs are neglected.

A global allocation of tasks to processors is a function

$$\alpha : \mathcal{T} \times \mathcal{P} \times \mathbb{N}_0 \to \mathbb{B},$$

where $\mathbb{B} = \{true, false\}$ is the set of boolean values.

We derive allocation functions for the individual tasks $T \in \mathcal{T}$ as follows:

$$\alpha_T : \mathbb{N}_0 \to \mathbb{N}_0$$

$$\alpha_T(t) := |\{P \in \mathcal{P} : \alpha(T, P, t) = true\}|$$

As we constrict ourselves to homogeneous architectures, the individual processors need not be distinguished here.

We extend the allocation functions α_T to intervals of time as follows:

$$\alpha_T : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$$

$$\alpha_T[t_1; t_2[:= \begin{cases} \sum_{t=t_1}^{t_2-1} \alpha_T(t) & \text{if } t_2 > t_1 \\ 0 & \text{otherwise} \end{cases}$$

$\alpha_T[t_1; t_2[$ denotes the number of time units task T is assigned in the interval $[t_1; t_2[$.

Note that due to the tree structure of the application, in general more than one processor can be allocated to a task at a certain point in time, depending on its position in the hierarchy. Consequently, for the allocation during an interval, $[t_1; t_2[$, one can state that $0 \leq \alpha_T[t_1; t_2[\leq \max(0, t_2 - t_1) \cdot |\mathcal{P}|$.

We identify a set of allocations with the corresponding global allocation and write $\alpha = \{\alpha_{T_1}, \ldots, \alpha_{T_k}\}$.

Based on a specific allocation, α, one can define the local time function of a task T as follows:

$$\tau_T : \mathbb{N}_0 \times (\mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0) \to \mathbb{N}_0$$

$$\tau_T(t, \alpha_T) := \alpha_T[0; t[\tag{1}$$

We will now introduce an application model suitable for combination of anytime behaviour and deadline specifications. We realized that there exist (amongst possibly others) two basic relationships between tasks, which we wanted to provide for. One of them is the situation where the execution of all the elements of a set of tasks is desireable and these tasks compete for shared resources (in our case the processors only). The other situation is a choice between alternatives, so that the execution of more than one of the tasks does not yield any advantage compared to executing only one. We therefore considered a hierarchical task model to be most suitable for our needs. An application in this model can be represented as a tree structure with three different kinds of nodes: *and*, *or*, and *atomic*. Leaves of the tree have to be so-called *atomic* tasks, which can be thought of as instances of basic algorithms taken from a predefined algorithm library. Each *atomic* task T_{at} is associated a monotonic increasing time discrete quality function

$$q_{T_{at}} : \mathbb{N}_0 \to \mathbb{R}_0^+.$$

And and *or* type nodes reflect the notion of competing tasks or alternative tasks, respectively. All task nodes T have time discrete utility functions

$$u_T : \mathbb{N}_0 \to \mathbb{R}_0^+ \cup \{-\infty\}$$

and a release time, $\rho_T \in \mathbb{N}_0$.

Figure 4 shows an example graph with *and* type (symbol: $\wedge$), *or* type (symbol: $\vee$), and *atomic* type (symbol: $\circ$) tasks. Tasks in this example are annotated with release times, ρ, and (firm) deadlines, δ.

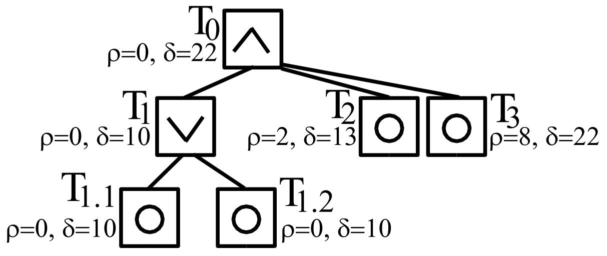

Figure 4: Example application graph

Certain restrictions have to be applied to allocations of processors to tasks according to their logical type and their position within the application tree. An allocation α and its derived allocations α_T are called *conflict-free*, if

- for every atomic task T_{at}, no more than one processor can be allocated at a time:

$$\forall t \in \mathbb{N}_0 : \alpha_{T_{at}}(t) \leq 1 \tag{2}$$

- the allocation of resources to the atomic tasks does not exceed the maximum number of resources available:

$$\forall t \in \mathbb{N}_0 : \sum_{T \in \{T' \in \mathcal{T} : T' \text{ atomic}\}} \alpha_T(t) \leq |\mathcal{P}| \tag{3}$$

- for all child nodes $T_1, \ldots, T_k$ of an *or* type task T_{or}, their allocations are no greater than the parent node's allocation:

$$\forall t \in \mathbb{N}_0 : \forall_{i=1}^{k} : \alpha_{T_i}(t) \leq \alpha_{T_{or}}(t) \tag{4}$$

- for all child nodes $T_1, \ldots, T_k$ of an *and* type task T_{and}, the sum of their allocations is no greater than the parent node's allocation:

$$\forall n \in \mathbb{N}_0 : \sum_{i=1}^{k} \alpha_{T_i}(n) \leq \alpha_{T_{and}}(n) \tag{5}$$

For every allocation α_T to task T with child nodes $T_1, \ldots, T_k$, we denote the subset of $(\mathbb{N}_0 \to \mathbb{N}_0)^k$ of conflict-free allocations for subnodes of T with $A(\alpha_T)$.

Value functions for the inner nodes are calculated as aggregate functions of the value functions of the child nodes, where the method of aggregation depends on the logical type of the node. Value functions for *atomic* tasks are calculated, as described above, as pointwise product of quality and utility function. The value function for *or* type nodes is a simple pointwise maximum function of the child nodes' value functions, whereas the calculation of value functions for *and* type nodes constitutes a complex optimization problem. It is therefore essential for efficiency reasons to apply lazy evaluation in the calculation of value functions.

Instead of defining value functions of two time variables, as outlined above, we chose a notation which includes the global time and the allocation of cpu time to a task. Remember the task-local time can be derived from the task's allocation. Hence, the value function of a task T is a function $v_T : \mathbb{N}_0 \times (\mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0) \rightarrow \mathbb{R}_0 \cup \{-\infty\}$.

First, let T be an *atomic* type task node with quality function q_T and utility function u_T. Given an allocation α_T for the atomic task, its value function is defined as:

$$v_T(t, \alpha_T) := \begin{cases} u_T(t - \rho_T) \cdot q_T(t - \rho_T + \tau_T(t, \alpha_T)) & \\ \qquad\qquad\qquad\qquad \text{if } t \geq \rho_T & \\ 0 \qquad\qquad\qquad\quad \text{otherwise} & \end{cases} \tag{6}$$

Now, let T be an *or* type task node with child task nodes $T_1, \ldots, T_k$ and utility function u_T. Given an allocation α_T for the *or* type task, its value function is defined as:

$$v_T(t, \alpha_T) := \begin{cases} u_T(t - \rho_T) \cdot \max_{i=1}^{k} v_{T_i}(t, \alpha_T) & \\ \qquad\qquad\qquad \text{if } t \geq \rho_T & \\ 0 \qquad\qquad\quad \text{otherwise} & \end{cases} \tag{7}$$

Finally, let T be an *and* type task node with child task nodes $T_1, \ldots, T_k$ and utility function u_T. With $\vec{\alpha} = (\alpha_1, \ldots, \alpha_k)$ being a vector of child node allocations and given an allocation α_T for the *and* type task, its value function is defined as:

$$v_T(t, \alpha_T) := \begin{cases} u_T(t - \rho_T) \cdot \max_{\vec{\alpha} \in A(\alpha_T)} \sum_{i=1}^{k} v_{T_i}(t, \alpha_i) & \\ \qquad\qquad\qquad\qquad \text{if } t \geq \rho_T & \\ 0 \qquad\qquad\qquad\quad \text{otherwise} & \end{cases} \tag{8}$$

Note that vector $\vec{\alpha}$ is taken from the set of conflict-free allocations to subnodes of T and hence is restricted according to equation 5. For example, to calculate the aggregate function v_T for *and* task T with subnodes T_1 and T_2, we impose a constraint $\alpha_{T_1}(t) + \alpha_{T_2}(t) \leq \alpha_T(t)$ at all times t and calculate the sum $v_{T_1}(t, \alpha_{T_1}) + v_{T_2}(t, \alpha_{T_2})$. We can then receive the new value function by finding maxima in the resulting profiles. Figures 5 and 6 show profiles for the cases $t = 100, \alpha_T[0; t[= t$ and $t = 50, \alpha_T[0; t[= t$. It is easy to see from this description that calculating aggregate functions basically means solving the knapsack problem.

Optimizing resource (i.e., processor) allocation to the individual nodes thus means applying the appropriate definition for value functions at all nodes and trying to maximize the value obtained at the root node. To start the calculation of value functions, one has to set an allocation for the root node. We assume the application defined by an application graph has exclusive access to the resources. Hence, the allocation used for the root node, T_0, is

$$\forall P \in \mathcal{P} : \forall t \in \mathbb{N}_0 : \alpha(T_0, P, t) = true$$

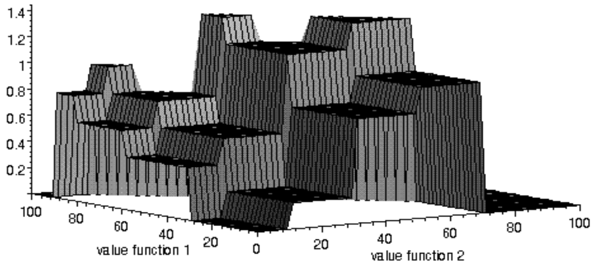

Figure 5: Profile for $t = 100, \alpha_T[0; t[= t$

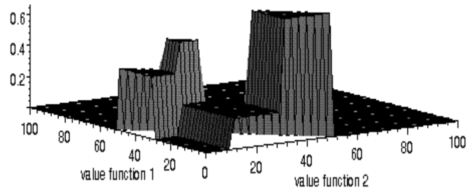

Figure 6: Profile for $t = 50, \alpha_T[0; t[= t$

or

$$\forall t \in \mathbb{N}_0 : \alpha_{T_0}(t) = |\mathcal{P}|$$

Note that the simplified description in this section is valid only for the situation that none of the tasks affected have been allocated any units of cpu time prior to the time of running the scheduling algorithm. A dynamic scheduler, however, is called multiple times during the run-time of the application and cannot generally rely on this assumption. Furthermore, the model does so far not take into account precedence relations; we will outline the idea of how to integrate this notion of dependencies between tasks into our model briefly in the following section.

Scheduling Algorithm

We now present a feasible scheduling algorithm for a simplified version of the general problem described in the previous sections. Here, we choose utility functions representing firm relative deadlines only, i.e., for every task T, its utility function u_T is defined as follows:

$$u_T(t) = \begin{cases} 1 & \text{if } t < \delta_T \\ 0 & \text{if } t \geq \delta_T \end{cases}$$

δ_T is called the deadline of T.

We propose a dynamic scheduling scheme which allows for partial calculation of schedules. Given a lookahead parameter, the algorithm aims at optimizing the resource allocation for the interval defined by the current time and the lookahead. Obviously the quality of the optimization depends largely on the setting of this parameter.

Calculation of Active Intervals

In a preparatory step the scheduling algorithm represents the problem in a way suitable for an optimization algorithm. We note that we need not generally take into account every single point of time within the scheduling interval, as the tasks are by definition interruptible at any time without cost. It is easy to understand that the allocation of processors to tasks depends only on the allocation within certain intervals, whereas the allocation at exact points of time is irrelevant. Hence, we define the set of active intervals I_T for all tasks $T \in \mathcal{T}$ as follows:

- For an *atomic* type task T_{atomic}, the only active interval starts at the release time of the task and ends at its deadline:
$$I_{T_{atomic}} := \{[\rho_T; \delta_T[\}$$
- For an *or* or *and* type task $T_{and/or}$ with child tasks $T_1, \ldots, T_k$, the set of active intervals is derived from the child tasks:
$$\begin{aligned} I_{T_{and/or}} := \{ & [t_s; t_e[: t_s < t_e \\ \wedge \quad & (\exists i, j \in \{1, \ldots, k\} : \\ & (\rho_{T_i} = t_s \vee \delta_{T_i} = t_s) \\ & \wedge(\rho_{T_j} = t_e \vee \delta_{T_j} = t_e)) \\ \wedge \quad & (\nexists l \in \{1, \ldots, k\} : \\ & t_s < \rho_{T_l} < t_e \\ & \vee t_s < \delta_{T_l} < t_e) \\ \wedge \quad & (\exists m \in \{1, \ldots, k\} : \\ & \rho_{T_m} \leq t_s \\ & \wedge \delta_{T_m} \geq t_e)\} \end{aligned}$$

Figure 7 explains the calculation of active intervals for the application of figure 4 in bottom-up manner.

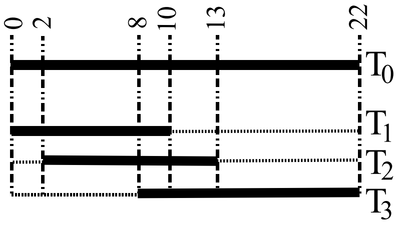

Figure 7: Calculation of active intervals for example graph

Table 1 gives the set of active intervals for all nodes in the graph.

node	active intervals
T_0	$[0;2[$, $[2;8[$, $[8;10[$, $[10;13[$, $[13;22[$
T_1	$[0;10[$
T_2	$[2;13[$
T_3	$[8;22[$
$T_{1.1}$	$[0;10[$
$T_{1.2}$	$[0;10[$

Table 1: Active intervals for nodes of example graph

Table 2 shows the tasks active in a specific interval. Atomic tasks are written bold.

We can then use allocations on active intervals $I = [t_s; t_e[$ instead of individual points of time.

Primary Allocation

Having established the set of active intervals for all tasks in bottom-up manner, the optimization algorithm starts off

interval	active tasks
$[0;2[$	T_0, T_1, $\mathbf{T_{1.1}}$, $\mathbf{T_{1.2}}$
$[2;8[$	T_0, T_1, $\mathbf{T_{1.1}}$, $\mathbf{T_{1.2}}$, $\mathbf{T_2}$
$[8;10[$	T_0, T_1, $\mathbf{T_{1.1}}$, $\mathbf{T_{1.2}}$, $\mathbf{T_2}$, $\mathbf{T_3}$
$[10;13[$	T_0, $\mathbf{T_2}$, $\mathbf{T_3}$
$[13;22[$	T_0, $\mathbf{T_3}$

Table 2: Active nodes for all intervals

with an allocation for the root node. The basic scheme for the distribution of cpu time is derived from the intervals previously calculated, as can be seen in figure 8. Once again, we make the assumption of full allocation of resources to the root node. Figure 9 shows a distribution of cpu time as it might be used at the beginning of the optimization process. To root node T_0, the allocation of cpu cycles in an interval $[t_s; t_e[$ is $t_e - t_s$ units. Allocations at *or* nodes are passed on to the children to full extent, whereas allocations at *and* nodes are distributed approximately uniformly amongst the child nodes.

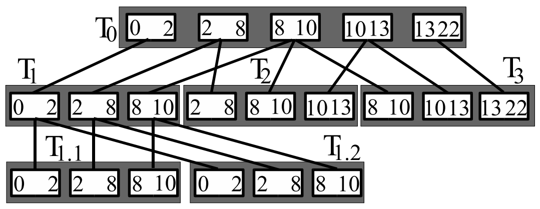

Figure 8: Basic scheme for distribution of cpu time

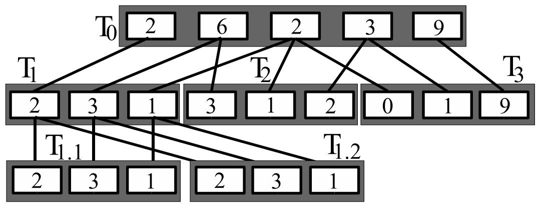

Figure 9: Primary distribution of cpu time

Optimization

According to the logical type of the task nodes, allocations for the child nodes are calculated as follows: The optimization code for every type of node consists basically of two functions, *optimize()* and *evaluate()*. Figures 10 and 11 show the pseudocode for the functions used for *and* nodes, figures 12 and 13 the pseudocode for the functions used for *or* nodes.

Assume the step quality functions for the *atomic* nodes as given in table 3, so that $q(t) = \max_{0 \leq t' \leq t} q(t')$.

The allocation of figure 9 would then yield an overall value of 1.3, the optimized allocation of figure 14 an overall value of 1.9. The Gantt chart for the resulting schedule is shown in figure 15.

```
$optimize_{and}(\alpha_{and})$
  forall intervals $i$ do
    $c :=$ #active children in interval $i$
    $a := \alpha_{and}(i)$
    for $n = 1$ to $c$ do
      $\alpha_{child(n)} :=$
        $\lfloor \frac{a}{c} \rfloor + 1 - \min(1, \left\lfloor \frac{n-1}{\max(1, a \mod c)} \right\rfloor)$
    od
  od
  $Temp := Temp_{start}$
  $V := evaluate(\alpha_{child(1)}, \ldots, \alpha_{child(k)})$
  while($Temp > Temp_{min}$) do
    for $n = 1$ to $\#repetitions$ do
      $(\alpha'_{child(1)}, \ldots, \alpha'_{child(k)}) :=$
          $searchStep(\alpha_{child(1)}, \ldots, \alpha_{child(k)})$
      optimize and evaluate children
      $V' := evaluate(\alpha_{child(1)}, \ldots, \alpha_{child(k)})$
      if($V' > V$) then
        $(\alpha_{child(1)}, \ldots, \alpha_{child(k)}) :=$
            $(\alpha'_{child(1)}, \ldots, \alpha'_{child(k)})$
        $V := V'$
      else
        with probability $\min(1, e^{\frac{V-V'}{Temp}})$ do
          $(\alpha_{child(1)}, \ldots, \alpha_{child(k)}) :=$
              $(\alpha'_{child(1)}, \ldots, \alpha'_{child(k)})$
          $V := V'$
        od
      fi
      $Temp := cooldownFactor \cdot Temp$
    od
  od
end
```

Figure 10: *optimize()* function for *and* node

```
$evaluate_{and}()$
  return $\sum_{\text{children c}} value(c)$
end
```

Figure 11: *evaluate()* function for *and* node

```
$optimize_{or}(\alpha_{or})$
  forall children $c$ do
    $\alpha_c := \alpha_{or}$
  od
  optimize and evaluate children
end
```

Figure 12: *optimize()* function for *or* node

```
$evaluate_{or}$
  return $\max_{\text{children c}} value(c)$
end
```

Figure 13: *evaluate()* function for *or* node

Node	Step 1		Step 2		Step 3		Step 4	
	t	q	t	q	t	q	t	q
$T_{1.1}$	0	0.0	4	0.2	6	0.6		
$T_{1.2}$	0	0.0	2	0.4	8	1.0		
T_2	0	0.0	2	0.1	4	0.2	6	0.3
T_3	0	0.0	4	0.3	8	0.4	12	0.8

Table 3: Assumed quality functions of atomic tasks

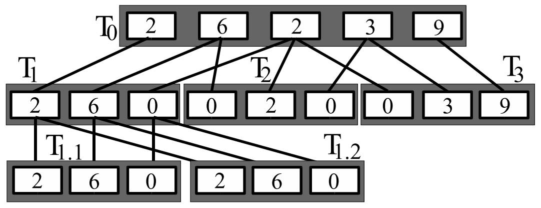

Figure 14: Final distribution of cpu time

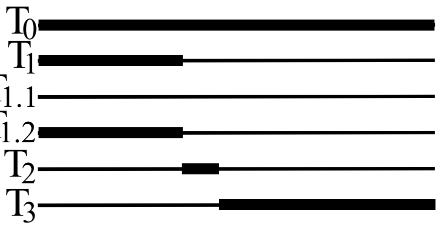

Figure 15: Gantt chart of result schedule

Value dependencies

Precedence relations between tasks describe which task has to be executed prior to another one due to application-specific constraints like, e.g., dataflow conditions.

In the context of value-based scheduling, there is no intrinsic equivalent to the completion of execution in traditional task models. One can, however, emulate this property in our model by using quality functions with two values only, e.g., $q(t) = \begin{cases} 0 & \text{if } t < t_0 \\ 1 & \text{if } t \geq t_0 \end{cases}$.

Precedence relations between tasks can be stated as weighted directed edges, where the interpretation of an edge is as follows: A weight of 0 indicates that the origin of the edge is no precondition for the target node at all; the edge is interpreted as non-existent. A weight of 1 means that the origin of the edge has full impact on the value of the target node. We call this kind of relationship *value dependency*.

Our model assumes that all tasks can be executed independently of each other. Value dependencies affect only the calculation of aggregate value functions and hence the overall value achieved by the application. However, traditional precedence relations can be imitated as follows. Imagine task T_1 has to be executed prior to T_2 because of a dataflow dependency. Our scheduler would then assign a lower value to the pair of tasks when executed the other way around; hence, the optimization algorithm would effectively avoid this situation.

Value dependencies form a graph structure in addition to the hierarchy graph. The algorithm fragments shown above can be used basically unchanged if the dependency graph is acyclic; a modification allowing graphs to be cyclic would require a more sophisticated concept of task instances, which has not yet been integrated into this model. The tasks $\mathcal{T}$ together with the (value) dependency edges, DE, of an application form the (value) dependency graph, $DG = (\mathcal{T}, DE)$.

Of the many possible interpretations of value dependencies, we chose the following, for which we extend the definition of the value function for node T in the interval $I_i \in I = \{I_1, \ldots, I_{|I|}\}$:

$$v'_T(I_i, \alpha_T) = \begin{cases} 0 & \text{if } \forall_{j=1}^{i} \alpha_T(I_j) = 0 \\ \xi \cdot v_T(I_i, \alpha_T) & \text{otherwise} \end{cases} \tag{9}$$

where the *impact factor* ξ is

$$\xi = \prod_{T' \in pred(T)} (v'_{T'}(I_\sigma))^{weight(T',T)}, \tag{10}$$

the *start interval index* σ is

$$\sigma = \min_{j=1}^{l} \{j : \alpha_T(I_j) > 0\}, \tag{11}$$

and the set of predecessor nodes $pred(T)$ of T is

$$pred(T) = \{T' \in \mathcal{T} : \exists (T, T') \in DE : weight(T, T') > 0\}. \tag{12}$$

This interpretation assumes a value flow to happen via the edges of the value dependency graph when a task starts executing on a processor for the first time (in the interval I_σ). Later rise in value at the predecessor nodes does not affect the successor's value any more. Under this assumption, a scalar, ξ (the impact factor), is sufficient to fully describe the predecessors' influence on a task. Note that this way it is possible that a node passes on different values to its successor nodes, which may very well make sense due to different timing constraints and edge weights.

Implementation issues

This section presents some important notes connected to the implementation of the scheduling algorithm.

Caching

To avoid multiple optimizations for the same allocations at a node, it is essential to maintain a cache of previously calculated solutions for each node. The cache can be used to find an exact match (in which case no new optimization has to be performed) or to find a solution for a similar allocation to be used as a hopefully good starting point for the search.

Deterministic clusters

As stated before, our model comprises both predictable and unpredictable specifications of arrival times for tasks. It is typically possible to extract determinstic clusters (i.e., subtrees) within the application graph. For these clusters, it is possible to maintain separate caches, which can be consulted whenever the root node of these clusters is instantiated.

Parameters of Simulated Annealing Algorithm

The optimization for *and* nodes is being done by simulated annealing, and the structure of the application graph helps direct the search. To keep the optimization algorithm scalable and useful for dynamic scheduling, it is important to distribute the limited time allowed for running the scheduling algorithm among the individual task nodes. A helpful parameter can be the size of the (local) search space. Consider a node with active intervals $I = \{i_1, \ldots, i_{|I|}\}$, an allocation of $\alpha(i)$ for every interval I_i and c_i children of the node sharing with the parent node interval I_i as active interval. For example, in figure 8 node T_0 has an active interval $[10; 13[$, which it shares with two of its child nodes, T_2 and T_3. It is easy to see that the search space of the node, given allocation α, has size:

$$searchSpaceSize = \prod_{i \in I} \frac{(\alpha(i) + c_i - 1)!}{(\alpha(i))! \cdot (c_i - 1)!}$$

Assume we want to scale the scheduling effort at each node according to the size of the local search space and a parameter pc (percentage of the search space size). Supposing further the number of temperature levels should be the same as the number of repetitions within a temperature level, we can calculate some of the parameters of the simulated annealing algorithm as follows:

- number of repetitions within temperature level:

$$\#repetitions := \left\lceil \sqrt{searchSpaceSize \cdot pc} \right\rceil$$

- cooldown factor

$$cooldownFactor = \left(\frac{Temp_{min}}{Temp_{start}} \right)^{\#repetitions^{-1}}$$

Partial Calculation of Value Functions

In the previous definitions allocation functions are defined over discrete points of time taken from $\mathbb{N}_0$. Calculating value functions for an infinite number of times is obviously intractable. We calculate value functions and allocations iteratively for a limited interval of time (lookahead) and thus receive increasing prefixes of the schedule. On the other hand, it appears obvious that scheduling for too small a period of time degrades the performance of a scheduling algorithm. For periodic task sets, it is known that scheduling has to be done in the interval between 0 and the least common multiple (lcm) of the task periods. For sporadic task sets, the entire scheduling time domain is finite. However, in the presence of nondeterminism, arising from aperiodic tasks or release time jitter, recalculation of schedules will frequently be necessary. The proper choice of the scheduling lookahead is one of the main parameters to adapt the algorithm to a specific application.

Simulation

The algorithm described has been implemented in Pascha, an integrated specification and simulation environment for scheduling problems. Figure 16 shows an application graph specified using the enviroment's editor, figure 17 the result of a simulation run.

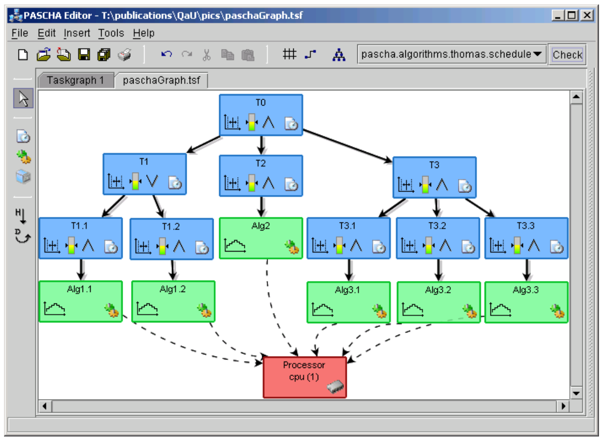
PASCHA Editor - T:\publications\QaU\pics\paschaGraph.tsf
File Edit Insert Tools Help
pascha.algorithms.thomas.schedule
Check
Taskgraph 1
paschaGraph.tsf
T0
T1
T2
T3
T1.1
T1.2
Alg2
T3.1
T3.2
T3.3
Alg1.1
Alg1.2
Alg3.1
Alg3.2
Alg3.3
Processor
cpu (1)

Figure 16: Application graph in the Pascha editor

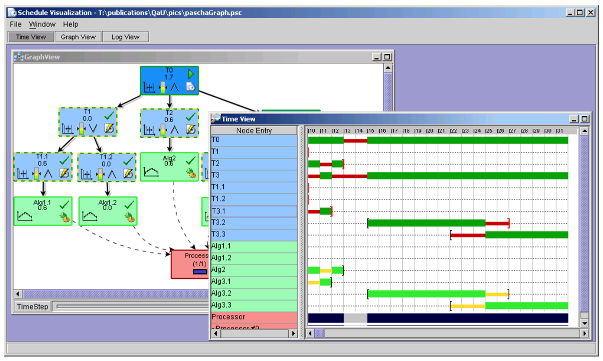

Figure 17: Pascha simulation run

Results

Simulation runs have been performed for generic loads composed of typical mixed sets of periodic and sporadic tasks (in the range of 100 task instances), for a simulation space of 500 units of time, and various task sizes from 5 to 150 time units. Apart from running the example application graphs with the simulated annealing algorithm described above, the optimal distribution of cpu time was calculated, and for further comparison a value-based version of EDF (using the maximum level of value functions to determine the worst-case execution times of the corresponding tasks)

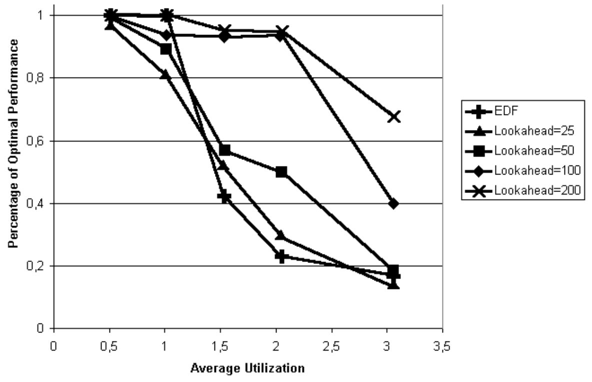

Figure 18: Performance profiles of simulated annealing and EDF scheduling algorithms

was applied. The results derived from these simulations can be seen in figure 18.

As expected, EDF (which is known to be optimal if the system is not in overload) outperforms our algorithm disregarding any parameter settings up to a utilization of 1.0 and degrades rapidly beyond this value. Furthermore, a higher lookahead for our algorithm is obviously rewarded with better overall results, but has, of course to be paid for by higher costs of scheduling.

Figure 19 shows that the performance profiles of the algorithm for different loads can themselves be interpreted as monotonic increasing time-value functions; compared to the rather complex optimization algorithm, the scheduling overhead for EDF is negligible and not shown in the diagram.

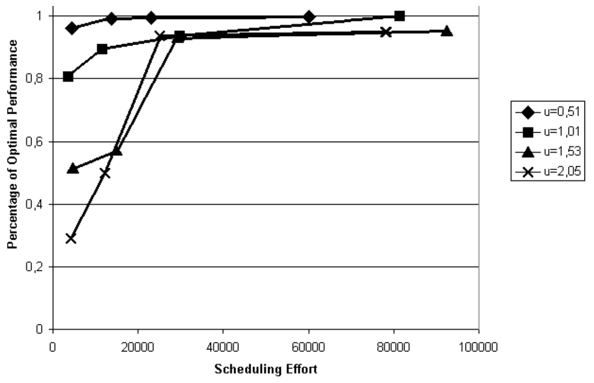

Figure 19: Time-value functions for simulated annealing algorithm

Due to this fact, the scheduler can be integrated into the application model by introducing a new task node, T_S, representing the scheduling algorithm, a new common parent node, T_0', for scheduler and application root node, and a dependency edge between them, as outlined in figure 20.

Apart from the utilization and the lookahead, further important parameters influencing the performance of the algorithm are the diversity of the task set and the degree of unpredictability. The optimization problem is much easier if the task periods are harmonic (i.e., they are multiples

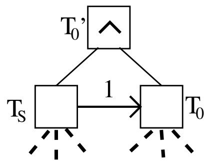

Figure 20: Extended application graph

of each other) or in the same order of magnitude. Obviously, nondeterminism arising from release time jitter or aperiodic task arrival times makes a proper estimation of the near future behaviour of the system more difficult and may necessitate recalculations or adaptions to the precalculated partial schedule.

Conclusion

In this work we presented a paradigm for the common specification of local properties (quality) and context properties (deadlines, utility) of tasks in a real-time application alongside with a suggestion for the implementation of an algorithm based on this paradigm. We also showed the quality of the resulting schedules and the applicability to dynamic schedulers. Our future work will include evaluation of other optimization methods for the formulated problem, extension of the task model to explicitly handle instances and to make cyclic dependency graphs possible, and investigating into the effects of different sources of nondeterminism on the performance of the algorithm. Another interesting topic will be the relaxation of the assumption of very simple utility functions representing deadlines only. It appears to be reasonable to assume that more general utility functions do not have much effect on the performance of global, but most likely on the performance of local search algorithms, as the topology of the search space may tend to make the search much more difficult. For the profiles in figures 5 and 6, e.g., it would no longer suffice to look for maxima at the edges ($\alpha_{T_1}(t) + \alpha_{T_2}(t) = \alpha_T(t)$), as is the case for the simplified problem. Furthermore, the comfortable concept of active intervals is not applicable in the case of generalized utility functions.

References

Brandt, S., and Nutt, G. 2002. Flexible soft real-time processing in middleware. *Real-Time Systems* 22:77–118.

Burns, A.; Prasad, D.; Bondavalli, A.; Giandomenico, F. D.; Ramamritham, K.; Stankovic, J.; and Strigni, L. 2000. The meaning and role of value in scheduling flexible real-time systems. *Journal of Systems Architecture* 46:305–325.

Castorino, A., and Ciccarella, G. 2000. Algorithms for real-time scheduling of error-cumulative tasks based on the imprecise computation approach. *Journal of Systems Architecture* 46:587–600.

Chen, K., and Muhlethaler, P. 1996. A scheduling algorithm for tasks described by time value function. *Real-Time Systems* 10:293–312.

Cheng, A. M. 2002. *Real-Time Systems*. John Wiley & Sons.

Decker, K. 1996. TAEMS: A Framework for Environment Centered Analysis & Design of Coordination Mechanisms. In *Foundations of Distributed Artificial Intelligence, Chapter 16*, 429–448. G. O'Hare and N. Jennings (eds.), Wiley Inter-Science.

Duda, K., and Cheriton, D. 1999. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*.

Hamidzadeh, B.; Atif, Y.; and Ramamritham, K. 1999. To schedule or to execute: Decision support and performance implications. *The International Journal of Time-Critical Computing Systems* 16:281–313.

Horvitz, and Rutledge. 1991. Time-dependent utility and action under uncertainty. In *Proceedings of Seventh Conference on Uncertainty in Artificial Intelligence*, 151–158.

Liu, C., and Layland, J. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM* 30:46–61.

Liu, J.; Lin, K.; Shih, W.; Yu, A.; Chung, J. Y.; and Zhao, W. 1991. Algorithms for imprecise computations. *IEEE Computer* 24:58–68.

Liu, J. 2000. *Real-Time Systems*. Prentice-Hall.

Lu, C. 2001. *Feedback Control Real-Time Scheduling*. Ph.D. Dissertation, University of Virginia.

McElhone, C., and Burns, A. 2000. Scheduling optional computations for adaptive real-time systems. *Journal of Systems Architecture* 46:49–77.

Mittal, A.; Manimaram, G.; and Murthy, C. S. R. 2000. Integrated dynamic scheduling of hard and QoS degradable real-time tasks in multiprocessor systems. *Journal of Systems Architecture* 46:793–807.

Regehr, J. 2001. *Using Hierarchical Scheduling to Support Soft Real-Time Applications in General-Purpose Operating Systems*. Ph.D. Dissertation, University of Virginia.

Shih, W.-K.; Liu, W.; and Chung, J. 1989. Fast algorithms for scheduling imprecise computations. In *Proceedings of the Real-Time Systems Symposium*, 12–19.

Streich, H. 1994. Task Pair-Scheduling: An approach for dynamic real-time systems. In *Proceedings of the 2nd Workshop on Parallel and Distributed Real-Time Systems*.

West, R. 2000. *Adaptive Real-Time Management of Communication and Computation Resources*. Ph.D. Dissertation, Georgia Institute of Technology.

Zhang, L. 1991. Virtual clock: A new traffic control algorithm for packet-switched networks. *ACM Transactions on Computer Systems* 9(2):p101–124.

Zilberstein, S. 1993. *Operational Rationality through Compilation of Anytime Algorithms*. Ph.D. Dissertation, University of California at Berkeley.