History-Cognisant Time-Utility-Functions for Scheduling Overloaded Real-Time Control Systems

Florian Kluge, Florian Haas, Mike Gerdes, Theo Ungerer
Department of Computer Science, University of Augsburg, Germany
{kluge,haas,gerdes,ungerer}@informatik.uni-augsburg.de

Abstract—Time-utility functions (TUFs) as means for scheduling allow to build flexible real-time systems. TUFs yield utility values for single job executions. We extend this concept to history-cognisant utility functions (HCUFs) that assign an overall utility value to a whole task. We aim to use such HCUFs to improve the single-task performance in control systems. We present two extensions of the EDF policy for overloaded firm real-time systems and compare them with related approaches. First promising results are shown.

I. INTRODUCTION

The notion of deadlines as the sole scheduling criterion is often too strict and can lead to inflexible and performance-sacrificing real-time systems. Instead, Jensen et al. [10] propose to use time value functions (TVFs, also Time Utility Functions, TUFs) as a basis for scheduling. A TUF represents a task’s value that it will contribute to the system if it is completed up to a certain time. A TUF can be based on task deadlines, but may also incorporate other parameters, e.g. the significance a task has for the operation of a system.

Jensen et al.’s work [10] was extended for best effort scheduling [14] and tasks with dependent activities [8]. In the meantime, several heuristic scheduling algorithms have been proposed (e.g. [7], [12], [20]). The notion of time-value is used in scheduling of real-time systems in general (see [19] for an overview, or e.g. [1], [6], [9]) and in the special case of overloaded real-time systems (e.g. [4], [11], [15], [16]). Applications of time-value scheduling can be found in dynamic reconfiguration of systems [5], Ethernet packet scheduling [20] and robotics [2].

To the best of our knowledge, hitherto existing works on TUF-based scheduling only take the possible value of the current task instance into account, but do not care about a task’s previous execution behaviour. In our work, we target periodic control loops that can tolerate sporadic deadline misses. If a control algorithm is built robust and executed at a rate high enough, the system can tolerate single iterations to fail, as it is able to recompense for these in the following iterations [17]. Such behaviour has been formalised e.g. in the \<(m, k)\>-firm real-time task model [18] and the weakly-hard real-time model [3]. Nevertheless, control loops are most often implemented as periodic tasks with hard deadlines, thus sacrificing performance and flexibility.

Our aim is to exploit the robustness of such control loops and execute them in a more flexible manner. Scheduling decisions influence the behaviour of the control system and can lead to a degradation of its quality. In this paper, we present an approach to distribute such degradations equally over all control tasks in the system at hand. Therefore, we introduce history-cognisant utility functions (HCUFs). A HCUF maps a task’s execution history into a single value which can be evaluated by scheduler. We demonstrate how these HCUFs can be used for scheduling overloaded control systems. If an overload situation is detected, we cancel single jobs until all other jobs can meet their deadlines. Using HCUFs as base for the decision which jobs shall be cancelled, we achieve that (1) cancellations are distributed equally over all tasks in the system, and (2) the number of subsequent cancellations affecting one task is reduced. While our approach is currently based on timing parameters solely, it is possible to refine the utility functions for an actual implementation using further metrics that might also allow for an interaction between the control algorithm and the scheduler.

This paper is structured as follows: In section II we review the concept of TUFs and introduce history-cognisant utility functions. A scheduling approach based on HCUFs is presented in section III. Preliminary evaluation results are shown in section IV. We conclude this paper in section V.

II. UTILITY FUNCTIONS

Utility functions are applied to jobs that are generated by tasks. A task \(\tau_i\) generates jobs \(j_{i,k}\) at times \(a_{i,k}\). Each job has a deadline \(d_i\) relative to its activation time \(a_{i,k}\) and an absolute deadline \(\hat{d}_{i,k} = a_{i,k} + d_i\). The completion time of job \(j_{i,k}\) is denoted as \(c_{i,k}\). All utility functions map into the utility domain \(U := [0,1] \cup \{-\infty\}\). \(1\) represents maximum benefit for the system that can degrade down to 0 meaning no benefit, \(-\infty\) stands for a failed job with possibly catastrophic consequences.

A. Traditional Utility Functions

Time-utility functions can be used to assess single job executions. The concept of firm real-time jobs that are worthless once they exceed their deadline can be represented by the following utility function which is also depicted in figure 1(a):

\[
u_F(j_{i,k}) = \begin{cases} 1 & \text{ if } c_{i,k} \leq \hat{d}_{i,k} \\ 0 & \text{ else} \end{cases}
\] (1)

A utility function for soft real-time jobs which are allowed to miss their deadlines can be defined exemplarily in the
following manner (see also figure 1(b)):

\[ u_S(j_i,k) = \begin{cases} 1 & \text{if } c_i,k \leq d_i,k \\
1 - \frac{c_i,k - d_i,k}{d_i,k} & \text{if } c_i,k > d_i,k \leq d_i,k + d_i \\0 & \text{else} \end{cases} \]

(2)

\[ \text{Example 1: Let } U_E(j_i,k) \text{ be the utility of task } \tau_i \text{ after the execution of the } k\text{-th job } j_i,k. \text{ Using some weight } w \in (0,1) \text{ and a TUF } u(j_i,k), \text{ we define } U_E(j_i,k) \text{ as:} \]

\[ U_E(j_i,0) = 1 \]

\[ U_E(j_i,k) = (1-w)U_i(j_i,k-1) + wu(j_i,k) \]

(6)

In our future work we will investigate the influence of the parameter \( w \) on an application. Also, we plan to define further HCUFs that e.g. take into account how often a task deviates from a desired behaviour succeedingly.

III. UTILITY-BASED SCHEDULING

Jensen et al. [10] and Locke [14] proposed an extension to the earliest deadline first (EDF) scheduling algorithm [13] to handle possible overload situations. As long as no overload occurs, their best-effort algorithm (in the following called BE) implements the original EDF online policy. If high probability for an overload is detected during runtime, the algorithm modifies an EDF schedule based on the value density of each task. The value density for a task \( \tau_i \) is calculated as \( \bar{V}_i/C_i \), with \( \bar{V}_i \) being the value of the task, and \( C_i \) its processing time. The algorithm removes tasks with lowest value density from the schedule until the overload probability drops below a predefined threshold.

Aldarmi and Burns [1] address one problem of this approach stemming from the static value density (SVD) used in BE: in choosing a task to remove from a schedule, BE does not care whether the task has already started executing. If a running task is chosen for removal, the work it has performed until that point will be lost, and the computing time used up was therefore wasted. Aldarmi and Burns propose to use dynamic value density (DVD) as base for scheduling decisions. They calculate a task \( \tau_i \)’s priority \( P_i(t) \) at some time \( t \) as \( P_i(t) = V_i(t)/C_i(t) \) with \( C_i(t) \) being the task’s remaining execution time. Thus, once a task has started executing, its priority will increase and probability of cancellation decreases.

We notice another problem that can occur with the use of BE: Jobs of tasks with a low value density are cancelled with a higher probability during overload situations. Concerning control applications, such behaviour can be counterproductive: complex control loops with a high execution time might be cancelled more often. In this case, it will be helpful if all tasks suffer a similar degradation, but still are able to provide some guaranteed value to the system.

To address this problem, we propose to use one HCUFs as base for priority calculation to make the scheduling process history-cognisant. Our approach is based on the fact that robust control loops can tolerate single executions to fail. However, such failures must not occur too often. If the scheduler has to perform job cancellations to make a schedule valid, we want to prefer such tasks for cancellation that performed successfully before. Tasks that suffered from cancellations before shall be preferred for execution.

We modify the BE scheduling approach by using different criteria for removal of jobs from an overloaded schedule. Instead of removing low-value-density jobs, we choose jobs
Table I

<table>
<thead>
<tr>
<th>Task</th>
<th>( P_i/0_i )</th>
<th>( C_i )</th>
<th>( d_i )</th>
<th>( p_A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>6/0</td>
<td>2</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>p2</td>
<td>6/2</td>
<td>2</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>p3</td>
<td>6/4</td>
<td>2</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>s1</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>0.35</td>
</tr>
</tbody>
</table>

Table II

<table>
<thead>
<tr>
<th>Task</th>
<th>( P_i )</th>
<th>( C_i )</th>
<th>( d_i )</th>
<th>( p_A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>15</td>
<td>3</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>p2</td>
<td>57</td>
<td>8</td>
<td>13</td>
<td>-</td>
</tr>
<tr>
<td>p3</td>
<td>111</td>
<td>9</td>
<td>19</td>
<td>-</td>
</tr>
<tr>
<td>p4</td>
<td>42</td>
<td>10</td>
<td>18</td>
<td>-</td>
</tr>
<tr>
<td>p5</td>
<td>37</td>
<td>5</td>
<td>19</td>
<td>-</td>
</tr>
<tr>
<td>s1</td>
<td>50</td>
<td>5</td>
<td>10</td>
<td>0.4</td>
</tr>
<tr>
<td>s2</td>
<td>10</td>
<td>2</td>
<td>5</td>
<td>0.2</td>
</tr>
<tr>
<td>s3</td>
<td>100</td>
<td>13</td>
<td>29</td>
<td>0.3</td>
</tr>
</tbody>
</table>

whose tasks have accumulated a maximum utility \( U \) until now. This leads to our first history-cognisant EDF extension, which we call HC1 in the following. Additionally, the tasks shall also profit from the findings concerning DVD. Therefore, we also include a job’s remaining execution time \( \overline{C}(t) \) in our removal metric. This leads to a removal of jobs that have maximum values of \( U(t) \cdot \overline{C}(t) \). We call this second history-cognisant EDF extension HC2.

IV. EVALUATION

We have performed preliminary use case evaluations of the HCUFs to compare the four extensions BE, DVD, HC1 and HC2 of the EDF scheduling policy. Any extension becomes active, if an overload situation is detected and removes tasks from the schedule. We are interested, how the different scheduling approaches influence the overall completion rates of the single tasks. Also, we investigate how cancellations are distributed over a single task’s life time.

A. Scenario

All scenarios that we investigated so far are based on the following assumptions: time is divided into discrete time steps. If a task \( \tau_i \) is activated, it generates a job \( j_{i,k} \), \( k = 0, 1, \ldots \) that must be executed. All tasks \( \tau_i \) are firm real-time with deadline \( d_i \) relative to their activation time. Their utility is calculated according to equation (1). Periodic tasks \( p_i \) are activated at the beginning of their period \( P_i \), heeding a possibly positive offset \( o_i \). Periodic tasks may have deadlines equal to their period, but can also have shorter ones. Sporadic tasks \( s_i \) may be activated with a certain probability \( p_A \) in each time step after their minimum separation time has elapsed. Any task \( \tau_i \) accumulates utility through a history-cognisant utility function \( U_i \) as defined in equation (6). In our simulations we chose \( w = 0.5 \).

So far, we have performed evaluations with several task sets. For reasons of space, we only present the results of two task sets. Task set 1 (table I) is rather well-natured. The periodic tasks create 100% processor utilisation. Any time the sporadic task is activated, the scheduler has to cope with an overload situation. We chose this task set because the overload is well under control and easy comprehensible. Task set 2 (table II) is ill-conditioned. The periodic tasks alone often lead to overload situations that can be aggravated by additional sporadic tasks. The aim of this task set is to estimate, how the different scheduling approaches perform under very hard conditions. For any sporadic task, the number recorded in the \( P_i \) column indicates the task’s minimum separation time between to subsequent activations.

In our simulations, jobs are generated and executed for one million time steps. After this time no new jobs are generated, and only those already activated are finished.

B. Completion Rates

In task set 1, the periodic tasks generate a basic load of 100%. The sporadic task (s1) gains a completion rate of 1 under BE, DVD and HC2 (see fig. 2). This happens at the cost of the periodic tasks. However, their degradation depends on the concrete policy. The completion rates under BE are lower than those under DVD and HC2. This is due to the fact that DVD and HC2 heed the remaining execution times of tasks that might be cancelled. Both approaches prefer those tasks for cancellation that have not yet started execution. HC1 achieves a nearly equal distribution of task cancellations. In task set
2. BE and DVD discriminate against tasks that have longer execution times (see fig. 3). HC1 nearly aligns the completion rates of all tasks. Due to the stronger irregularity of the task set, it cannot achieve this completely. HC2 again behaves similar to DVD, but can weaken the discrimination in some places.

All in all, HC1 achieves the best alignment of completion rates among the tasks of a task set. This results from HC1 only taking a task’s previous completion behaviour into account, if a cancellation is necessary. HC2 performs similar to DVD, as both approaches also regard a job’s (remaining) execution time.

C. Subsequent Cancellations

Next, we examine how cancellations are distributed over a task’s lifetime. Therefore, we count how often any sequence of n subsequent cancellations for each task occurs during our simulations of task sets 1 and 2. In task set 1, the longest sequence had length 9, occurring under BE, while all other policies lead to only isolated cancellations. The maximum lengths that occurs during the execution of task set 2 are displayed in table III. Again, longest cancel sequences occurred under BE scheduling, and shortest ones under HC1. Though not fully evaluated yet, these first results indicate that task that are executed using HC1 will most probably not suffer from extensive cancellation sequences.

V. CONCLUSIONS AND FUTURE WORK

We have presented history-cognisant utility functions as an extension of the well-known time-utility functions that can be used for real-time scheduling. We have demonstrated the use of HCUFs through an extension of the EDF scheduling algorithm. The results obtained in our evaluations indicate that the use of apt utility functions can help improving single-task performance in overloaded firm real-time systems. In the future, we will perform extensive evaluations with random task sets to get a more detailed image of the behaviour of our approach. Additionally, we plan to investigate further TUFs and HCUFs. We will compare our approach to other overload schedulers. Especially, we will extend our approach such that it can give the same guarantees as the (m,k)-firm real-time scheduler. Finally, we aim to integrate our HCUFs into current TUF-based schedulers.

Table III

<table>
<thead>
<tr>
<th></th>
<th>BE</th>
<th>DVD</th>
<th>HC1</th>
<th>HC2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>p2</td>
<td>7</td>
<td>9</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>p3</td>
<td>34</td>
<td>10</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>p4</td>
<td>22</td>
<td>13</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>p5</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>s1</td>
<td>8</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>s2</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>s3</td>
<td>134</td>
<td>73</td>
<td>3</td>
<td>48</td>
</tr>
</tbody>
</table>

REFERENCES