Value Based Overload Handling of Aperiodic Tasks in Offline Scheduled Real-Time Systems

Jan Carlson, Tomas Lennvall and Gerhard Fohler
Department of Computer Engineering,
Mälardalen University, Sweden
{jcn,tlv,gfr}@mdh.se

1 Introduction

This paper describes a runtime scheduling method for a mix of off-line scheduled and value based dynamic tasks with a focus on overload situations. We present a formulation of overload handling as a general binary optimisation problem and give an algorithm for solving it. Our method is based on slot shifting [4] to provide for the integration of offline and online scheduling. Dynamic tasks are scheduled according to basic EDF, extended with an algorithm for overload detection and resolution. We include a penalty value for tasks which have been guaranteed but missed their deadline, e.g., due to rejection under overload.

The mixed taskset enables designers to choose a tradeoff between predictability and flexibility for each activity in the system individually and to guarantee predictable execution of offline tasks even under overload situations. The algorithm ensures that overload situations are handled efficiently with respect to the values of the dynamic tasks.

[3], [2], [8], and [7] discuss overload handling based on earliest deadline first scheduling. [1] presents scheduling scheme calculating priorities dynamically based on value and deadlines. These focus on online scheduling. Our method handles overload situations while guaranteeing the feasible execution of offline scheduled tasks.

Assumptions We assume that the periodic tasks have been scheduled by some standard offline scheduler. Aperiodic tasks have a value (V) associated, indicating the benefit to the system if completed in time. A penalty value (PV) is used to express loss if the task has been accepted, and possibly started, but not completed in time. This way we can include tasks which provide, little value upon completion, but harm the system if aborted during execution. Unimportant tasks with critical sections, database access with locks, or actuating tasks, for example, demand this notion.

Additionally, an aperiodic task is characterized by its remaining, worst case, execution time (C) and deadline (D). The arrival times of aperiodic tasks are unknown at design time.

The system is considered to be distributed, i.e., one that consists of several processing and communication nodes [9]. The overload algorithm is performed locally, but the distributed system influences the design of the algorithm.

We assume a discrete time model [6]. Time ticks are counted globally, by synchronized clocks with granularity of slot length. Slots have uniform length, and task periods and deadlines must be multiples of the slot length.

2 Task Scheduling and Objectives

At run time, scheduling is performed via the slot shifting scheme as detailed in [4] allowing the execution of aperiodic tasks only if this does not cause an off-line task to miss its deadline.

Upon arrival, aperiodic tasks are inserted into the ready queue of dynamic tasks according to EDF, ensuring optimal performance in non-overload situations. There are two basic requirements that we want to meet. We want to maximise the accumulated value of tasks that meet their deadlines, minus the penalty value of tasks that are accepted but miss their deadline. Also, we want the ability to remove tasks early, rather than simply allowing them to miss their deadlines, since it might be possible to route them to another node in the system.

Our algorithm ensures that the EDF queue is always free from overload. When a new task arrives, the algorithm tests if it causes overload, and if so, which tasks to reject in order to resolve this.

2.1 Value and Penalty Handling

When comparing the value of a new task against that of tasks already in the queue, the difference has to be considered. Since the previously added tasks have been accepted, aborting them would not only cause a loss of value, but also an increased penalty. Rejecting the new task, on the other hand, implies a loss of value but no extra penalty.

An easy way to handle this is to define V_i', the *current value* of task τ_i, as $V_i' = V_i$ if τ_i has not been guaranteed yet, and $V_i' = V_i + PV_i$ otherwise. The current value denotes the decrease of total value associated with the removal of the task, and can be used to compare tasks fairly.

2.2 The Rejection Problem

We formulate the problem of efficient task rejection as follows. Given a set of tasks, we want to remove all overload while minimizing the total current value of the removed tasks. This can be formulated as a general binary optimisation problem.

Let $\{\tau_1, \ldots, \tau_n\}$ be the current aperiodic task set, including the new task. Also, let ft_i be the finishing time of τ_i, assuming that no tasks are removed and that no new tasks will arrive during the execution of the tasks in the set. We represent by $x_i = 1$ the selection of τ_i as a candidate for removal, and $x_i = 0$ means that τ_i is to be kept. Note that ft_i is not affected by changes to these variables. Finally, let $as[t_1, t_2]$ represent the amount of slots originally available for aperiodic tasks in the interval.

A linear method for computing finishing times of aperiodic tasks with respect to the spare capacities of the offline schedule is presented in [5], and $as[D_i, ft_i]$ can be computed in a similar way.

Now, the rejection problem can be described as follows:

$$
\begin{array}{lll}
\text{min} & V_1'x_1 + V_2'x_2 + \ldots + V_n'x_n & \\
\text{when} & C_1x_1 & \geq as[D_1, ft_1] \\
& C_1x_1 + C_2x_2 & \geq as[D_2, ft_2] \\
& \vdots & \\
& C_1x_1 + C_2x_2 + \ldots + C_nx_n & \geq as[D_n, ft_n] \\
& x_1, x_2, \ldots, x_n \in \{0, 1\} &
\end{array}
$$

Intuitively, the right hand side of the ith inequality represents the number of slots (currently used by aperiodic tasks) that must be freed in order for τ_i to meet its deadline. A negative value implies leeway, that it would be possible to execute the task later without deadline violation. In this case, the inequality is trivially satisfied. The left hand side states that the needed slots must be gathered by removing some of the tasks $\tau_1, \ldots, \tau_i$.

Example Let the active aperiodic tasks at time 10, where (D_i, C_i, V_i') represents τ_i, be:

$\tau_1 : (16, 2, 20)$	$\tau_3 : (22, 4, 60)$	$\tau_5 : (24, 6, 65)$
$\tau_2 : (18, 3, 20)$	$\tau_4 : (23, 2, 22)$	$\tau_6 : (29, 3, 60)$

τ_5 has just arrived. For simplicity, we assume a simple offline schedule, with a offline task scheduled to execute in slot 16 or 17. This gives $as[t_1, t_2] = t_2 - t_1$ for most intervals. The values in brackets indicate the impact from the offline task on the as values.

$$
\begin{array}{l}
\text{min} \\
\quad 20x_1 + 20x_2 + 60x_3 + 22x_4 + 65x_5 + 60x_6 \\
\\
\text{when} \\
\begin{array}{lcll}
2x_1 & \geq & as[16, 12] = -4 & (0) \\
2x_1 + 3x_2 & \geq & as[18, 15] = -2 & (1) \\
2x_1 + 3x_2 + 4x_3 & \geq & as[22, 20] = -2 & (1) \\
2x_1 + 3x_2 + 4x_3 + 2x_4 & \geq & as[23, 22] = -1 & (1) \\
2x_1 + 3x_2 + 4x_3 + 2x_4 + 6x_5 & \geq & as[24, 28] = 4 & (1) \\
2x_1 + 3x_2 + 4x_3 + 2x_4 + 6x_5 + 3x_6 & \geq & as[29, 31] = 2 & (1)
\end{array} \\
\\
\quad x_1, x_2, \ldots, x_6 \in \{0, 1\}
\end{array}
$$

In this case, adding τ_5 caused overload. This corresponds to a restriction which is not trivially satisfied, such as the last two restrictions above.

3 Overload detection and resolution

We assume the restrictions ordered by increasing length, as in the definition above.

Even under the assumption that $as[D_i, ft_i] \leq 0$ for $1 \leq i < n$, when all restrictions except the last one are trivially satisfied, the problem is hard to solve. In fact, it has been reduced to the well known NP-hard binary knapsack problem and an optimal solution is not likely to be found. Our algorithm is based on heuristics that exploit properties of this particular problem.

One such property is that each restriction contains less variables than the subsequent ones. We also notice that a good solution to a restriction is quite often a good solution to all subsequent restrictions. If some restrictions are not solved by this solution, it is still

a reasonably good partial solution. The reason for this is that the variables are equally weighted in all restrictions.

The rejection algorithm traverses the ordered restrictions, solving each of them individually. Since the task set was free from overload before the new task τ_y arrived, the first $y-1$ restrictions are trivially satisfied.

Initially, all x_i variables are set to 0, wich means that no tasks are to be removed. Each restriction is solved by changing some of the variables to 1, or possibly leaving them unchanged. Once we have solved a restriction with $x_j = 1$, this variable is never changed during the solving of subsequent restrictions.

A single restriction, unless trivially satisfied by the current variable settings, is solved in two steps. First from the variables $x_i = 0$ of the left-hand side of the restriction, such that $x_i = 1$ would solve the restriction, we find the one with lowest V_i'. Next, we collect variables from the left-hand side that will not solve the restriction on their own, but may solve it together. The collection is based on a greedy approach, starting with the one with lowest value density (V_i'/C_i). Finally, V_i' of the best single choice is compared against the summed V_i values of the collection, to decide what the final choice should be.

Example For the task set of the previous example, the algorithm works in the following way.

Since the new task was added at position five, the fifth restriction is solved first:

$$2x_1 + 3x_2 + 4x_3 + 2x_4 + 6x_5 \geq 4$$

It is not trivially satisfied, since all variables are 0. To find the best single candidate, we choose between x_3 or x_5 and, since $V_3' < V_5'$, x_3 is chosen. The variables to consider for the collection, sorted by value density, are x_2,x_1 and x_4. After adding x_2 and x_1 to the collection, the restriction is satisfied. Comparing $V_3' = 60$ against $V_2' + V_1' = 40$ we finally decide to solve the fifth restriction by $x_1 = x_2 = 1$

Continuing the traversal of restrictions, we find that the current variable values solves the sixth restriction as well. The found solution is:

$$x_1 = x_2 = 1,\ x_3 = x_4 = x_5 = x_6 = 0$$

meaning that the new task τ_5 is accepted while τ_1 and τ_2 is removed. The remaining tasks τ_3, τ_4 and τ_6 are kept.

4 Discussion and ongoing work

The complexity of computing all ft_i and $as[D_i, ft_i]$ values is linear. This, and the fact that we can keep a separate list of references to the tasks, sorted by increasing value density, gives us a complexity in $O(n^2)$ for the whole overload algorithm. From early results, we conclude that finding (and solving) more than a few deadline misses when a new task arrives, does not occur frequently. Thus, we might restrict the number of deadline misses that are considered in our search when a new task arrives. If we detect additional overload, we simply decide to reject the new task. This should give us an algorithm in $O(n)$ without any significant performance loss.

Another important issue we are studying is what to do with the rejection candidate tasks. We can keep them locally in a reject queue as suggested in [3]. Since the overload detection is based on worst case execution times, an overload situation might not necessarily cause deadline misses. Tasks in the reject queue can be considered for execution again if a task finishes earlier than its worst case execution time implies. But since we have a distributed system, they can also be moved to other nodes where they might be accepted. We experiment with making the reject queues visible and accessible to all the other nodes in the system, so that they can try to "steal" the tasks that appear promising to them. Accepting a stolen task may result in overload and the removal of other tasks from the node. In this way we expect some value balancing in the system, that is the overall value of the system will go up if some tasks are distributed.

Simulations are carried out to evaluate the linear version of the algorithm, and to compare the distributed stealing method against other distributed algorithms.

So far we have always kept the offline tasks, and never considered them for rejection. We can relax this constraint by using one of the properties of slot shifting, namely that the offline preparation creates EDF tasks. Then we can treat some of the offline tasks in the same way as the aperiodic tasks while still guaranteeing the execution of the other offline tasks.

References

[1] S. A. Aldarmi and A. Burns. Dynamic value-density for scheduling real-time systems. In *Proceedings 11th Euromicro Conference on Real-Time Systems*, Dec 1999.

[2] G. Buttazzo, M. Spuri, and F. Sensini. Value vs. deadline scheduling in overload conditions. In *Real-Time Systems Symposium*, Pisa, Italy, Dec 1995.

[3] G. Buttazzo and J. Stankovic. Red: A robust earliest deadline scheduling algorithm. In *Proceedings of 3rd International Workshop on Responsive Computing Systems*, 1993.

[4] G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proc. 16th Real-time Systems Symposium*, Pisa, Italy, 1995.

[5] D. Isovic and G. Fohler. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, Orlando, Florida, USA, Nov. 2000.

[6] H. Kopetz. Sparse time versus dense time in distributed real time systems. *In Proc. of the Second Int. Workshop on Responsice Comp. Sys., Saitama, Japan*, Oct. 1992.

[7] G. Koren and D. Shasha. Gilad koren and dennis shasha. skip-over: Algorithms and complexity for overloaded systems that allow skips. In *Proceedings Real-Time Systems Symposium*, Pisa, Italy, Dec 1995.

[8] S.Baruah and J. Haritsa. Scheduling for overload in real-time systems. *IEEE Transactions on Computers*, 46(9), September 1997.

[9] J.A. Stankovic, K. Ramamritham, and C.-S. Cheng. Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems. *IEEE Trans. on comp., 34(12)*, Dec 1995.