Enhancing Time Triggered Scheduling with Value Based Overload Handling and Task Migration

Jan Carlson, Tomas Lennvall, and Gerhard Fohler
Department of Computer Science and Engineering
Mälardalen University, Sweden
{jan.carlson, tomas.lennvall, gerhard.fohler}@mdh.se

Abstract

Time triggered methods provide deterministic behaviour suitable for critical real-time systems. They perform less favourably, however, if the arrival times of some activities are not known in advance, in particular if overload situations have to be anticipated. In many systems, the criticality of only a subset of activities justify the cost associated with the time triggered methods.

In this paper we consider distributed systems where a subset of critical activities are handled in a time triggered fashion, via an offline schedule. At runtime, the arrival of aperiodic tasks may cause overload that demands to be handled in such a way that i) time triggered activities still meet all their original constraints, ii) execution of high-valued tasks are prioritised over tasks with lower value, iii) tasks can be quickly migrated to balance the overall system load.

We give a precise formulation of overload detection and value based task rejection in the presence of offline scheduled tasks, and present a heuristic algorithm to handle overload. To benefit from the distributed setting, the overload handling includes an algorithm that integrates migration of rejected tasks with resource reclaiming and an acceptance test of newly arrived tasks.

Simulation results underline the effectiveness of the presented approach.

1. Introduction

The time triggered approach has been shown to be suitable for critical real-time systems [9] [10]. By applying strict temporal control, critical activities can be performed in a deterministic way. Since scheduling is performed offline, sufficient time can be spent constructing a feasible schedule to allow complex constraints, e.g., concerning task separation or jitter.

However, time triggered scheduling performs less favourably with activities for which the arrival time is not known in advance. If overload situations have to be anticipated to occur at runtime, a time triggered design would typically restrict the number of activities for the entire system lifetime, although solving occasional overload situation by rejection of less important tasks would be acceptable. Designing the system for worst case load would in many cases result in a prohibitively overdimensioned system.

In distributed systems, it is possible that overload situations occur on a set of processing nodes although the system is globally underloaded. Such situations can be resolved by migrating tasks from overloaded nodes to such with lower load.

For many systems, only a subset of activities justifies the cost associated with time triggered methods. In addition to this critical subset, the system may perform a number of other activities of lower importance which may be of different relative importance to the overall system performance.

As an example, imagine a system with a critical core responsible for system stability, but less stringent applications; while a failure in the core system is unacceptable, reduced application performance can be tolerated at overload, as in, e.g., a telephone switch.

In this paper we consider distributed systems where a subset of activities are handled in a time triggered fashion. At runtime, the arrival of aperiodic tasks may cause overload, which cannot be planned for in advance. If overload occurs, it must be detected and resolved in such a way that:

i) time triggered activities still meet all their original constraints,
ii) execution of high-valued tasks are prioritised over tasks with lower value,
iii) tasks can be quickly migrated between nodes to balance the overall system load.

We describe how the time triggered approach can be enhanced to suit distributed real-time systems where overload situations must be anticipated. We give a precise formulation of overload detection and value based task rejection in

the presence of offline scheduled tasks, and present a heuristic overload handling algorithm. Overload situations are detected immediately when the offending tasks arrive, and resolved by rejection of low value tasks.

The overload handling includes a task migration algorithm to benefit from the distributed setting, that integrates migration of rejected tasks with resource reclaiming and the acceptance test of newly arrived tasks.

We assume that the critical tasks are scheduled offline, but the schedule is handled in a flexible way at runtime to facilitate the inclusion of aperiodic tasks. This is achieved by including mechanisms from the slot shifting algorithm [7] that allow the planned execution of offline scheduled tasks to be shifted in time, while still ensuring that no critical constraints are violated. This allows the designer to choose, for each activity individually, the tradeoff between guaranteed timely execution, and less resource demanding non-guaranteed handling based on values.

Value based overload handling has been thoroughly investigated. In [3], a number of methods that use values and deadlines to handle overload are compared. For a wide range of overload conditions, the best performance was achieved by EDF scheduling extended with a value based overload recovery mechanism and resource reclaiming. An example of such an algorithm is RED [4]. For very high overloads, scheduling based on value density outperforms EDF based methods. In [1], task priorities are calculated dynamically from values and remaining execution times. They consider tasks with soft deadlines, i.e., values that decrease if the deadline is missed, rather than become zero or negative. In [2], an overload algorithm is presented for the special case when a minimum slack factor for every task is known. Also, tasks are assumed to be equally important.

These methods do not consider distributed scheduling, or overload handling in the presence of offline scheduled critical tasks.

Distributed overload handling is addressed in, e.g., [12], where an acceptance test is performed upon arrival of aperiodic tasks. If it fails, the node initiates an intricate bidding procedure in which nodes cooperate to decide where to migrate the task. The problem considered in this paper requires an overload handling where values are taken into account. Another difference is that in our method migration is initiated by the receiving node rather than the current owner of the task, and that migration is integrated with resource reclaiming and the acceptance test of new aperiodic tasks.

The rest of this paper is organised as follows: Section 2 lists task and system assumptions and discusses our method in general. The task migration algorithm is presented in Section 3, followed by a description of the local overload handling in Section 4. Simulation results are given in Section 5. Finally, Section 6 concludes the paper.

2. System assumptions and basic idea

This section describes system assumptions, task model and value model we used. It also includes a brief description of how the mixed task set is handled, and the basic idea of the proposed method.

We consider a *distributed* system, i.e., one that consists of several processing and communication nodes [13]. All nodes are assumed to have access to the static parameters, including code, of all aperiodic tasks. This simplifies task migration since only task identifiers are sent over the network. Also, we only migrate tasks that have not started executing on a node, and thus no additional data transfer is required.

We assume a discrete time model [8]. Time ticks are counted globally by a synchronised clock, and assigned numbers from 0 to ∞. The time between two consecutive ticks is called a *slot*. Slots have uniform length, and they start and end at the same time for all nodes in the system.

2.1. Task model

We assume two different task types in the system: offline scheduled tasks and aperiodic tasks, described below. All tasks are fully preemptive and communicate with the system via data read at the beginning and data written at the end of execution. Hard deadlines must be met under any circumstance. Firm deadlines can be missed, but a result delivered after the deadline is of no use to the system.

Offline scheduled tasks have hard deadlines and can have complex constraints, such as distribution, precedence, instance separation, jitter, etc. Solving these constraints online is not feasible in the general case, due to the high complexity. Instead, the offline scheduled tasks are transformed by an offline scheduler, into simple runtime tasks with simple constraints: earliest start time, worst case execution time (*wcet*), and a relative deadline.

Transformation of complex constraints into simpler ones is discussed in [6], where an offline scheduler, e.g., [11] is used and the resulting schedule analysed to establish the new parameters.

Aperiodic tasks have firm deadlines, and arrival times unknown at design time. We also assume aperiodic tasks to be independent of each other, and of the offline scheduled tasks. An aperiodic task is characterised by the following set of parameters: arrival time, remaining worst case execution time (c), firm absolute deadline (dl), and value (v).

The term *aperiodic* reflects the fact that the system has no knowledge of arrival times and thus do not consider the arrival of future instances when scheduling. Aperiodic tasks can still be used to handle non-critical periodic activities.

Value is a measure of the benefit to the system associated with completing the task in time. Only aperiodic tasks are associated with values, since offline scheduled tasks are never considered for rejection when resolving overload situations. The values are considered to be cumulative, i.e., two sets of tasks can be compared by their respective sum of values. Tasks contribute with their value to the system if they finish in time, otherwise they do not contribute at all. In this paper we assume static values ranging from 1 to *MaxValue*, where a higher value indicates a greater benefit.

2.2. Handling the mixed task set

At runtime, local scheduling uses the slot shifting algorithm described in [7], except for the guarantee mechanism. Slot shifting introduces flexibility into the offline schedule by allowing offline scheduled tasks to be shifted in time, but never in such a way that their timely execution is impeded.

Information about this flexibility, i.e., available resources and leeway in the offline schedule, is represented as *spare capacity* of disjoint time intervals. This information is used by the runtime scheduler to decide for each slot whether to execute an aperiodic or an offline task. In this paper, the spare capacity of these fixed intervals are only considered as a way to determine the spare capacity of arbitrary future intervals when handling overload.

2.3. Basic idea

As outlined above, slot shifting is used to decide when aperiodic tasks can be allowed to run without causing an offline scheduled task to miss its deadline. In addition, the scheduler must decide which aperiodic task to execute. In the proposed method, aperiodic tasks are served according to EDF once accepted by the overload detection mechanism.

To handle overload situations, each node keeps the *ready queue*, containing the aperiodic tasks ready to be executed on that node, constantly free from overload. When new aperiodic tasks arrive, they are inserted into the ready queue based on their deadlines. Then, the queue is processed to detect future overload situations and to resolve them to make the queue free from overload again.

All tasks removed from the ready queue due to overload are stored in a separate *maybe-later queue*, as long as they have positive laxity. This queue is similar to the *reject queue* in RED [4], but used for tasks migration as well as resource reclaiming.

The basis of the task migration algorithm is that selected tasks from maybe-later queues are retried, possibly on other nodes. Retrying tasks locally is required to reclaim resources when tasks finish in less time than wcet. If a task is accepted on the new node, it is immediately migrated. An important aspect of this scheme is that a task is only migrated if it has been found non-profitable for local execution, and if there is room for it on the new node, possibly after rejecting a number of lower valued tasks.

3. Remote task stealing

A distributed system with runtime task migration must somehow decide when and where to move tasks in order to maximise the total value of executed tasks. These decisions become increasingly important when the load, or the value of tasks, varies a lot between nodes. Ensuring optimal global scheduling is an NP-hard problem, and we therefore aim for a sub-optimal solution.

In order to cope with the complexity of the problem, scheduling is primarily handled locally on each node, as discussed in Section 4. Task migration is handled together with acceptance tests of new tasks, and local resource reclaiming. Further, task migration is always initiated by the node the task is to migrate to, and not the current owner. Therefore, we use the term *task stealing*, rather than migration.

To keep network usage low, and to simplify the algorithm by ruling out the possibility of conflicting thefts, only one node at a time is allowed to steal tasks. This is ensured by something similar to a conceptual token ring, where the owner of the token may steal tasks from any other node during one slot, before the token is passed to the next node in the ring.

By some arbitrary communication scheme, the maybe-later queues (or parts of them) are made visible to all nodes in the system. At the start of a slot, each node adds newly arrived aperiodic tasks to its ready queue. In addition, the node holding the token may add tasks from any maybe-later queue in the system, including its own. After adding tasks, each node applies the overload handling algorithm to resolve any overload situations.

Since only one node is allowed to steal tasks from any maybe-later queue at the start of each slot, and no additional data have to be sent over the network, the stealing node may execute one of the stolen task immediately (in the current slot).

The Flea Market algorithm[1]

The parameter *MaxTheft* is used to adjust the algorithm w.r.t. network capacity and system size. At the start of every slot, each node performs the following algorithm:

1. Let A be the set of all aperiodic tasks currently in the ready queue.

[1] The name reflects that once a node is no longer interested in a task, the task is offered to the other nodes of the system, and given to the first node that wants it.

2. Add to A all aperiodic tasks that arrived to the node at this tick.
3. This step is only performed by the node currently holding the token. Gather tasks from the maybe-later queues of all nodes in the system. From the maybe-later queues of other nodes, consider only tasks that are movable. Add to A the tasks with highest value density, at most *MaxTheft* tasks.
4. Apply the overload algorithm to A. The result is a boolean value x_i for each $a_i \in A$, where 0 represents acceptance and 1 rejection. For each x_i, perform the following action depending on whether the task a_i was added during step 1, 2 or 3 of this algorithm.

x_i	step	action
1	1	Remove a_i from ready queue, and insert it in the maybe-later queue.
1	2	Insert a_i into maybe-later queue.
1	3	Do nothing.
0	1	Do nothing.
0	2	Insert a_i into ready queue.
0	3	Insert a_i into ready queue, and inform the current owner (possibly yourself) of the theft.

5. If the node holds the token, send it to the next node.

3.1. Node communication

The algorithm is described as if the whole maybe-later queues are visible to all nodes, but this is actually not required. The node holding the token is interested only in the *MaxTheft* tasks with highest value density. By keeping maybe-later queues sorted according to value density, it is sufficient to make the *MaxTheft* first tasks in each queue visible. Also, since aperiodic tasks are assumed to reside on all nodes in the system, only tasks identifiers are sent over the network.

Furthermore, only one node uses the maybe-later queues each slot. Thus, the distribution of maybe-later queue information in a system of n nodes can be accomplished by a total of $n-1$ messages, each consisting of *MaxTheft* task identifiers and remaining execution time.

Communication is also required in order to migrate tasks. Since only one node may steal tasks from the maybe-later queues in each slot, the only communication needed in order to migrate a task is to inform the current owner of the theft. Thus, a stolen task may execute on the new node in the same slot as it is stolen. At most $n-1$ messages, each containing one task identifier, are sent each slot due to task migration.

The algorithm, as described above, assumes that the network is fast enough to permit the following communication during a single slot:

- The node holding the token sends theft messages to all nodes.
- When receiving the theft message, each node sends its new maybe-later queue information to the next token holder.

If the network does not permit this within a single slot, but within t slots, the algorithm can be modified so that the token is inactive for $t-1$ slots when it arrives to a node. Figure 1 and 2 show the communication between three nodes for $t=1$ and $t=3$. Ticks are denoted by vertical lines, and the scheduling performed in each slot is represented by a grid. Horizontal lines denote the token holder, and dashed lines represent that the token is inactive. Arrows starting in a grid are messages concerning stolen tasks, and those starting in the middle of a slot are messages containing maybe-later queue information.

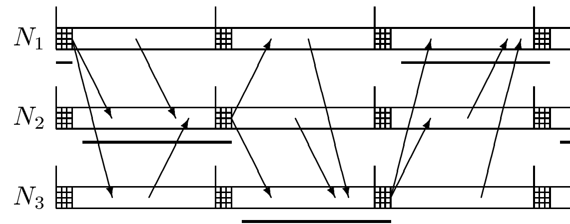


Figure 1. Node communication ($t=1$).

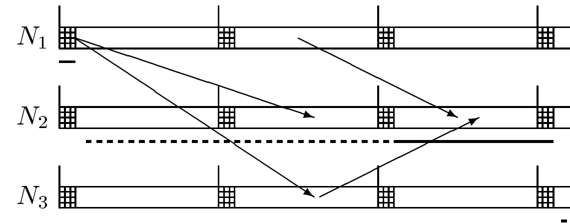


Figure 2. Node communication ($t=3$).

4. Overload handling

At run time, scheduling is performed locally via the slot shifting scheme, which decides for each slot if an aperiodic task can be allowed to execute without causing an offline scheduled task to miss its deadline.

Aperiodic tasks are served according to EDF, which gives good performance in non-overload situations. When the system is overloaded, two important issues must be addressed. In general, high valued tasks should be preferred over tasks with low value. Additionally, tasks should be removed as early as possible, rather than simply being allowed to miss their deadlines, since an early removal might allow the task to be stolen by another node in the system.

Our algorithm ensures an overload-free ready queue, i.e., all tasks in the queue can be executed without missing their deadlines, also in the presence of offline scheduled tasks. When new aperiodic tasks arrive, the algorithm checks if they cause overload, and if so, which tasks to reject in order to resolve this efficiently.

4.1. Problem formulation

Detection and removal of overload can be formulated as a general binary optimisation problem. This allows us to abstract on details, since the dynamic aspects of the rejection problem (e.g., that rejecting a task influences the finishing times of the others) are represented by static restrictions. This facilitates the development of a suitable algorithm.

Let $\tau_1, \ldots, \tau_n$ be the aperiodic tasks currently in the ready queue, including the ones that just arrived, sorted according to EDF. For each task τ_i we use a boolean variable x_i to represent whether the task should be kept in the ready queue ($x_i = 0$), or rejected ($x_i = 1$). These variables are the output of the overload algorithm, used by the Flea Market algorithm described in Section 3.

To explain the problem formulation, we first consider a simpler setting without offline scheduled tasks, and then proceed by showing the modifications needed to incorporate offline scheduled tasks as well.

Consider a single aperiodic task τ_i. To detect if there is a risk of this task missing its deadline, we need the expected finishing time, denoted ft_i. In a pure EDF setting, with no offline scheduled tasks to consider, this would be computed by adding the remaining execution times $c_1, \ldots, c_i$ to the current time.

However, detecting overload is not enough. To solve it efficiently we need to know the size of each deadline miss, so we denote by σ_i the overload amount of τ_i, defined in the simple setting as $ft_i - dl_i$. In order to ensure that τ_i does not miss its deadline, at least σ_i slots must be freed, by removing some of the tasks $\tau_1, \ldots, \tau_i$. This is represented by the following restriction:

$$c_1x_1 + c_2x_2 + \ldots + c_ix_i \quad \geq \sigma_i$$

Similar reasoning can be applied to each of the tasks in the ready queue, resulting in the following set of restrictions:

$$\begin{array}{ll} c_1x_1 & \geq \sigma_1 \\ c_1x_1 + c_2x_2 & \geq \sigma_2 \\ \vdots & \\ c_1x_1 + c_2x_2 + \ldots + c_nx_n & \geq \sigma_n \end{array}$$

Note that these restrictions give a static formulation of the problem, since the σ-values are defined in term of the current ready queue, and do not depend on the x-values.

An assignment of the values 0 or 1 to the x-variables corresponds to a potential solution to the task rejection problem. Furthermore, any assignment that satisfies the restrictions corresponds to a solution that would result in a ready queue free from overload. However, we do not simply look for a solution (rejecting all tasks is always a valid possibility), we want a solution that gives as high value as possible to the system. This means that the summed values of the removed tasks should be minimised, which is represented as:

$$\min \quad v_1x_1 + v_2x_2 + \ldots + v_nx_n$$

So far, we have considered a simplified system that contains only aperiodic tasks. In order to construct similar restrictions when offline scheduled tasks also have to be considered, the definition of σ_i must be modified.

Let $sc[a, b]$ be the spare capacity of the interval from a to b, i.e., the number of slots in the interval that is not required to execute offline scheduled tasks in time. Now, σ_i can be defined as follows:

$$\sigma_i = sc[dl_i,\ ft_i]$$

This definition requires the expected finishing time to be computed, and now that the system contains offline scheduled tasks as well, this is not straightforward. Instead, we use the following definition, which is equivalent to the previous one except that it assigns negative values rather than zero to tasks that finish before the deadline. In this definition, t_c denotes the current time.

$$\begin{array}{rcll} \sigma_1 & = & c_1 - sc[t_c,\ dl_1] & \\ \sigma_i & = & \sigma_{i-1} + c_i - sc[dl_{i-1},\ dl_i] & (1 < i \leq n) \end{array}$$

The modified definition of σ_i allows the same restrictions to be used as in the simplified setting, and the final representation of task rejection as a optimisation problem is:

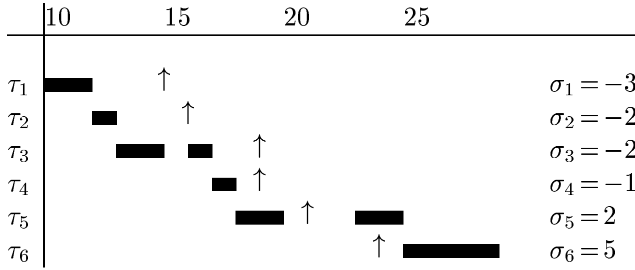
$$\begin{array}{llll} \min & v_1x_1 + v_2x_2 + \ldots + v_nx_n & & \\ \text{when} & c_1x_1 & \geq & \sigma_1 \\ & c_1x_1 + c_2x_2 & \geq & \sigma_2 \\ & \vdots & & \\ & c_1x_1 + c_2x_2 + \ldots + c_nx_n & \geq & \sigma_n \\ & x_1, x_2, \ldots, x_n \in \{0, 1\} & & \end{array}$$

Example: Let the ready queue contain the following aperiodic tasks at the beginning of slot 10, where (dl_i, c_i, v_i) represents τ_i.

$$\begin{array}{lll} \tau_1 : (15, 2, 20) & \tau_3 : (19, 3, 10) & \tau_5 : (21, 4, 20) \\ \tau_2 : (16, 1, 10) & \tau_4 : (19, 1, 5) & \tau_6 : (24, 4, 20) \end{array}$$

The tasks τ_3 and τ_6 have just arrived, and might have caused overload. If no more tasks were to arrive, the execution of

the aperiodic tasks would look as follows. The arrows denote deadlines, and the gaps indicate slots needed to execute offline tasks. For simplicity, we assume that the offline schedule has a low load in the interval.



The corresponding optimisation problem is:

$$\begin{array}{lll}
\min & 20x_1 + 10x_2 + 10x_3 + 5x_4 + 20x_5 + 20x_6 & \\
\text{when} & 2x_1 & \geq -3 \\
& 2x_1 + 1x_2 & \geq -2 \\
& 2x_1 + 1x_2 + 3x_3 & \geq -2 \\
& 2x_1 + 1x_2 + 3x_3 + 1x_4 & \geq -1 \\
& 2x_1 + 1x_2 + 3x_3 + 1x_4 + 4x_5 & \geq 2 \\
& 2x_1 + 1x_2 + 3x_3 + 1x_4 + 4x_5 + 4x_6 & \geq 5 \\
& x_1, x_2, \ldots, x_6 \in \{0, 1\} &
\end{array}$$

The last two inequalities correspond to the overload at τ_5 and τ_6, and describe what must be done in order to resolve this.

4.2. Rejection algorithm

Even when all restrictions except the last one are trivially satisfied ($\sigma_i \leq 0$ for $1 \leq i < n$), the problem is hard to solve. In fact, it has been reduced to the well known NP-hard binary knapsack problem, which indicates that an optimal alogrithm is not feasible. Instead, 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. Furthermore, a good solution (w.r.t. the minimisation criteria) to a single restriction is a reasonably good partial solution to all subsequent restrictions, since the variables are equally weighted in all restrictions.

Algorithm description. Initially, all x_i variables are set to 0, which represents a solution where no tasks are removed. The rejection algorithm traverses the restrictions top-down, solving each of them individually.

The restrictions are solved by changing some of the variables from 0 to 1. Once a variable is set to 1, this variable is never changed during the solving of subsequent restrictions.

Each restriction, unless already satisfied by the current variable settings, is solved in three steps.

- First, we consider the variables of the left-hand side of the restriction that are currently set to 0, and would solve the restriction if set to 1. From these we select as our *best single candidate* the one with lowest v_i.
- Next, we construct the *collection candidates*. From the remaining left-hand side variables that are currently set to 0 (i.e., those that would not solve the restriction if set to 1), we collect variables from right to left until the restriction would be solved if all variables in the collection are set to 1.
- Finally the value of the best single candidate is compared against the summed values of the collection candidates (if a large enough collection was found), to decide what the final choice should be.

Complexity. Computing σ-values for n aperiodic tasks can be done in linear time. The algorithm has been left out due to space limitations, but can be found in [5]. In the worst case, all n restrictions have to be solved and none of the solutions solve any subsequent restriction. Solving a single restriction requires a linear traversal of all earlier tasks, which gives the algorithms a worst case complexity in $O(n^2)$.

In practical applications, this worst case complexity can be handled in essentially two ways. We can restrict the overload algorithm to consider only a prefix of the ready queue. Simulations presented in [5] show a moderate impact on system performance when this type of restriction is applied. Another possibility is to restrict the number of non-trivially solved restrictions that are considered at each call to the overload algorithm. If this number is reached, the system rejects all new tasks, which is always a valid option.

Efficiency improvements. Let τ_y be the new task that has the earliest deadline. Since the task set was free from overload before the new tasks arrived, the first $y-1$ restrictions are trivially satisfied and do not need to be considered.

If, at any point, the sum $\sum_{i=0}^{n} v_i x_i$ becomes greater than the summed value of the newly arrived tasks, the algorithm stops, returning an answer where all new tasks are rejected, and all old tasks are kept. This improvement ensures that the total value of the ready queue is never decreased when new aperiodic tasks arrive.

Example: For the task set in the previous example, the algorithm works in the following way. The first new task was added at position three, so we need only to check restrictions three to six. The third and fourth restrictions are trivially satisfied, because the σ-values are negative, but restriction five needs to be solved.

$$2x_1 + 1x_2 + 3x_3 + 1x_4 + 4x_5 \geq 2$$

To find the best single candidate, we choose between x_1, x_3 and x_5. Since v_3 is smaller than v_1 and v_5, x_3 is chosen

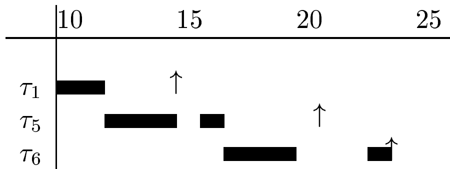
as single candidate. Constructing the collection, both x_4 and x_2 are added before the collection is large enough to solve the restriction. Comparing $v_3 = 10$ against $v_4 + v_2 = 15$ we finally decide to solve the fifth restriction by $x_3 = 1$.

Continuing the traversal of restrictions, we now consider the last one:

$$2x_1 + 1x_2 + 3x_3 + 1x_4 + 4x_5 + 4x_6 \geq 5$$

We find that the current variable values do not satisfy this restriction, and the procedure of finding the best single candidate and a collection is repeated. This time, the collection has a lower value, and the restriction is solved by $x_2 = x_4 = 1$

The solution to the whole problem is $x_2 = x_3 = x_4 = 1$, $x_1 = x_5 = x_6 = 0$, meaning that τ_6 is accepted, while τ_2, τ_3 and τ_4 are removed from the ready queue. Since all restrictions are satisfied by this solution, the ready queue is once again free from overload. The future execution of aperiodic tasks, assuming no further arrivals, is:



5. Simulations

We have implemented the described method, and have run simulations for various scenarios. The simulated system consists of 8 processing nodes, connected via a network where all necessary messages can be sent during one time slot.

Each simulation has a length of 2000 slots. The offline schedules are created from randomly generated precedence graphs, an offline scheduler transforms the precedence graphs to offline schedules. Each node has one offline schedule with a load of 0.4 and a length between 300 and 1000 slots.

Worst case computation time for both offline and aperiodic tasks varies uniformly in the range 1–10. Aperiodic tasks are assigned an actual execution time uniformly distributed between 0.5 and 1.0 of its wcet, and relative deadlines varying between 1–3 times wcet.

Arrival times of aperiodic tasks are distributed over the simulation length, with the restriction that no task have a deadline exceeding the simulation length. Finally, values of aperiodic tasks vary uniformly in the range 1–100.

The average node load varies between 0.8 and 3.0, the offline load of 0.4 included. The load parameter is based on wcet, and thus represents the load as perceived by the overload algorithm. The actual system load is lower[2], since execution time is less than wcet.

We have studied the total accumulated value of aperiodic tasks that finished in time, and the following methods have been compared:

1. The full method presented in the paper.
2. The overload handling algorithm, without task migration.
3. A basic algorithm that uses the offline schedule, assigning idle slots to the aperiodic tasks based on value density.
4. Same as 3, but aperiodic tasks are ordered by value.
5. Same as 3, but aperiodic tasks are ordered EDF.
6. Same as 3, but aperiodic tasks are serviced in order of arrival.

Methods 1 and 2 implement the efficiency improvements suggested in Section 4.2. Each point in the figures represents some 300 simulations.

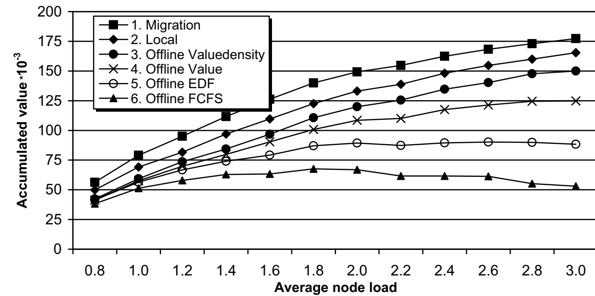


Figure 3. Even load distribution.

In the first experiment, all nodes in the system are subject to the same amount of load. The result is presented in Figure 3. Because all nodes are overloaded, the possibility of task migration does not provide any significant improvement. Compared to the basic methods, the proposed method performs better.

The second experiment, shown in Figure 4, is a scenario of unevenly distributed load. Half of the nodes have no aperiodic tasks arriving, only offline scheduled tasks. Here, the task migration algorithm clearly increases the system performance, compared to overload handling without migration, because tasks can migrate to nodes with no aperiodic load.

6. Conclusions

In this paper we have described how the time triggered approach can be enhanced to suit distributed real-time

[2]The actual system load varies approximately between 0.7 and 2.35 in the experiments, based on the distribution of actual execution times

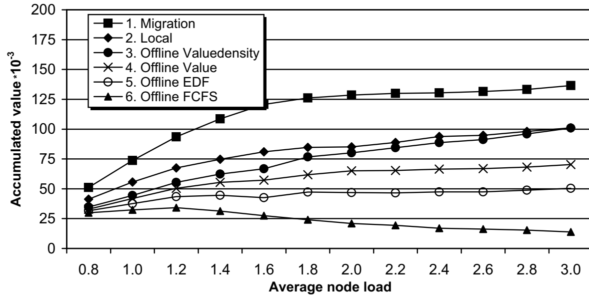


Figure 4. Uneven load distribution.

systems where overload situations have to be anticipated. Overload situations are resolved w.r.t. task value, possibly by migration of tasks to nodes of lower load, without impeding the timely performance of critical activities.

For many systems, the cost associated with time triggered methods is only justified for a subset of activities. In addition to this critical subset, the system may perform a number of other non-critical activities, which may be of different relative importance to the overall system performance.

We have formulated a binary optimisation problem that represents overload detection and value based task rejection in the presence of offline scheduled tasks that are guaranteed a timely execution.

We have also presented a heuristic overload handling algorithm that detects overload situations immediately when the offending tasks arrive, and resolve them by rejection of low value tasks. The overload resolver, although not optimal, never decreases the value of aperiodic tasks in the overload-free ready queue.

As distributed systems were considered, the overload handling includes a task migration algorithm that integrates migration of rejected tasks with resource reclaiming and the acceptance test of newly arrived tasks. Task migration is initiated by the receiving node. It is only applied to tasks that have been rejected by their current owner, and will increase the value of the receiving node. A task can execute on the new node in the same slot it was migrated.

Critical tasks are scheduled offline, which allows complex constraints, such as distribution, precedence and jitter, to be considered. Using mechanisms from the slot shifting method, the schedule is handled in a flexible way at runtime to facilitate the execution of aperiodic tasks, while still ensuring that no critical constraints are violated.

This enables designers to choose the tradeoff between predictability and flexibility individually for each activity in the system. It guarantees predictable execution of critical activities even under overload situations, while minimising response times and maximising accumulated values.

Simulation results show the effectiveness of our approach for loads up to 3.0, evenly and unevenly distributed over the nodes, compared to a basic algorithm that uses the offline schedule directly and assigns idle slots to execution of aperiodic tasks in order of arrival. The results also show the performance increase due to task migration.

An interesting future extension to this work would be to allow offline scheduled tasks to be associated with values as well, and thus included in the rejection process. The difference, compared to including them as aperiodic tasks in the current method, would be that the information about future instances could be taken into account.

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

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

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

[5] J. Carlson, T. Lennvall, and G. Fohler. Simulation results and algorithm details for value based overload handling. Technical report, Department of Computer Engineering, Mälardalen University, Sweden, May 2002.

[6] G. Fohler. *Flexibility in Statically Scheduled Hard Real-Time Systems*. PhD thesis, Technische Universität Wien, Austria, Apr. 1994.

[7] G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proceedings of the 16th Real-Time Systems Symposium*, Pisa, Italy, Dec. 1995.

[8] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *12th International Conference on Distributed Computing Systems*, pages 460–467, Washington, D.C., USA, June 1992. IEEE Computer Society Press.

[9] H. Kopetz. Time-triggered model of computation. In *Proceeedings 19th Real-Time Systems Symposium*, pages 168–177, Dec 1998.

[10] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The MARS Approach. *IEEE Micro*, 9(1):25–40, Feb. 1989.

[11] K. Ramamritham. Allocation and scheduling of complex periodic tasks. In *Internation Conference on Distributed Computing Systems*, pages 108–115, 1990.

[12] K. Ramamritham, J. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Transactions on Computers*, C-38(8):1110–1123, August 1989.

[13] J. 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.