Mälardalen University Licentiate Thesis
No.12

Handling Aperiodic Tasks and Overload in Distributed Off-line Scheduled Real-Time Systems

Tomas Lennvall
May 2003

Department of Computer Science and Engineering
Mälardalen University
Västerås, Sweden

ISBN 91-88834-03-4
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

[1] Paper A, B, and C © IEEE

Abstract

System designers face many choices when designing a real-time system. They have to decide how to deal with the original requirements imposed on the system, which operating system (OS), OS functionality, and scheduling algorithm. Ideally designers have a lot of freedom when choosing the most suitable configuration for the system. Unfortunately this is not the case in most present day situations.

Real-time applications impose complex constraints, such as precedence, end-to-end deadlines, jitter, and distribution, in addition to non-complex constraints, such as periods and deadlines. There might also be a need to handle on-line activities, such as aperiodic or sporadic tasks, or even to anticipate overload during run-time.

On-line scheduling provide flexibility and supports overload handling, but handling complex constraints can be costly or even intractable.

On the other hand, *off-line* scheduling resolves complex constraints and provides determinism at the cost of flexibility.

This disparity between scheduling paradigms forces designers to choose either flexibility or determinism.

OS kernels are usually monolithic, meaning that kernel functionality, dispatcher, and scheduling algorithms are usually intertwined to achieve higher performance, at the cost of limiting the designers choice.

In this thesis we provide designers with some methods to alleviate these problems. We increase the flexibility in off-line scheduled systems by using the total bandwidth server (TBS) for aperiodic task handling.

Furthermore we provide overload handling in off-line scheduled systems, which is handled in such a way that the original constraints are still met. Tasks can also be migrated from overloaded nodes to provide load balancing.

We also propose a plug-in scheduling architecture where we disentangle the scheduling algorithm from the kernel routines providing for easy replacement of algorithm, as opposed to the monolithic kernels. This gives designers more flexibility in choosing the appropriate scheduling algorithm independently from the OS.

Acknowledgements

There are many people I wish to thank for supporting me during the years I have spent working for this licentiate thesis.
First I wish to thank my supervisor Gerhard Fohler for guiding me through my graduate education and for many interesting technical discussions, most related to this work but also some about common spare time interests.
Furthermore I wish to thank my colleagues here at the Department of Computer Science and Engineering, especially the “salsart” group: Damir Isović, Radu Dobrin, and Larisa Rizvanović for all the help they have provided during this time.
Many thanks also to all my other colleagues here at the department, for making this place fun to work in.
During my studies I have also met colleagues from other countries, especially from Italy. I want to thank all the people in the Retis Lab, Scuola Superiore S.A., Pisa, who took care of me during my visit, thanks Giorgio, Gerardo, Peppe, Luigi, Luca, and Marco, and of course my good friend Paolo Gai.
Finally I want to thank my parents (and my brother) for their support during my studies.

Västerås, March 2003
Tomas Lennvall

Contents

List of Figures

List of Publications

The following articles are included in this licentiate[2] thesis:

A *Improved Handling of Soft Aperiodic Tasks in Off-Line Scheduled Real-Time Systems using Total Bandwidth Server*, Tomas Lennvall, Giorgio Buttazzo and Gerhard Fohler, In Proceedings of 8th International Conference on Emerging Technologies and Factory Automation, Nice, France, October 2001.

B *Handling Aperiodic Tasks in Diverse Real-Time Systems using Plug-Ins*, Tomas Lennvall, Björn Lindberg, and Gerhard Fohler, In Proceedings of the 5th International Symposium on Object-Oriented Real-Time Distributed Computing, Washington D.C., USA, April-May 2002.

C *Enhancing Time Triggered Scheduling with Value Based Overload Handling and Task Migration*, Jan Carlson, Tomas Lennvall, and Gerhard Fohler, In Proceedings of 6th International Symposium on Object-Oriented Real-Time Distributed Computing, Hakodate, Japan, May 2003.

D *Simulation Results and Algorithm Details for Value Based Overload Handling*, Jan Carlson, Tomas Lennvall, and Gerhard Fohler, Technical Report, Mälardalen University, 2002.

[2] A licentiate degree is a Swedish graduate degree halfway between MSc and PhD.

Chapter 1

Introduction

1.1 Real-Time Systems

Real-time systems are becoming more and more commonplace, and are used in applications ranging from cars, airplanes, and factory automation, to mobile phones, and multimedia systems. Real-time systems are defined as systems where: *"not only the functional but also the timely correctness is important"*, i.e., not only the logical correctness of the computations performed are important, but also at which time these computations are complete [1].
Each computation has an associated *deadline*. If the computation does not complete before the deadline the system is considered to fail, and depending on the category of the system, this can lead to serious consequences.
Real-time systems are usually divided into two categories: *hard* and *soft*.
Hard real-time systems have stringent requirements on both the functional and timely behavior, if either of these fail the consequences can be catastrophic, such as damage to people or property.
Soft real-time systems, on the other hand, are systems that can tolerate an occasional failure of the timing requirements without any severe consequences, e.g., only a possible degradation of performance or quality.
Real-time systems consist of applications and resources, where resources usually consist of one or several CPUs. Applications consist of tasks that cooperate to achieve the global goal of the application. To avoid contention and conflict between these tasks over the resources that exist in the system, i.e., the CPU(s), a scheduling algorithm need to determine when to execute the tasks (in which order). There are two main scheduling algorithms paradigms, *off-line*, or *on-*

line scheduling. Off-line scheduling takes place before run-time and provides predictability and support for general constraints at the cost of flexibility. On-line scheduling, on the other hand, provides flexibility and dynamic run-time activities, but at the expense of less support for handling multiple constraints.

1.1.1 On-Line and Off-line Scheduling

On-line scheduling provides flexibility for partially, or non specified, activities, i.e., for run-time aperiodic and sporadic activities. Feasibility tests determine whether a given task set can be feasibly scheduled according to the rules of the particular algorithm applied. On-line scheduling allows to efficiently reclaim any spare time coming from early completions and allows to handle overload situations according to actual workload conditions. On-line schedulers are divided in two categories, dynamic priority schedulers, i.e., earliest deadline first (EDF), and fixed priority schedulers (FPS) as defined in [2]. In our work we focus on EDF scheduling.
Off-line scheduling, also called *table driven scheduling*, is capable of constructing schedules for distributed applications with complex constraints, e.g., precedence, jitter, and end-to-end deadlines. The inclusion of additional constraints into an offline scheduler is typically straightforward, e.g., by including the constraints in a feasibility test applied during schedule construction. As a result, the off-line scheduler produces a table containing task execution positions. During run-time only a table lookup is necessary to execute the schedule, resulting in very simple run-time dispatching.
This approach has been shown to be suitable for critical hard real-time systems in [3, 4]. By applying strict temporal control, critical activities can be performed in a deterministic way.

1.1.2 Event and Time Triggered Systems

Real-time systems are usually further classified as event or time-triggered, with respect to how the real-time activities are controlled.
In *event-triggered* systems, the activities happen in response to external events. The typical example of this is the sensor-actuator example: a sensor detects an external event and activates a task that reacts to this event (performs a computation), after which the task sends it's output to an actuator. This is an example of a system reacting and adjusting to an external event.
One of the main problems with event-triggered systems is that external events can cause many tasks to be activated, thus, causing overload in the system,

potentially leading to system failure. On-line scheduling is suitable for event triggered systems as it provides the ability to handle dynamic on-line events. SPRING [5] is an example of an event-triggered real-time operating system.
Time-triggered systems, on the other hand, require an a priori knowledge about all activities. In distributed time-triggered systems, each node must have the same notion of time, implying that clock synchronisation is needed. The main advantage of time-triggered systems is the predictable behavior they provide at the cost of low run-time flexibility. Time triggered systems are scheduled using off-line scheduling, which provides a time table containing task activation times, corresponding to the external events. An example of a time-triggered real-time operating system is MARS [6].
In this thesis we combine time and event triggered activities. In particular, we provide overload and efficient aperiodic task handling in time-triggered systems.

1.2 Problem Formulation

In this section we discuss some of the choices real-time system designers face. The choice of which scheduling algorithm to use partly depends on the requirements imposed on the system. One part of the system might require hard real-time guarantees and have complex constraints while another part have less stringent demands. This requires the designer to choose either an off-line or on-line scheduling algorithm, or, as in the situation above, maybe a combination of both.
Another influence on the choice of scheduling algorithm is the potential need to anticipate dynamic run-time activities, such as aperiodic or sporadic tasks and overload.
A related problem is that most OS's only provide a fixed scheduling algorithm that is tightly integrated with the kernel. This setup does not provide designers with any choice of which scheduling algorithm to use, even if the provided algorithm is not the most appropriate one.

1.2.1 Constraints

We have mentioned general constraints and that off-line scheduling provides more capability of handling them than on-line scheduling, but this actually depends on what class of constraint it is. Constraints originate from the demands of the application and impose requirements on the system. We define

two classes of constraints as:
Simple can easily be handled by on-line scheduling algorithms. Examples of such constraints are periods, start times, and deadlines.
Complex constraints, on the other hand, cause problems for on-line schedulers. However, some of them can be solved at the cost of a higher overhead. Here we give some examples of complex constraints:

Jitter causes the start or end of tasks to vary, which means that the interval between task instance invocations will vary. Some applications require the jitter between task instances to be constant or to have a small variation. In the extreme, periodic tasks can have invocations back-to-back or at the start of the first period and end of the second period.

Distribution cooperating tasks can execute on different nodes in a system, which can require synchronisation. Many real-time systems are distributed by nature, requiring synchronisation and communication between the parts. In order to provide determinism in such systems, the scheduler requires a global view of the whole distributed system.

Precedence means that a series of tasks must execute in a predefined order (also called a transaction). The basic example is the already mentioned sensor-actuator example, where the sensor measures some data, then computation take place, and finally data is sent to the actuator.

End-to-end deadline are deadlines for whole transactions of tasks, i.e., when the first task starts, a deadline is determined for the whole transaction of tasks. In the sensor-actuator example an end-to-end deadline exists for the whole transaction, from the sampling to the actuation.

1.2.2 Constraint and Task Handling

As mentioned, on-line scheduling methods provide high flexibility. However adding constraints, increases scheduling overhead [7] or requires new, specific schedulability tests which may have to be developed yet. Handling complex constraints can be very costly in terms of overhead.
Off-line systems require a priori knowledge about all system activities and events which will occur during runt-time. This information may be hard or even impossible to obtain. The lack of flexibility prevents effective handling of dynamic run-time activities, such as aperiodic or sporadic tasks and overload.

This disparity of capabilities of the on-line and off-line scheduling paradigms imposes the choice of choosing the benefits of either algorithm at the expense of the other's, between general constraints or run-time flexibility.

1.2.3 Overload

Overload is defined as a situation when there is not enough CPU time available to schedule all tasks to completion, i.e., some tasks will miss their deadlines. Overload situations are usually sudden and transient, i.e., if a system reacts to a sudden event by activating many tasks. When overload situations will occur, is very hard, if not impossible to predict.
Traditional on-line scheduling algorithms such as EDF or FPS behave poorly in overload situations, as shown in [8]. In the worst case they might even cause all tasks to miss their deadlines.
Off-line schedulers also handles overload poorly because of the lack of flexibility they provide. If eventual overload situations have to be included as a consideration when creating an off-line schedule, the result will in many cases lead to in an over-constrained system. At the same time, because of the low flexibility, the number of allowed overload activities would usually be restricted over the system lifetime.
Since real-time system can also be distributed, it is possible that overload situations occur on a set of processing nodes although the system is globally underloaded. Such situations could be resolved by migrating tasks from overloaded nodes to nodes with underload.

1.2.4 Operating Systems

As mentioned earlier, operating system functionality, e.g., scheduling algorithms, are usually fixed and integrated into the kernel (monolithic operating system kernels).
These monolithic kernel approaches involve a very close coupling between the scheduler and operating system. Implementation of the actual real-time scheduling algorithm is integrated with other kernel routines such as task switching, dispatching, and bookkeeping to form scheduling/dispatching module. Usually, changing scheduler in such a system results in redesign of both the system and application, which is often considered to costly.
This contrasts actual industrial demands: designers want to select various types of functionality without consideration of which package they come from or which operating system functionality to use. They are reluctant to abandon

trusted methods and to switch packages for the sake of an additional functional module only. Instead, there is a need to seamlessly integrate new functionality with a developed system, enabling designers to choose the best of various packages and to focus on the application and its demands.

1.3 Contribution

In this section we identify the scientific contributions of the work in this thesis.

- Reservation of bandwidth in an off-line schedule so that it later can be used by on-line server algorithms, such as the total bandwidth server (TBS) [9, 10]. The bandwidth is available in any interval over the length of the off-line schedule, which usually is equal to the least common multiple (LCM) of the periods of the tasks.

- Precise formulation of overload detection and value based rejection in the presence of off-line scheduled tasks. Overload situations are detected immediately when the offending tasks arrive, and resolved by rejection of low value tasks.

  The problem of overload detection and removal can be reduced to the well known binary knapsack problem, an we use this similarity to model the problem.

- Heuristic based overload handling algorithm, we guarantee timely execution of the off-line scheduled tasks. We need to use a heuristic solution since the binary knapsack problem is NP-hard, indicating that an optimal solution is not feasible.

- The proposed overload handling we propose includes possible task migration to benefit from the distributed system (all nodes are not necessarily overloaded). Overloaded nodes store rejected tasks in a temporary queue and under-loaded nodes can request tasks from this queue if they can schedule them.

- To disentangle the real-time scheduling from the operating system (OS) kernel routines and the actual dispatching, we propose a plug-in based scheduler architecture. Each scheduling algorithm is fitted into a plug-in module, with a common interface, and can thus be interchanged without any changes to the application or to the OS.

Designers get greater freedom of choice in deciding which algorithm to use, i.e., the one that is most appropriate to the situation. This counters the current fashion of forcing the designer to use the algorithms supplied by the real-time OS, which may not suit.

- The interface provided for the plug-in module architecture is small and intuitive. Adding new scheduling algorithms is straightforward and only requires that the design conforms to the intended behavior of the interface.

1.4 Related Work

Off-line scheduling has been extensively investigated in [11, 12] by Kopetz et al. The MARS operating system, in [4], is an off-line scheduled real-time operating system, suitable for critical hard real-time applications.
On-line scheduling has also received lot of attention in research, both EDF and FPS has been extended to handle various types of tasks and situations. EDF has also been extended to handle aperiodic tasks by using server algorithms, in [9, 10, 13] various server algorithms are presented, including the total and constant bandwidth servers. The advantage of the server approach is that the server can be treated as a "normal" task in the temporal analysis. In [2], Liu and Leyland provide analysis for a least upper bound for the processor utilisation of both EDF and FPS.

1.4.1 Combining On-Line and Off-Line Scheduling

How to introduce flexibility into off-line schedules has been studied in [14], in which a method, called slot-shifting is presented. Slot-shifting transforms an off-line schedule where tasks originally have complex constraints into a set of independent on-line tasks. As a result, each of these independent on-line tasks has an execution window in which they have to run. To keep track of available resources in a schedule slot-shifting uses intervals and spare-capacity. Slot-shifting also includes handling of hard and soft aperiodic tasks.
In [15] the slot-shifting algorithm is extended to handle sporadic tasks as well, and a new more efficient on-line guarantee test is provided.
From the on-line side, the integration of different scheduling paradigms in the same system requires a resource reservation mechanism to isolate the temporal behavior of each schedule. In [16], Mercer, Savage, and Tokuda propose a scheme based on processor capacity reserves, where a fraction of the CPU

bandwidth is reserved to each task. This approach removes the need of knowing the worst-case computation time (WCET) of each task because it fixes the maximum time that each task can execute in each period. Since the periodic scheduler is based on the Rate Monotonic algorithm, the classical schedulability analysis can be applied to guarantee hard tasks, if any present.
In [17], Liu and Deng describe a two-level scheduling hierarchy which allows hard real-time, soft real-time, and non real-time applications to coexist in the same system. According to this approach, each application is handled by a dedicated server, which can be a Constant Utilization Server [18] for tasks that do not use non-preemptable sections or global resources, and a Total Bandwidth Server [10, 9] for the other tasks. At the lowest level, all jobs coming from the different applications are handled by the EDF scheduling algorithm. Although this solution can isolate the effects of overloads at the application level, the method requires the knowledge of the WCET even for soft and non real-time tasks.
The use of information about amount and distribution of unused resources for non periodic activities is similar to the basic idea of slack stealing [19], [20] which applies to fixed priority scheduling. Our method applies this basic idea in the context of off-line and EDF scheduling. Chetto and Chetto [21] presented a method to analyze idle times of periodic tasks based on EDF. Our scheme analyzes off-line schedules, which can be more general than strictly periodic tasks, e.g., for control applications.

1.4.2 Overload Handling

Value based overload handling has been thoroughly investigated. In [22], 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 [23].
For very high overloads, scheduling based on value density outperforms EDF based methods. In [24] 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 [25], an overload algorithm is presented for the special case when a minimum slack factor for every task is known. In this work they assume that all tasks are equally important. In our solution we have two different types of

tasks, and off-line scheduled tasks are more important than on-line aperiodic tasks.
These methods do not consider distributed scheduling, or overload handling in the presence of off-line scheduled critical tasks.
Distributed overload handling is addressed in, e.g., [26], 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 thesis 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. Thus, task migration is integrated with resource reclaiming and the acceptance test of new aperiodic tasks.

1.5 Overview of Papers

1.5.1 Paper A: *Improved Handling of Soft Aperiodic Tasks in Off-Line Scheduled Real-Time Systems using Total Bandwidth Server*

Summary In this paper we propose a method to efficiently handle soft real-time tasks in off-line scheduled real-time systems using total bandwidth server. It uses off-line scheduling to handle complex constraints and reduce their complexity by transforming a constructed feasible schedule into independent tasks on single nodes with start times and deadline constraints only. These are suited for flexible earliest deadline first scheduling methods at run-time. The concrete off-line scheduler used for this work [27] allows for preemptive tasks with precedence constraints, end-to-end deadlines, distribution, network scheduling and jitter control. Furthermore, the off-line scheduler guarantees that the resulting task set guarantees the availability of a minimum bandwidth for use at run-time, extending the range of applicable methods and constraints. Via the transformed task set and bandwidth, our method provides an interface to combine off-line and on-line scheduling algorithms.
On-line scheduling is used to efficiently handle those activities that cannot be completely characterized off-line in terms of worst-case behavior, and hence cannot receive *a priori* guarantee. Examples of these activities include soft aperiodic tasks (e.g., multimedia tasks) whose computation time or inter-arrival times can have significant variation from instance to instance. Moreover, on-

line scheduling is used to reclaim any spare time coming from early completion. A bandwidth reservation technique [13] is used to isolate the temporal behavior of the two schedules and prevent the event-driven tasks to corrupt the off-line scheduled tasks. This reservation technique uses the available bandwidth guaranteed by the off-line scheduler.

1.5.2 Paper B: *Handling Aperiodic Tasks in Diverse Real-Time Systems using Plug-Ins*

Summary In this paper we propose the use of a plug-in approach to add functionality to existing scheduling schemes and provide for easy replacement at the operating system level. In particular, we present an architecture to disentangle actual real-time scheduling from dispatching and other kernel routines with a small API, suited for a variety of scheduling schemes as plug-ins. We detail two plug-ins for aperiodic task handling and how they can extend two target systems, table-driven and EDF scheduling using the presented approach. The plug-in module architecture makes it possible to hide the differences between scheduling algorithms behind a common interface.

A number of aperiodic task handling methods have been presented [28, 19, 10], but within their respective packages only. Instead of extending an existing scheduling package, we concentrate the functionality into a module, define the interface and discuss its application to off-line and on-line scheduling methods as examples.

1.5.3 Paper C: *Enhancing Time Triggered Scheduling with Value Based Overload Handling and Task Migration*

Summary In this paper we consider distributed systems where a subset of critical activities are handled in a time triggered fashion, via an off-line schedule. At run-time, 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. Overload situations are detected immediately when the offending tasks arrive, and resolved by rejection of low value tasks.

We describe how the off-line scheduling approach can be enhanced to suit distributed real-time systems where overload situations must be anticipated. 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 off-line, but the schedule is handled in a flexible way at run-time to facilitate the inclusion of aperiodic tasks. This is achieved by including mechanisms from the slot shifting algorithm [29] that allow the planned execution of off-line 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 trade-off between guaranteed timely execution, and less resource demanding non-guaranteed handling based on values.
Simulation results underline the effectiveness of the presented approach.

1.5.4 Paper D: *Simulation Results and Algorithm Details for Value Based Overload Handling*

Summary This paper contains the simulation results for a proposed algorithm, in *paper B*, for value based task rejection in the presence of offline scheduled tasks for which a timely execution have to be guaranteed. We also give a detailed description of the algorithm for computing the overload amounts.

1.6 Summary

In this thesis we present methods to aid real-time system designers with choices related to scheduling. We specifically extend the capabilities of off-line scheduling, by including aperiodic task and overload handling.
We present a method to efficiently handle aperiodic tasks in off-line scheduled real-time system by using total bandwidth server (TBS). The off-line scheduler reserves bandwidth in the off-line schedule for later use by TBS during run-time. All tasks in the off-line schedule are transformed into independent on-line tasks with start-time and deadline pairs without violating any of the original constraints. At run-time, these independent tasks are scheduled using EDF, while TBS handles the aperiodic activities using the off-line reserved bandwidth.
Furthermore, we include overload handling in time triggered real-time systems with mixed task sets consisting of both off-line and on-line tasks. Overload is

handled in such a way that the off-line scheduled tasks are still guaranteed to meet all the original timing constraints.
We give a precise formulation of overload detection and value based task rejection in the presence of the off-line scheduled tasks, and present a heuristic algorithm to handle overload.
The method includes the possibility to quickly migrate tasks from overloaded to under loaded nodes to balance the load.
We also present a plug-in approach to add functionality to existing scheduling schemes and provide for easy replacement of scheduling algorithms on the operating system level, as opposed to monolithic kernels. In particular, we present an architecture to disentangle actual real-time scheduling from other kernel routines with a small API, suited for a variety of scheduling schemes as plug-ins.

Bibliography

[1] J. A. Stankovic and K. Ramamritham. *IEEE Tutorial: Hard Real-Time Systems*. IEEE Computer Society Press, Washington, D.C., USA, 1988.

[2] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in Hard Real-Time Environment. *Journal of the ACM, 20, 1*, January 1973.

[3] H. Kopetz. Time-Triggered Model of Computation. In *In Proceeedings 19th Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[4] H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger. Distributed Fault-Tolerant Real-Time Systems: The MARS Approach. *IEEE Micro*, February 1989.

[5] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-Time Operating Systems. *IEEE Software*, May 1991.

[6] H. Kopetz, G. Fohler, G. Grünsteidl, H. Kantz, G. Pospischil, P. Puschner, J. Reisinger, R. Schlatterbeck, W. Schütz, A. Vrchoticky, and R. Zainlinger. The Distributed, Fault-Tolerant Real-Time Operating System MARS. *IEEE Operating Systems Newsletter*, 6(1), 1992.

[7] V. Yodaiken. Rough notes on Priority Inheritance. Technical report, New Mexico Institute of Mining, 1998.

[8] C. D. Locke. Best-Effort Decision Making for Real-Time Scheduling, 1986.

[9] M. Spuri and G. Buttazzo. Efficient Aperiodic Service under Earliest Deadline Scheduling. In *In Proceedings of the 15th IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, December 1994.

[10] M. Spuri and G. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. In *In Proceedings of the IEEE Real-Time Systems Symposium*, Washington D.C., USA, December 1996.

[11] H. Kopetz and G. Grunsteidl. TTP - a Protocol for Fault-Tolerant Real-Time Systems. *Computer*, 27, 1994.

[12] H. Kopetz. Why Time-Triggered Architectures will Succeed in Large Hard Real-Time Systems. In *Proceedings of the 5th IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, Chengju, Korea, August 1995.

[13] L. Davis and G. Buttazzo. Integrating Multimedia Applications in Hard Real-Time Systems. In *In Proceedings of the 19th Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[14] G. Fohler. *Flexibility in Statically Scheduled Hard Real-Time Systems*. PhD thesis, Technishe Universität, Wienna, Austria, 1994.

[15] D. Isovic. Handling Sporadic Tasks in Real-Time Systems - Combined Offline and Online Approach. Technical report, Mälardalens Högskola, Västerås, Sweden, 2001.

[16] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. In *In Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, Boston, USA, May 1994.

[17] Z. Den and J. W. S. Liu. A Scheme for Scheduling Hard Real-Time Applications in Open System Environment. In *Proceedings of the 9th Euromicro Workshop on Real-Time Systems*, Toledo, Spain, June 1997.

[18] Z. Den, J. W. S. Liu, and J. Sun. A Scheme for Scheduling Hard Real-Time Applications in Open System Environment. In *Proceedings of the 9th Euromicro Workshop on Real-Time Systems*, Toledo, Spain, June 1997.

[19] S. R. Thuel and J.P. Lehoczky. Algorithms for Scheduling Hard Aperiodic Tasks in Fixed-Priority Systems using Slack Stealing. In *In Proceedings of the 15th Real-Time Systems Symposium*, San Juan, Puerto Rico, December 1994.

[20] R.I. Davis, K.W. Tindell, and A. Burns. Scheduling Slack Time in Fixed Priority Pre-emptive Systems. In *In Proceedings of the 14th Real-Time Systems Symposium*, Raleigh-Durham, USA, December 1993.

[21] H. Chetto and M. Chetto. Some Results on the Earliest Deadline Scheduling Algorithm. *Transactions on Software Engineering*, 15, October 1989.

[22] G. Buttazzo, M. Spuri, and F. Sensini. Value vs. Deadline Scheduling in Overload Conditions. In *In Proceedings of the 16th Real-Time Systems Symposium*, Pisa, Italy, December 1995.

[23] G. Buttazzo and J. Stankovic. RED: A Robust Earliest Deadline Scheduling Algorithm. In *In Proceedings of the 3rd International Workshop on Responsive Computing Systems*, September 1993.

[24] S. A. Aldarmi and A. Burns. Dynamic Value-Density for Scheduling Real-Time Systems. In *In Proceedings of the 11th Euromicro Conference on Real-Time Systems*, York, England, June 1999.

[25] S. Baruah and J. Haritsa. Scheduling for Overload in Real-Time Systems. *IEEE Transactions on Computers*, September 1997.

[26] K. Ramamritham, J.A. Stankovic, and W. Zhao. Distributed Scheduling of Tasks with Deadlines and Resource Requirements. *IEEE Transactions on Computers*, August 1989.

[27] G. Fohler. Analyzing a Pre Run-Time Scheduling Algorithm and Precedence Graphs. Technical report, Technishe Universität, 1992.

[28] S. R. Thuel and J.P. Lehoczky. On-Line Scheduling of Hard Deadline APeriodic Tasks in Fixed-Priority Systems. In *In Proceedings of the 14th Real-Time Systems Symposium*, Raleigh-Durham, USA, December 1993.

[29] G. Fohler. Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems. In *In Proceedings of the 16th Real-Time Systems Symposium*, Pisa, Italy, December 1995.

Chapter 2

Paper A: Improved Handling of Soft Aperiodic Tasks in Offline Scheduled Real-Time Systems using Total Bandwidth Server

Tomas Lennvall, Gerhard Fohler, and Giorgio Buttazzo
In Proceedings of 8th International Conference on Emerging Technologies and Factory Automation, Nice, France, October 2001

Abstract

Real-world industrial applications impose complex constraints, such as distribution, end-to-end deadlines, and jitter control on real-time systems. Most scheduling algorithms concentrate on single or limited combinations of constraints and requirements only. Offline scheduling resolves complex constraints, but provides only very limited flexibility. Online scheduling on the other hand, supports flexibility, resource reclaiming, and overload handling, but handling constraints such as distribution or end-to-end deadline can be costly, if not intractable.

In this paper, we propose a method to efficiently handle soft real-time tasks in offline scheduled real-time systems using total bandwidth server. In a first step, the offline scheduler resolves complex constraints, reduces their complexity, and provides for guaranteed available bandwidth. The constructed schedule is translated into independent tasks on single nodes with starttimes and deadline constraints only. These are then executed using earliest deadline first, total bandwidth server scheduling at runtime.

2.1 Introduction

Real-world industrial applications impose complex constraints on real-time systems and their scheduling algorithms: Even simple control problems for industrial plants or automotive scenarios demand distribution, end-to-end deadlines, jitter control, periodic and non periodic tasks, as well as efficiency, cost effectiveness and more. In addition to these basic temporal constraints, a system has to fulfill complex application demands which cannot be expressed as generally. Control applications may require constraints on individual instance, rather than periods. Reliability demands can enforce allocation and separation patterns, or engineering practice may require relations between system activities, all of which cannot be expressed directly with basic constraints.

Furthermore, the type and number of constraints rarely remains fixed during the development of a system or product line. Rather, new constraints, often beyond periods and deadlines, are added during system construction or its maintenance.

Most scheduling algorithms presented concentrate on single or limited combinations of constraints and requirements only. Two main lines of algorithms, following the paradigms of performing scheduling *offline*, i.e., before the system runtime, or *online*, excel at general constraints or flexibility, resp. Offline scheduling is capable of constructing schedules for distributed applications with complex constraints, such as precedence, jitter, and end-to-end deadlines. As only a table lookup is necessary to execute the schedule, runtime dispatching is very simple. The prerequisite knowledge about all system activities and events may be hard or impossible to obtain. The lack of flexibility prevents handling not completely specified events.

Online scheduling overcomes this shortcoming and provides flexibility for partially or non specified activities. Feasibility tests determine whether a given task set can be feasibly scheduled according to the rules of the particular algorithm applied. Online scheduling allows to efficiently reclaim any spare time coming from early completions and allows to handle overload situations according to actual workload conditions.

These feasibility tests typically apply to a fixed set of constraints; changes in the set of constraints often require development of theory for tests. Handling constraints such as distribution or end-to-end deadline can be costly, if not intractable in the general case.

This disparity of capabilities of scheduling paradigms imposes the choice of choosing the benefits of either algorithm at the expense of the other's, between general constraints or runtime flexibility. Rather, the application under consid-

eration should dictate constraints and properties, with the choice of algorithm being only a secondary one.

In this paper, we propose a method to efficiently handle soft real-time tasks in offline scheduled real-time systems using total bandwidth server. It uses offline scheduling to handle complex constraints and reduce their complexity by transforming a constructed feasible schedule into independent tasks on single nodes with start times and deadline constraints only. These are suited for flexible earliest deadline first scheduling methods at runtime. Furthermore, the offline scheduler guarantees that the resulting task set guarantees the availability of a minimum bandwidth for use at runtime, extending the range of applicable methods and constraints. Via transformed task set and bandwidth, our method provides an interface to combine offline and online scheduling algorithms.

The concrete offline scheduler used for this work [1] allows for preemptive tasks with precedence constraints, end-to-end deadlines, distribution, network scheduling and jitter control. The transformation technique we present can be applied to a variety of other offline scheduling algorithms with similar constraints, e.g., [2]. The inclusion of additional constraints into an offline scheduler is typically straightforward, e.g., by including the constraint in a feasibility test applied during schedule construction.

Online scheduling is used to efficiently handle those activities that cannot be completely characterized offline in terms of worst-case behavior, and hence cannot receive *a priori* guarantee. Examples of these activities include soft aperiodic tasks (e.g., multimedia tasks) whose computation time or interarrival times can have significant variation from instance to instance. Moreover, online scheduling is used to reclaim any spare time coming from early completion. A bandwidth reservation technique [3] is used to isolate the temporal behavior of the two schedules and prevent the event-driven tasks to corrupt the off-line plan.

The MARS system [4] is an example of a system with entire offline planning of all activities. On the other side of the spectrum, SPRING [5] is using planning and global task migration [6] for handling a variety of constraints online. Its planning efforts are expensive; a dedicated scheduling chip is suggested. In our approach, the online scheduling is very simple as we only compute new deadlines.

The use of free resources in offline constructed schedules for aperiodic tasks has been discussed in [7]. The resulting flexibility is limited since aperiodic tasks are inserted into the idle times of the schedule only. Slot shifting [8] analysis offline schedules for unused resources and leeway, which is represented as execution intervals and spare capacities. This information is used at

runtime to shift task executions, accommodate dynamic tasks, and to perform online guarantee tests. It provides increased flexibility, but focuses on hard and firm tasks only.
From the online side, the integration of different scheduling paradigms in the same system requires a resource reservation mechanism to isolate the temporal behavior of each schedule. In [9], Mercer, Savage, and Tokuda propose a scheme based on processor capacity reserves, where a fraction of the CPU bandwidth is reserved to each task. This approach removes the need of knowing the worst-case computation time (WCET) of each task because it fixes the maximum time that each task can execute in each period. Since the periodic scheduler is based on the Rate Monotonic algorithm, the classical schedulability analysis can be applied to guarantee hard tasks, if any present.
In [10], Liu and Deng describe a two-level scheduling hierarchy which allows hard real-time, soft real-time, and non real-time applications to coexist in the same system. According to this approach, each application is handled by a dedicated server, which can be a Constant Utilization Server [11] for tasks that do not use nonpreemptable sections or global resources, and a Total Bandwidth Server [12, 13] for the other tasks. At the lowest level, all jobs coming from the different applications are handled by the EDF scheduling algorithm. Although this solution can isolate the effects of overloads at the application level, the method requires the knowledge of the WCET even for soft and non real-time tasks.
The use of information about amount and distribution of unused resources for non periodic activities is similar to the basic idea of slack stealing [14], [15] which applies to fixed priority scheduling. Our method applies this basic idea in the context of offline and EDF scheduling. Chetto and Chetto [16] presented a method to analyze idle times of periodic tasks based on EDF. Our scheme analyzes offline schedules, which can be more general than strictly periodic tasks, e.g., for control applications.
The rest of this paper is organized as follows: First, we define terminology in section 2.2. The techniques for integration of offline and online are presented in section 2.3. An example in section 2.4 illustrates the methods. We conclude the paper with section 2.6.

2.2 Terminology and assumptions

We consider a system consisting of three types of tasks: hard, soft, and non real-time tasks. Any task τ_i consists of a sequence of jobs $J_{i,j}$, where $r_{i,j}$

denotes the arrival time (or request time) of the j^{th} job of task τ_i.

A hard real-time task is characterized by two additional parameters, (C_i, T_i), where C_i is the WCET of each job and T_i is the minimum interarrival time between successive jobs, so that $r_{i,j+1} \geq r_{i,j} + T_i$. The system must provide an a priori guarantee that all jobs of a hard task must complete before a given deadline $d_{i,j}$. In our model, the absolute deadline of each hard job $J_{i,j}$ is implicitly set at the value $d_{i,j} = r_{i,j} + T_i$.

A soft real-time task is also characterized by the parameters (C_i, T_i), however the timing constraints are more relaxed. In particular, for a soft task, C_i represents the *mean* execution time of each job, whereas T_i represents the *desired* activation period between successive jobs. For each soft job $J_{i,j}$, a soft deadline is set at time $d_{i,j} = r_{i,j} + T_i$. Since mean values are used for the computation time and minimum interarrival times are not known, soft tasks cannot be guaranteed a priori. In multimedia applications, soft deadline misses may decrease the QoS, but do not cause critical system faults.

Tasks can be synchronized via *precedence constraints*, forming execution chains with end-to-end constraints. Precedence constraints and tasks, can be viewed as a directed acyclic graph. Tasks are represented as nodes, precedence constraints as edges. Tasks that have no predecessors are called *entry* tasks, tasks without successors *exit* tasks. The *period* of a precedence graph PG is the time interval separating two successive instances of PG in the schedule. *Deadline intervals* are defined for the maximum execution of each individual precedence graph. Entry tasks of instance i of a precedence graph PG become ready for scheduling at time $i \times period(PG)$. Exit tasks of instance i of a precedence graph PG have to be completed by time $i \times period(PG) + deadline(PG)$.

We consider a *distributed* system, i.e., one that consists of several *processing nodes* and *communication nodes* An *offline schedule* is a sequence of slots, i.e., some time granules, $slot_i$ $i = 0, \ldots,$ N-1, N stands for the number of slots in the schedule. For online schedules, N is typically equal to the least common multiple (lcm) of all involved periods. Communication nodes, i.e., the communication medium, are slotted and pre scheduled as well. Protocols with bounded transmission times, e.g., TDMA or token ring are applicable.

2.3 Integration

2.3.1 Rationale

The rationale of our method to provide for complex application constraints and efficient runtime flexibility is to concentrate all mechanisms to handle complex constraints in the *offline* phase, where they are transformed into *simple constraints* suitable for earliest deadline first scheduling, which is then used for *online* execution. The offline determined simple constraints serve as "interface" between offline preparations and online scheduling. Specifically, we use the offline scheduler presented in [1][1], starttime, deadline pairs as simple constraints, and EDF based Total Bandwidth server [12] and constant bandwidth server [3] as runtime algorithms. The amount of desired flexibility can be set in this step as well.

Our transformation technique can extract maximum flexibility. By tightening start time and deadlines of certain tasks, it is possible to constrain the execution of some tasks, e.g., for reasons of testing or reliability.

Our method works by reducing complexity (NP hard in the case of distributed, precedence constrained executions with end-to-end deadlines) offline by instantiating a set of independent tasks with starttimes, deadlines constraints on single processors which fulfill application constraints and guaranteed bandwidth requirements. The issues of allocation to nodes, subtask deadline assignment, fulfilling jitter requirements are resolved by the offline scheduler. This allows the use of time intensive algorithms to resolve the constraints, since they are performed offline, i.e., before the system is deployed, and flexible scheduling at runtime.

Once tasks with starttime, deadline constraints have been derived and analyzed, earliest deadline first scheduling is performed on single nodes individually at runtime; the original set of complex constraints, distribution, etc., remains hidden from online scheduling.

The resulting instance of simple constraints will not generally be optimum. Since it is performed offline, however, additional analysis can be performed, possibly resulting in another instantiation with different simple constraints. Consider a subtask deadline assignment which induces tight constraint on one node. Performance analysis may show a different, more relaxed setting to be more appropriated.

[1] This serves as example; a number of other offline scheduling algorithm can be applied, e.g., the one presented in [2].

2.3.2 Offline schedule construction and bandwidth reservation strategies

As an additional requirement, the offline scheduler has to create a schedule such that a desired fraction U_s of the processor utilization (i.e., a desired bandwidth) is reserved for online aperiodic service. This means that, if a bandwidth U_s is reserved on a node, then for any interval [t_1, t_2], there must be at least $(t_2 - t_1)U_s$ time available for aperiodic processing.

A trivial approach is to replace the worst case execution time of each task with $\frac{C}{1-U_S}$. No modifications to the scheduler are required to guarantee a bandwidth of U_S. This method, however, does not consider spare capacities in the schedule for bandwidth reservation. Response times in the resulting scheduling can thus be prohibitively long.

Our bandwidth reservation method during offline schedule construction analyzes the schedule for idle resources and their distribution. It maximizes flexibility by considering the leeway in the schedule, as per the specification constraints. In particular, the offline scheduler contains a function with tests the feasibility of the schedule constructed so far; it is extended by testing the availability of the specified bandwidth as well.

2.3.3 Transformation technique

The feasible schedule with guaranteed bandwidth is transformed into independent tasks with starttimes, deadline pairs. Our method is based on the preparations for online scheduling in slot shifting [8].

The offline scheduler allocates tasks to nodes and resolves the precedence constraints. The scheduling tables list fixed start- and end times of task executions, that are less flexible than possible. The only assignments fixed by specification are starts of first and completion of last tasks in chains with end-to-end constrains, and tasks sending or receiving inter-node messages. The points in time of execution of all other tasks may vary within the precedence order. We calculate earliest start-times and latest finish-times for all tasks per node based on this knowledge. As we want to determine the maximum leeway of task executions, we calculate the deadlines to be as late as possible.

Let $end(PG)$ denote the end and $start(PG)$ the start of a precedence graph PG according to the schedule. The start of an inter-node message transmission M is denoted $start(M)$, the time it is available at all receiving nodes $end(M)$. These are the only fixed start times and deadlines, all others are calculated recursively with respect to precedence successors.

These fixed constraints are derived first: The *deadline* of task T, d_T, of precedence graph PG in a schedule is:
If T is exit task in PG: $d_T = dl(PG)$,
if T sends an inter-node message M: $d_T = start(M)$.
The *earliest start time* of task T, r_T, of precedence graph PG in a schedule is calculated in a similar way:
If T is entry task: $r_T = start(PG)$,
if T receives an inter-node message M: $r_T = end(M)$.
Next, constraints of predecessors and successors of tasks with fixed constraints are derived:
A predecessor P of a task T with fixed deadline is assigned a deadline so as to be executed before T with EDF, i.e.,
$d_P = d_T - C_T$.
A successor S of a task T with fixed starttime is assigned the same starttime as T. An appropriate deadline and EDF with ensure P preceding T.
$r_P = r_T$.
This step is applied recursively.

2.3.4 Online scheduling

Once the transformation is performed off line and a bandwidth U_s is reserved on each processing node, on line scheduling of aperiodic tasks can be handle by a Total Bandwidth Server (TBS). This service mechanism was proposed by Spuri and Buttazzo [13, 12] to improve the response time of soft aperiodic requests in a dynamic real-time environment, where tasks are scheduled according to EDF.
The server works as follows: whenever an aperiodic request enters the system, the total bandwidth (in terms of cpu execution time) of the server, is immediately assigned to it. This is done by simply assigning a suitable deadline to the request, which is scheduled according to the EDF algorithm together with the periodic tasks in the system. The assignment of the deadline is done in such a way to preserve the schedulability of the other tasks in the system.
In particular, when the k-th aperiodic request arrives at time $t = r_k$, it receives a deadline

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k^a}{U_s},$$

where C_k^a is the execution time of the request and U_s is the server utilization factor (i.e., its bandwidth). By definition $d_0 = 0$. The request is then inserted into the ready queue of the system and scheduled by EDF, as any periodic or

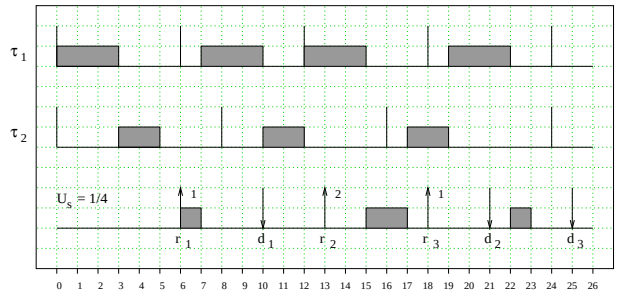


Figure 2.1: Total Bandwidth Server example

sporadic instance. Note that the maximum between r_k and d_{k-1} is needed to keep track of the bandwidth already assigned to previous requests.
Figure 2.1 shows an example of schedule produced with a TBS. The first aperiodic request, arrived at time $t = 6$, is assigned a deadline $d_1 = r_1 + \frac{C_1}{U_s} = 6 + \frac{1}{0.25} = 10$, and since d_1 is the earliest deadline in the system, the aperiodic activity is executed immediately. Similarly, the second request receives the deadline $d_2 = r_2 + \frac{C_2}{U_s} = 21$, but it is not serviced immediately, since at time $t = 13$ there is an active periodic task with a shorter deadline (18). Finally, the third aperiodic request, arrived at time $t = 18$, receives the deadline $d_3 = \max(r_3, d_2) + \frac{C_3}{U_s} = 21 + \frac{1}{0.25} = 25$ and is serviced at time $t = 22$.
Intuitively, the assignment of the deadlines is such that in each interval of time, the fraction of processor time allocated by EDF to aperiodic requests never exceeds the server utilization U_s. Since the processor utilization due to aperiodic tasks is at most U_s, the schedulability of a periodic task set in the presence of a TBS can simply be tested by verifying the following condition:

$$U_p + U_s \leq 1,$$

where U_p is the utilization factor of the periodic task set. This results is proved by the following theorem.

Theorem 1 (Spuri and Buttazzo, 96). *Given a set of n periodic tasks with processor utilization U_p and a TBS with processor utilization U_s, the whole set is schedulable* if *and* only if

$$U_p + U_s \leq 1.$$

The implementation of the TBS is straightforward, since to correctly assign the deadline to a new request, we only need to keep track of the deadline assigned

to the last aperiodic request (d_{k-1}). Then, the request can be inserted into the ready queue and treated by EDF as any other periodic instance. Hence, the overhead is practically negligible.

2.4 Example

In this section, we will illustrate our methods with an example. The system consists of two processing nodes; for the simplicity of the example, we assume that the sending of a message takes one time unit. There are 7 tasks to be offline scheduled: A, B, C, D, E, Y, Z, $A - E$ form a precedence constrained execution chain, with the following precedence constraints: $A \rightarrow B, B \rightarrow C, B \rightarrow E, C \rightarrow D, C \rightarrow E$. The global precedence graph is shown in Figure 2.2.

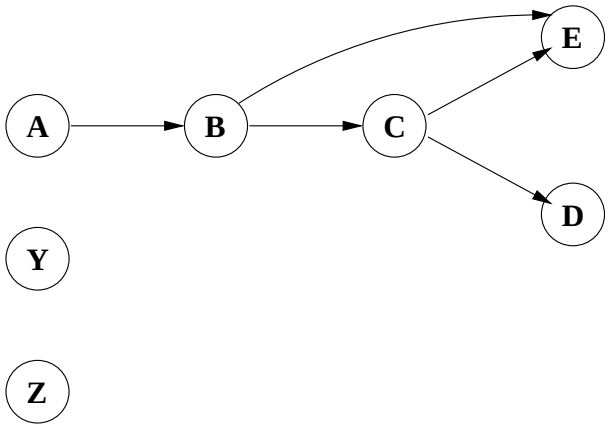


Figure 2.2: Precedence graph for the tasks of the example.

Tasks A, B, E are allocated to node 0 and C, D to 1. Task A has a jitter requirement: the variation in execution has to be less than or equal 1 for two succeeding instances. The starttime of the execution chain is 0, the end-to-end deadline, i.e., the maximum time interval between $start(A)$ and $max(end(D), end(E)$ 11. Y, allocated to node 0, has starttime of 4 and deadline 9, and Z on node 1 of 0 and 6, resp. Worst case execution times: $A = 2, B = 1, C = 1, D = 1, E = 1, Y = 2, Z = 2$. The required bandwidth $U_S = \frac{1}{3}$.

2.4.1 Transformation into simple constraints

The task schedule together with the original constraints is transformed into independent tasks on single nodes with start time, deadline pairs (denoted as r_i, d_i). First, it identifies constraints due to end-to-end, internode communication, and jitter:
End-to-end constraints:
$r_A = 0, d_E = d_D = 11$.
Internode communication: $d_B = 5, r_C = 6, d_C = 8, r_E = 9$.
Jitter: $d_A = r_A + wcet(A) - jitter = 3$.
Next, constraints for successors and predecessors of these tasks are derived:
$r_B = r_A = 0$, $r_D = r_C = 6$.
A, B cannot start before the starttime of the execution chain, thus they are assigned the same deadline. Since we use EDF for online scheduling, $d_A < d_B$ ensures the precedence order.
Y, Z are constrained by their original constraints:
$r_Y = 4, d_Y = 9; r_Z = 0, d_Z = 6$.
Table 2.3 summarizes the derived constraints.

task	r_i	d_i
A	0	3
B	0	5
C	6	8
D	6	11
E	9	11
Y	4	9
Z	0	6

Figure 2.3: Derived simple constraints.

These simple constraints comprise the original constraints as per the offline constructed schedule and guarantee a bandwidth $U_S = \frac{1}{3}$.
The resulting EDF schedule on the transformed task set is depicted in Figure 2.4.

2.4.2 Online Scheduling

The available bandwidth $U_s = 1/3$ reserved by the offline algorithm, can be exploited by a Total Bandwidth Server (TBS) to efficiently handle online ape-

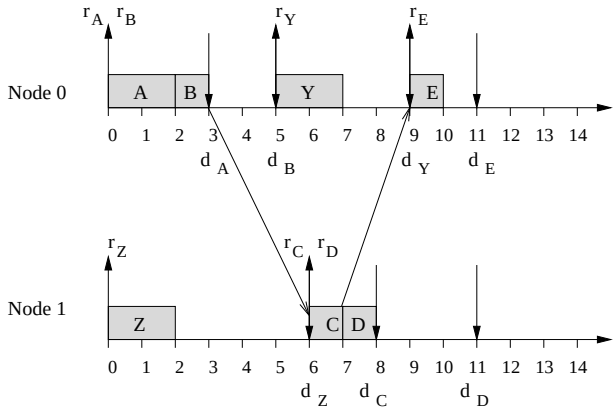


Figure 2.4: Schedule produced by EDF on the offline transformed task set.

riodic requests. In the example, a request J_1 with computation time $C_1 = 1$ arrives on Node 0 at time $t = 1$. Hence, the TBS assigns it a deadline $d_1 = t + C_1/U_s = 4$. As a consequence, J_1 is executed before task B, since $d_1 < d_B$.
At time $t = 5$, another request J_2 with computation time $C_2 = 2$ arrives on Node 0. This time, the request is assigned a deadline $d_2 = t + C_2/U_s = 11$, which is the same as the deadline of task E. However, since deadline ties are broken in favor of the server, J_2 executes before E.
On Node 1, a request J_3 with computation time $C_3 = 2$ arrives at time $t = 1$. Hence, according to the TBS rule, it is assigned a deadline $d_3 = t+C_3/U_s = 7$. At time $t = 5$, another request J_4 arrives before deadline d_3. In this case, the TBS rule states that the deadline has to be computed as $d_4 = max(t, d_3) + C_2/U_s = 10$, since the bandwidth U_s was already assigned to J_3 up to time d_3. As a result, since $d_4 < d_D$, the execution of J_4 precedes that of task D.
The integrated schedule is shown in Figure 2.5

2.5 Simulations

We implemented and simulated the described algorithm. All the offline and soft aperiodic tasks were randomly generated, and the load of the offline tasks is varied between 0 and 0.9, and the soft aperiodic task load is varied between three different levels over the lcm.

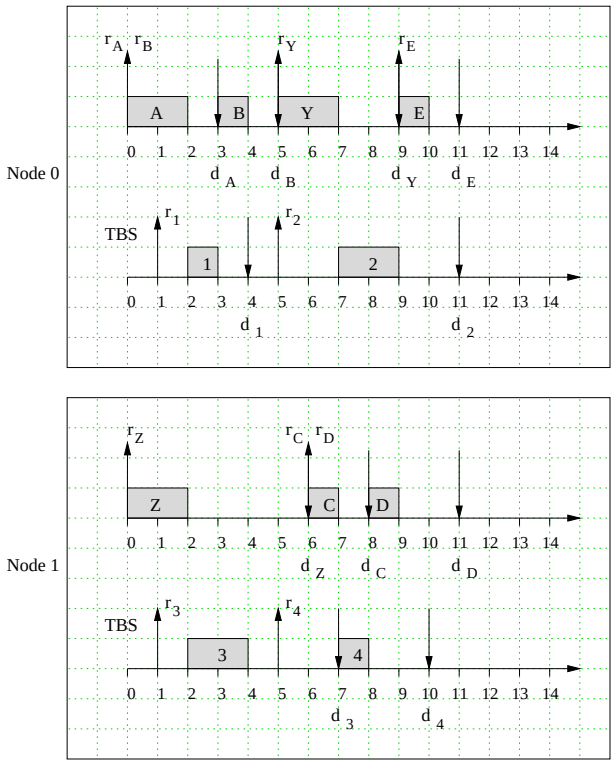


Figure 2.5: Schedule produce by EDF using the integrated approach.

The simulation length for each run was 10000 slots, and the soft aperiodic tasks arrived during that length.

We have studied the average response times of the soft aperiodic tasks, and compared our algorithm against background scheduling. We also check what happens with the soft tasks when the offline load is increased. When the offline load is equal to 0 only TBS is running.

Figure 2.6 shows the result of the simulation. The same task sets were run with both background scheduling and our method, and the response times are measured in slots.

Due to high offline load, many soft aperiodic tasks did not finish before the simulation ended and therefore the response times became higher with increased

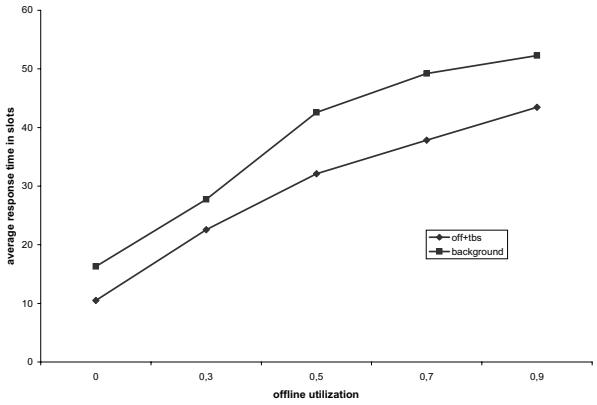


Figure 2.6: Response times for the soft aperiodic tasks.

load. Even when the offline load was 0 some soft aperiodic tasks were not able to finish because the ready queue was overloaded.

2.6 Conclusion

In this paper, we presented a method to efficiently handle soft aperiodic tasks in offline scheduled systems, to handle real-world constraints such as distribution, end-to-end deadlines, jitter control, periodic and non periodic tasks, as well as efficiency, cost effectiveness and more.
We propose using an offline scheduler to handle complex constraints, reduce their complexity, and provide for guaranteed available bandwidth. In order to apply standard earliest deadline first scheduling during system operation, we presented a technique, which transforms the schedule and original constraints into independent tasks with starttime, deadline constraints on single nodes only. Consequently, the runtime mechanisms are very simple, flexible, and efficient. We have shown how these resulting simple constraints can be used by the standard total bandwidth server algorithm for flexible and efficient execution, resource reclaiming, and overload handling.
Instead of different feasibility tests for different types of constraints, our method caters for a variety of constraints with only minor modifications in the offline scheduler. Since changes and additions to the set of constraints can be incorporated into an offline scheduler relatively easy, our method also provides for

variations and modifications in task constraints, as, e.g., induced by industrial design processes and system life cycles.

Bibliography

[1] G. Fohler. Analyzing a Pre Run-Time Scheduling Algorithm and Precedence Graphs. Technical report, Technishe Universität, 1992.

[2] K. Ramamritham. Allocation and scheduling of complex periodic tasks. In *In the 10th International Conference on Distributed Computing Systems*, Paris, France, May 1990.

[3] L. Davis and G. Buttazzo. Integrating Multimedia Applications in Hard Real-Time Systems. In *In Proceedings of the 19th Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[4] H. Kopetz, G. Fohler, G. Grünsteidl, H. Kantz, G. Pospischil, P. Puschner, J. Reisinger, R. Schlatterbeck, W. Schütz, A. Vrchoticky, and R. Zainlinger. The Distributed, Fault-Tolerant Real-Time Operating System MARS. *IEEE Operating Systems Newsletter*, 6(1), 1992.

[5] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-Time Operating Systems. *IEEE Software*, May 1991.

[6] W. Zhao, K. Ramamritham, and J. A. Stankovic. Preemptive Scheduling Under Time and Resource Constraints. *IEEE Transactions on Computers*, August 1987.

[7] K. Ramamritham, G. Fohler, and J.-M. Adan. Issues in the Static Allocation and Scheduling of Complex Periodic Tasks. In *In Proceedings of the 10th IEEE Workshop on Real-Time Operating Systems and Software*, New York, USA, May 1993.

[8] G. Fohler. Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems. In *In Proceedings of the 16th Real-Time Systems Symposium*, Pisa, Italy, December 1995.

[9] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. In *In Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, Boston, USA, May 1994.

[10] Z. Den and J. W. S. Liu. A Scheme for Scheduling Hard Real-Time Applications in Open System Environment. In *Proceedings of the 9th Euromicro Workshop on Real-Time Systems*, Toledo, Spain, June 1997.

[11] Z. Den, J. W. S. Liu, and J. Sun. A Scheme for Scheduling Hard Real-Time Applications in Open System Environment. In *Proceedings of the 9th Euromicro Workshop on Real-Time Systems*, Toledo, Spain, June 1997.

[12] M. Spuri and G. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. In *In Proceedings of the IEEE Real-Time Systems Symposium*, Washington D.C., USA, December 1996.

[13] M. Spuri and G. Buttazzo. Efficient Aperiodic Service under Earliest Deadline Scheduling. In *In Proceedings of the 15th IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, December 1994.

[14] S. R. Thuel and J.P. Lehoczky. Algorithms for Scheduling Hard Aperiodic Tasks in Fixed-Priority Systems using Slack Stealing. In *In Proceedings of the 15th Real-Time Systems Symposium*, San Juan, Puerto Rico, December 1994.

[15] R.I. Davis, K.W. Tindell, and A. Burns. Scheduling Slack Time in Fixed Priority Pre-emptive Systems. In *In Proceedings of the 14th Real-Time Systems Symposium*, Raleigh-Durham, USA, December 1993.

[16] H. Chetto and M. Chetto. Some Results on the Earliest Deadline Scheduling Algorithm. *Transactions on Software Engineering*, 15, October 1989.

Chapter 3

Paper B: Handling Aperiodic Tasks in Diverse Real-Time Systems via Plug-Ins

Tomas Lennvall, Björn Lindberg, and Gerhard Fohler
In the 5th International Symposium on Object-Oriented Real-Time Distributed Computing, Washington D.C., USA, April–May 2002

Abstract

Functionality for various services of scheduling algorithms is typically provided as extensions to a basic algorithm. Aperiodic task handling, guarantees, etc., are integrated with a specific basic scheme, such as earliest deadline first, rate monotonic, or off-line scheduling. Thus, scheduling services come in packages of scheduling schemes, fixed to a certain methodology.
A similar approach dominates operating system functionality: implementation of the actual real-time scheduling algorithm, i.e., take the decisions which task to execute at which times to ensure deadlines are met, are intertwined with kernel routines such as task switching, dispatching, and bookkeeping to form a scheduling/dispatching module.
Consequently, designers have to choose a single scheduling package, although the desired functionality may be spread over several ones. Instead, there is a need to seamlessly integrate new functionality with a developed system, enabling designers to choose the best of various packages.
In this paper, we propose the use of a plug-in approach to add functionality to existing scheduling schemes and provide for easy replacement on the operating system level. In particular, we present an architecture to disentangle actual real-time scheduling from dispatching and other kernel routines with a small API, suited for a variety of scheduling schemes as plug-ins. We detail two plug-ins for aperiodic task handling and how they can extend two target systems, table-driven and earliest deadline first scheduling using the presented approach.

3.1 Introduction

Scheduling algorithms have been typically developed around central paradigms, such as earliest deadline first (EDF) [1], rate monotonic (RM)[1], or off-line scheduling. Additional functionality, such as aperiodic task handling, guarantees, etc., is typically provided as extensions to a basic algorithm. Over time, scheduling packages evolved, providing a sets of functionality centered around a certain scheduling methodology.

EDF or fixed priority scheduling (FPS), for example, are chosen for simple dispatching and flexibility. Adding constraints, however, increases scheduling overhead [2] or requires new, specific schedulability tests which may have to be developed yet. Off-line scheduling methods can accommodate many specific constraints and include new ones by adding functions, but at the expense of runtime flexibility, in particular inability to handle aperiodic and sporadic tasks.

A similar approach dominates operating system functionality: implementation of the actual real-time scheduling algorithm, i.e., take the decisions which task to execute at which times to ensure deadlines are met, are intertwined with kernel routines such as task switching, dispatching, and bookkeeping to form a scheduling/dispatching module. Additional real-time scheduling functionality is added by including or "patching" this module. Replacement or addition of only parts is a tedious, error prone process.

Consequently, a designer given an application composed of mixed tasks and constraints has to choose which constraints to focus on in the selection of scheduling algorithm; others have to be accommodated as good as possible. Along with the choice of algorithm, operating system modules are chosen early on in the design process.

This contrasts actual industrial demands: designers want to select various types of functionality without consideration of which package they come from. They are reluctant to abandon trusted methods and to switch packages for the sake of an additional functional module only. Instead, there is a need to seamlessly integrate new functionality with a developed system, enabling designers to choose the best of various packages.

In this paper, we propose the use of a plug-in approach to add functionality to existing scheduling schemes and provide for easy replacement at the operating system level. In particular, we present an architecture to disentangle actual real-time scheduling from dispatching and other kernel routines with a small API, suited for a variety of scheduling schemes as plug-ins. We detail two plug-ins for aperiodic task handling and how they can extend two target systems, table-driven and EDF scheduling using the presented approach.

A number of aperiodic task handling methods have been presented [3, 4, 5], but within their respective packages only. Instead of extending an existing scheduling package, we concentrate the functionality into a module, define the interface and discuss its application to off-line and on-line scheduling methods as examples. S.Ha.R.K [6] is an operating system where scheduling algorithms including aperiodic servers are created in a modular fashion. The interface between the system and the scheduler in S.Ha.R.K is more complex than the interface we propose in this paper.
The rest of the paper is organized as follows: in section 3.2, we describe our notion of plug-in and target system, its diversity is described in section 3.3, followed by a description of the aperiodic task handling functionality in section 3.4. In section 2.4 we show an example and section 3.6 concludes the paper.

3.2 System and Plug-In Architecture

A *plug-in* can be thought of as a hardware or software module that adds a specific feature or service to an existing system. The purpose of a plug-in is to add functionality without calling for redesign or extensive modifications. To accomplish this it must be clear what services the plug-in provides and an interface between the plug-in and the target system must be defined.

3.2.1 Target System Architecture and Interface

Before we go into the details of the plug-in, we define the target system model that the plug-in will interact with. The model is presented in figure 3.1 and it consists of three separate modules as parts of the system:

Execution Sequence Table This is the table where the tasks are kept sorted in a certain order, depending on the plug-in module's scheduling algorithm. The plug-in module has exclusive modification rights on this table. To manipulate the table, the plug-in module uses the two methods *insert(task, pos)* and *remove(task)*.

Dispatcher It is responsible for taking the first task in the execution sequence table and execute it. The dispatcher has access to view the contents of the whole execution sequence table, but it cannot modify it. The plug-in module also has exclusive control over the dispatcher and it is activated by the *dispatch()* call. When the dispatcher is activated, it will check if

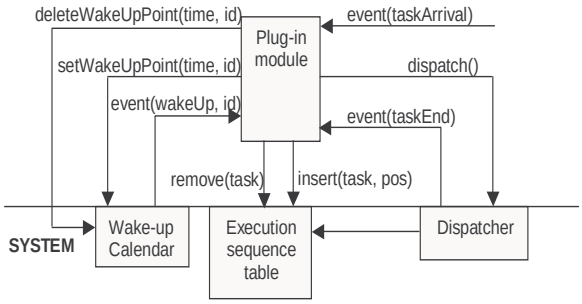


Figure 3.1: Plug-in and system architecture

there is an executing task and either preempt the task, if it exists in the execution sequence table, or else abort it.

Wake-up Calendar This calendar controls a set of watch-dog timers, all tasks will get entries set in the calendar corresponding to their deadlines (to catch deadline misses). The calendar will also hold other time critical points, such as the critical slots from [7]. To set or remove these wake-up points the plug-in module uses the *setWakeUpPoint(time, id)* and the *deleteWakeUpPoint(time)* methods. All the wake-up points are associated with an id. The id's represents deadlines, critical slots, and so on.

3.2.2 Plug-In Interface

The plug-in module encapsulates a scheduling algorithm for scheduling of user level tasks (not system level tasks), such that the rest of the system becomes completely decoupled from the scheduling. This means that the plug-in module is the only part of the system that knows about scheduling, and it is also the only part that needs to be changed, if the scheduling algorithm is being changed.
Therefore the interface to the plug-in module is kept small and simple such that it is clear how to write a new plug-in. This makes it easier for designers to create the scheduling package they want. The plug-in interface is used by the system, specifically the wake-up calendar and dispatcher, to activate the plug-in module at certain events or times. Thus each plug-in module that is implemented, is responsible for reacting correctly to the events that activates it.

The details of the plug-in module interface and the events it must react to follow below:

event(taskArrival) This event activates the plug-in when a new user level task has been activated. The plug-in is responsible for executing the appropriate acceptance test to either accept or reject the new task. If the task is accepted, the plug-in must insert it at the correct position in the execution sequence table and activate the dispatcher.

event(wakeUp, id) This event is sent by the wake-up calendar and it activates the plug-in at a certain point of time earlier set by the plug-in itself. Here, the plug-in must check what the wake-up activation corresponds to, by looking at the id, and take the appropriate actions.

event(taskEnd) The dispatcher sends this event to the plug-in when a task has finished its execution. The dispatcher does not care if the task is periodic (and should be reactivated later) or aperiodic, it's the job of plug-in module to make the correct decision based on this. Here, the plug-in should remove the task from the execution sequence table and activate the dispatcher.

3.2.3 System and Plug-In Interaction

In figure 3.1, we can see the interface the system and the plug-in uses to interact with with each other. In this section we will describe in more detail how this interaction works for some of the events that can happen during system execution.

Task arrival when a new user task is activated, *event(taskArrival)* is called to activate the plug-in module. The module executes its acceptance test to either accept or reject the task. If the task is accepted, the plug-in calls *setWakeUpPoint(dl, id)* to set a watchdog on the deadline of the task. Then, the task is inserted into the execution sequence table, using *insert(task, pos)* to set it at the correct position according to the scheduling algorithm. Finally the plug-in activates the dispatcher, by calling *dispatch()*, and then it suspends itself. The dispatcher is activated, looks at the front of the execution sequence table, picks that task for execution and then it suspends.

Task finishing execution when a task has finished its execution in a timely manner, the dispatcher gets activated and activates the plug-in module

by calling *event(taskEnd)*, then the dispatcher suspends. The plug-in removes the wake-up time for the task deadline with *removeWakeUp-Point(dl, id)*, then it removes the task from the execution sequence table by *remove(task)*. The plug-in also calls *dispatch()* again to activate the dispatcher. The dispatcher looks at the front of the execution sequence table and picks that task for execution, then it suspends.

Task deadline miss if a task has not finished execution before its deadline, the wake-up calendar will be activated by a timer interrupt. It will then use *event(wake-up, id)* to activate the plug-in module. The plug-in module sees that the *id* indicates a deadline miss and removes the task from the execution sequence table, and, if necessary takes other actions to handle a deadline miss. Then the plug-in calls the dispatcher, using *dispatch()*, to activate it. The dispatcher checks if the executing task exist in the execution sequence table. When it discovers that the task has been removed by the plug-in it will abort the task. The dispatcher also checks for the first task in the execution sequence table, picks it for execution, and suspends itself.

3.3 Target System Diversity and Plug-In Applicability

The plug-in module design makes it possible to hide the differences between scheduling algorithms behind a common interface. We will discuss how this architecture would be applied to the different scheduling paradigms that exist, and detail what the functions in the interface would do. Figure 3.2 shows the plug-ins.

3.3.1 Earliest deadline scheduled system

In an event-triggered system using the EDF scheduling algorithm, the tasks are characterized by start times, worst case execution time (WCET), and deadlines. The tasks can also be either periodic, and have the period as an additional attribute, or aperiodic. Before the start of the system, the plug-in sorts any existing tasks in the execution sequence table in EDF order. It also sets the wake-up events for the deadlines of the tasks in the wake-up calendar.
When the system is started, the plug-in activates the dispatcher and suspends itself. The dispatcher does it's job and suspends. If no new task arrives, the ex-

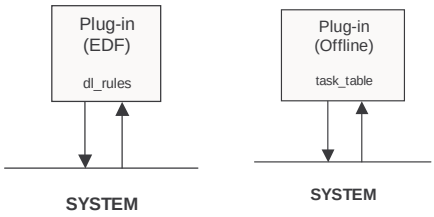


Figure 3.2: Example plug-ins

ecuting task will continue until it finishes its execution and then the dispatcher will activate the plug-in module again. The plug-in will see that it has been activated by a task-end event and remove that task from the execution sequence table. Then it activates the dispatcher again. This is how the plug-in and the system would interact if no new tasks would arrive or no deadline misses would occur.

If a new task arrives, the plug-in is activated and executes the acceptance test. If the task is accepted, it will be inserted into the execution sequence table at the correct position. The plug-in then activates the dispatcher and suspends, and the interaction continues as normal.

If a deadline miss occurs and activates the plug-in, the task will be removed from the execution sequence table. The plug-in then activates the dispatcher and the execution continues.

3.3.2 Off-line scheduled system

A target system using an off-line generated [8] schedule usually has more stringent task requirements, such as precedence constraints, than an on-line scheduled, event-triggered counterpart. In an off-line generated schedule, tasks have fixed starting and finishing times. In off-line scheduled systems there are only off-line scheduled task and no new task will dynamically arrive during the runtime of the system.

Before the execution of the system, the plug-in prepares the execution sequence table to correspond to the task table internally stored in the plug-in. The on-line execution of this plug-in will therefore be simpler with an EDF plug-in module. As with the EDF plug-in, wake-up points will also be set for the deadline of the tasks in the off-line schedule. The plug-in also sets wake-up points for every

time slot, like the MARS system described in [9].
When the system is activated, the plug-in immediately sets a wake-up point at the next time-slot. If no task has a start time equal to the current time, it suspends. The plug-in will be activated at the start of the next time-slot and repeat what it did in the previous time-slot.
If there is a task with the start time equal to the current time, the plug-in activates the dispatcher, then it suspend. The dispatcher activates the execution of the next task and suspends.
The plug-in will be activated every slot, and it will also get events when tasks end or if tasks miss their deadline. If a task finishes execution in a timely manner, the dispatcher activates the plug-in, which removes the task from the execution sequence table and then checks if there is a task ready.

3.4 Plug-Ins for Aperiodic Task Handling

Below we present two plug-ins that handles aperiodic tasks. These plug-ins are meant to be "plugged into" a scheduling module that makes scheduling decisions based on earliest start times and deadlines. The plug-ins work independently of the scheduling module and can be seen as a layer on top of it.
At all times, the scheduling module schedules task that are ready to execute, that is, tasks that are present in the ready-queue. The plug-in deals with the aperiodic tasks and places them in the ready-queue of the scheduling module, which then processes the aperiodic tasks as it would any other tasks in the system.
The mechanism for the two plug-ins for aperiodic task handling is based on the slot shifting [10], taking advantage of resources not needed by non-aperiodic tasks and using them to schedule aperiodic tasks.
We have named the different plug-ins, plug-in A and plug-in B to distinguish between the two different algorithms. Plug-in A focuses on guarantees and handling of single aperiodic tasks with fixed demands, e.g., execution time, while plug-in B is geared towards large number of aperiodic tasks with changing requirements.
Aperiodic tasks have *unknown* arrival times. The earliest start time of an aperiodic task is equal to its arrival time. Aperiodic tasks are considered independent. We assume that task dependencies are resolved in the off-line phase.

Known WCET Aperiodic tasks with known worst case times and deadlines

are termed *firm* aperiodic. If accepted, which is determined by a guarantee test, these tasks must be completed before their deadlines.

Unknown WCET Aperiodic tasks without deadlines and possibly without known maximum execution times are termed *soft* aperiodic. These are executed in a best effort fashion at lower priority than guaranteed tasks such that the timely execution of guaranteed tasks is not impaired.

3.4.1 Off-line Preparations - Slot Shifting

We propose to use the off-line transformation and on-line management of the slot shifting method [10]. Due to space limitations, we cannot give a full description here, but confine to salient features relevant to our new algorithms. More detailed descriptions can be found in [11], [10], [7]. It uses standard off-line schedulers, e.g., [8], [11] to create schedules which are then analyzed to define start-times and deadlines of tasks.
After off-line scheduling, and calculation of start-times and deadlines, the deadlines of tasks are sorted for each node. The schedule is divided into a set of *disjoint execution intervals* for each node. *Spare capacities (sc)* to represent the amount of available resources are defined for these intervals.
Each deadline calculated for a task defines the end of an interval I_i, $end(I_i)$. Several tasks with the same deadline constitute one interval. Note that these intervals differ from execution windows, i.e. start times and deadline: execution windows can overlap, intervals with *sc* are disjoint. The deadline of an interval is identical to that of the task. The start, however, is defined as the maximum of the end of the previous interval or the earliest start time of the task. The end of the previous interval may be later than the earliest start time. Thus it is possible that a task executes outside its interval, i.e., earlier than the interval start, but not before its earliest start-time.
The *sc* of an interval I_i are calculated as given in formula 3.1:

$$sc(I_i) \;=\; |I_i| - \sum_{T \in I_i} wcet(T) + min(sc(I_{i+1}), 0) \tag{3.1}$$

The length of I_i, minus the sum of the activities assigned to it, is the amount of idle time in that interval. These have to be decreased by the amount "lent" to subsequent intervals: Tasks may execute in intervals prior to the one they are assigned to. Then they "borrow" spare capacity from the "earlier" interval.
Obviously, the amount of unused resources in an interval cannot be less than zero, and for most computational purposes, e.g., summing available resources

up to a deadline are they considered zero, as detailed in later sections. We use negative values in the spare capacity variables to increase runtime efficiency and flexibility. In order to reclaim resources of a task which executes less than planned, or not at all, we only need to update the affected intervals with increments and decrements, instead of a full recalculation. Which intervals to update is derived from the negative spare capacities. The reader is referred to [11] for details.
Thus, we can represent the information about amount and distribution of free resources in the system, plus online constraints of the off-line tasks with an array of four numbers per task. The runtime mechanisms of the first version of slot shifting added tasks by modifying this data structure, creating new intervals, which is not suitable for frequent changes as required by sporadic tasks. The method described in this paper only modifies spare capacity.

3.4.2 Online Activities

Runtime scheduling is performed locally for each node. If the spare capacities of the current interval $sc(I_c) > 0$, EDF is applied on the set of ready tasks. $sc(I_c) = 0$ indicates that a guaranteed task has to be executed or else a deadline violation in the task set will occur. It will execute immediately. Since the amount of time spent is known and represented in *sc*, guarantee algorithms include this information.
After each scheduling decision, the spare capacities of the affected intervals are updated. If, in the current interval I_c, an aperiodic task executes, or the CPU remains idle for one slot, current spare capacity in I_c is decreased. If an off-line task assigned to I_c executes, spare capacity does not change. If an off-line task T assigned to a later interval $I_j, j > c$ executes, the spare capacity of I_j is increased - T was supposed to execute there but does not, and that of I_c decreased. If I_j "borrowed" spare capacity, the "lending" interval(s) will be updated. This mechanism ensure that negative spare capacity turns zero or positive at runtime. Current spare capacity is reduced either by aperiodic tasks or idle execution and will eventually become 0, indicating a guaranteed task has to be executed. See [10] for more details.

Guarantee Algorithm A

Assume that an aperiodic task T_a is tested for guarantee. We identify three parts of the total spare capacities available:

- $sc(I_c)_t$, the remaining sc of the current interval

- $\sum sc(I_i)$, $c < i \leq l$, $end(I_l) \leq dl(T_A) \wedge end(I_{l+1}) > dl(T_A)$, $sc(I_i) > 0$, the positive spare capacities of all *full* intervals between t and $dl(T_A)$
- $min(sc(I_{l+1}), dl(T_A) - start(I_{l+1}))$, the spare capacity of the last interval, or the execution need of T_A before its deadline in this interval, whichever is smaller

If the sum of all three is larger than $wcet(T_A)$, T_A can be accommodated, and therefore guaranteed. Upon guarantee of a task, the spare capacities are updated to reflect the decrease in available resources. Taking into account that the resources for T_A are not available for other tasks. This guarantee algorithm is $O(N)$, N being the number of intervals.

Guarantee Algorithm B

This plug-in uses a newer version of slot shifting as guarantee test and the basic idea behind it is based on the standard EDF guarantee. EDF is based on having full availability of the CPU, so we have to consider interference from the non-aperiodic tasks in S and pertain their feasibility.
Assume that at time t_1 we have a set of guaranteed aperiodic tasks G_{t_1} and a set of non-aperiodic tasks S. At time t_2 where $t_1 < t_2$, a new aperiodic A arrives to the plug-in module. Meanwhile, a number of tasks of G_{t_1} may have executed; the remaining task set at t_2 is denoted G_{t_2}. We test if $A \cup G_{t_2}$ can be accepted, considering tasks in S. If so, we add A to the set of guaranteed aperiodic tasks, G.
The finishing time of a firm aperiodic task A_i, with an execution demand of $c(A_i)$, is calculated with respect to the finishing time of the previous task, A_{i-1}. Without any off-line tasks, it is calculated the same way as in the EDF algorithm:

$$ft(A_i) = ft(A - i - 1) + c(A_i) \tag{3.2}$$

Since we guarantee firm aperiodic tasks together with tasks in S, we extend the formula above with a new term that reflects the amount of resources reserved for these tasks:

$$ft(A_i) = c(A_i) + \begin{cases} t + \text{R}[t, ft(A_1)] & , i = 1 \\ ft(A_{i-1}) + \text{R}[ft(A_{i-1}), ft(A_i)] & , i > 1 \end{cases} \tag{3.3}$$

where $R[t1, t2]$ stands for the amount of resources (in slots) reserved for the execution of the tasks in S between time t_1 and time t_2 . We can access $R[t1, t2]$

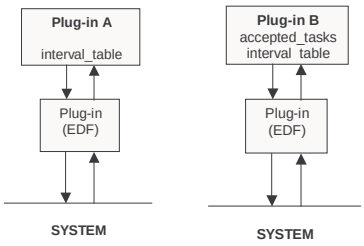


Figure 3.3: Plug-in A and Plug-in B

via spare capacities and intervals at runtime:

$$R[t_1, t_2] = [t_2 - t_1] - \sum_{I_c \in (t_1, t_2)} max(sc(I_c), 0) \tag{3.4}$$

As $ft(A_i)$ appears on both sides of the equation, a simple solution is not possible. But in [12] an algorithm, with a complexity of $O(N)$, for computing the finishing times of hard aperiodic tasks is presented.
In this plug-in module no explicit reservation of resources is done, which would require changes in the intervals and spare capacities, as done in the plug-in A module. Rather, resources are guaranteed by accepting the task only if it can be accepted together with the previous tasks in G and S. This enables the efficient use of rejection strategies, and simplifies the handling of the intervals and *sc*.

3.4.3 Guarantee Plug-Ins

When a plug-in is activated, it updates the intervals in conformity with the last task execution and checks if there are any pending aperiodic tasks. If so, it processes them and puts one or more of them into the ready-queue of the scheduler. Figure 3.3 show the two plug-ins and the data structures they contain.

Plug-In A

The plug-in keeps a table consisting of the intervals and their attributes (start, end, sc, and so on) that was created in the off-line phase. It must also keep track of which task executed last, when it started its latest execution, and how much

time it consumed, to be able to update the intervals table. Using this information, the plug-in updates interval spare capacities and possibly also wake-up points.

Plug-in B

Plug-in B also needs information about the last task execution to be able to update spare capacities and wake-up points in the intervals table it keeps locally. It focuses on handling large numbers of aperiodic tasks with changing requirements, therefore accepting tasks is done with explicit guarantees via modifying intervals and spare capacities. Rather, guarantees are including implicitly, by keeping a list of the so far accepted task. Should a task finish early, it is removed from the list and the resources reserved for it are freed without further provisions. It is well suited for efficient overload handling, since task removals do not require changes in intervals and spare capacities as in plug-in A.
After each scheduling decision, the spare capacities of the affected intervals are updated as for plug-in A.

3.5 Example

In this section we will use an example to illustrate how the two plug-in modules we defined earlier, plug-in A and plug-in B, work and interact with the rest of the system. We assume that there are three periodic tasks scheduled by the EDF algorithm, and the task-set is the following: $A = (1, 4)$, $B = (1, 6)$, $C = (2, 12)$, where (C, T) represents WCET and period. Deadline is assumed to be equal to the end of the period ($D = T$). The tasks have harmonic periods to make the example simple. Firm aperiodic tasks have the format: $Ta_f = (C, D)$, and soft aperiodic tasks have the following format: $Ta_s = (C)$.
Off-line In the off-line phase the plug-ins create a table that contains all the interval start and end points, the length of the interval, the *sc* and total execution time in an interval, and lastly the wake-up (wu) point of the interval. This table is stored within the plug-in and it will be updated during runtime to reflect the correct state. Both plug-ins create identical tables as shown in table 3.1. The table is created with a length equal to the least common multiple (LCM) of the periods of the tasks. This table will be restored and repeated when time t is equal to a multiple of the LCM.
The execution sequence table (ES-table) contains the following periodic tasks from the start: ES-table$= \{A_0, B_0, C_0\}$.

Interval	*start(I)*	*end(I)*	$\|I\|$	*sc(I)*	*wu(I)*
I_0	0	4	4	3	3
I_1	4	6	2	1	5
I_2	6	8	2	1	7
I_3	8	12	4	0	8

Table 3.1: The original interval table.

On-line The on-line behavior of the two models differs so we will show step by step how each of them behave, and what happens with the interval table at different times. Below we will see the actions taken during each step by the system and the plug-ins.

Time	System actions	Plug-in actions

This shows how the actions by the different parts will be represented. At each point time we can see the system's dispatcher, wake-up calendar actions, and the plug-in's actions.

$t = 0$	dispatch A_0	*setWakeUpPoint(3), dispatch()*

No new aperiodic tasks have arrived so the plug-in sets a wake-up point and suspends.

$t = 1$	dispatch Ta_f	*remove(A_0),* Guarantee-test, *deleteWakeUpPoint(3,critical-slot), setWakeUpPoint(4), insert(Ta_f,dl-pos), dispatch()*

ES-table= $\{B_0, C_0\}$ and a firm aperiodic task has arrived, $Ta_f = (1, 4)$.
Plug-in A The absolute deadline of Ta_f is 5, so $I_c = I_0$ and $I_f = I_1$ and the available sc in this interval is 4 ($sc(I_c) + sc(I_f)$), which is larger than Ta_f execution requirement, so Ta_f will be guaranteed. Since Ta_f 's deadline, 5, is not equal to $end(I_f)$, I_1 will have to be split. The *sc* is also updated after the split and the interval table for plug-in A is shown in table 3.2.
Plug-in B In this plug-in the set of guaranteed aperiodic tasks (G) is empty. The plug-in tests if Ta_f can be accepted together with the periodic tasks. This is done by calculating the finishing time of Ta_f, which is 2 in this case (according

Interval	*start(I)*	*end(I)*	$\|I\|$	*sc(I)*	*wu(I)*
I_0	0	4	4	3	3
I_{1a}	4	5	1	0	4
I_{1b}	5	6	1	0	5
I_2	6	8	2	1	7
I_3	8	12	4	0	8

Table 3.2: Updated interval table for plug-in A.

to formula 3.3). No interval split will occur in this plug-in, nor any change to the *sc* of the intervals table because an aperiodic task was accepted.
Both plug-ins will set an updated wake-up point. The wake-up point has been changed because task A_0 has executed one slot, and then suspend.

$t = 2$	*event(taskEnd)*, dispatch B_0	*remove(Ta_f)*, Internal-work, *dispatch()*

ES-table= $\{B_0, C_0\}$. No new aperiodic tasks has arrived, Ta_f has finished. The plug-ins will be activated by this task-end event, *plug-in A* will modify the wake up point of the interval Ta_f belonged to in the intervals table, $wu(I_1a = 5)$, and then suspend again. *Plug-in B* takes no action and suspends.

$t = 3$	*event(taskEnd)*, dispatch C_0	*remove(B_0)*, Internal-work, *dispatch()*

ES-table= $\{C_0\}$. No new aperiodic tasks have arrived. C_0 will execute. B_0 has finished, the wake up point is not modified because B_0 belongs to a later interval (but the wu in that interval is modified, so $wu(I_1) = 6$).

$t = 4$	*event(wakeUp)*, dispatch Ta_s	*insert(Ta_s,first-pos)*, *insert(A_1,pos)*, *setWakeUpPoint(5)*, *setWakeUpPoint(6)*, *dispatch()*

ES-table= $\{A_1, C_0\}$. Next instance of task A is ready. C_0 has finished executing and it belongs to a later interval, so the wu of that interval is modified ($wu(I_3) = 9$).

A soft aperiodic task $Ta_s = (4)$ has arrived. Both plug-ins will behave in the same manner: since $sc(I_c) > 0$, task Ta_s will be inserted first in the ready-queue. *Plug-in B* will set the next wake up point and suspend. *Plug-in A* will set the wake up to 5 even though the original $wu(I_c) = 4$, this has changed because Ta_f executed in an earlier interval and thus the $sc(I_c)$ increased to 1.

$t = 5$	*event(wakeUp)*, dispatch Ta_s	*setWakeUpPoint(6), dispatch()*

ES-table= $\{A_1, C_0\}$. Plug-in A is activated by the wake-up point event. Normally this means that the execution of the soft task must be stopped in favor of a periodic task. But in this case we have only an interval change, and the $sc(I_c) > 0$, so the soft task can continue to execute ($sc(I_c) > 0$ because B_0 executed in an earlier interval). *Plug-in A* resets the wake up point and suspends itself. *Plug-in B* is not activated.

$t = 6$	*event(wakeUp)*, dispatch Ta_s	*insert(B_1,EDF-pos), setWakeUp-Point(7), dispatch()*

ES-table= $\{A_1, B_1, C_0\}$. The second instance of task B is activated. Both plug-ins are activated by wake up points, this means that the execution of the soft task must be stopped in favor of a periodic task. Once again, there is only an interval change and a new wake up point can be set, and since the $sc(I_c) > 0$, Ta_s can continue to execute. Both the plug-ins suspend.

$t = 7$	*event(wakeUp)*, dispatch A_1	*remove(Ta_s), setWakeUpPoint(8), dispatch()*

ES-table= $\{A_1, B_1, C_0\}$. The plug-ins are activated due to the wake up point. Ta_s must be interrupted so A_1 won't miss it's deadline. The plug-ins set the next wake up point and suspend.

$t = 8$	*event(wakeUp)*, *event(taskEnd)*, dispatch Ta_s	*remove(A_1), insert(A_2,pos), insert(Ta_s,first-pos), setWakeUp-Point(9), dispatch()*

ES-table= $\{A_2, B_1, C_0\}$. The next instance of task A is activated. Since the $sc(I_c) > 0$, Ta_s will be put first in the ready queue and executed. The plug-ins set the next wake up point and suspend.

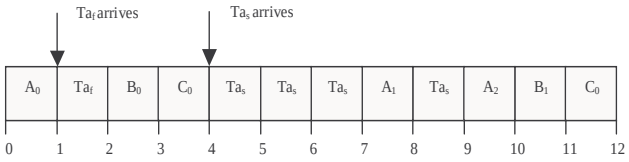


Table 3.3: Example execution trace

$t = 9$	*event(wakeUp),* *event(taskEnd),* dispatch A_2	*remove(Ta_s),* *setWakeUpPoint(10), dispatch()*

ES-table= $\{A_2, B_1, C_0\}$. Ta_s has finished executing, the plug-ins set the next wake up point and suspend.

$t = 10$	*event(wakeUp),* *event(taskEnd),* dispatch B_1	*remove(A_2),* *setWakeUpPoint(11), dispatch()*

ES-table= $\{B_1, C_0\}$. A_2 has finished it's execution, B_1 is executed. The plug-ins set the next wake-up point and suspend.

$t = 11$	*event(wakeUp),* *event(taskEnd),* dispatch C_0	*remove(B_1),* *setWakeUpPoint(12), dispatch()*

ES-table= $\{C_0\}$. B_1 has finished executing, the plug-ins set the next wake up point and suspend.

After this, because $t =$ total length of the interval tables, the plug-ins recreate the original intervals table by restoring the *sc* and *wu* of the intervals. If an aperiodic task arrives and has a deadline longer than the end of the interval table, the table will be extended by repeatedly adding the original table to the end of the extended table, until it is longer than the deadline. All the interval information (start, end, sc, and so on) of the extended table is adjusted to represent a larger table, and thus later time points.

3.6 Conclusion

In this paper we addressed the need for adding functionality to systems, in particular scheduling algorithms, without need for abandoning trusted methods or major revisions.
We proposed a plug-in approach for aperiodic task handling, presented two different plug-in modules, and showed their applicability to two scheduling schemes, EDF, and off-line scheduling. Our method concentrates the aperiodic task functionality into a software module with a defined interface.
We presented an architecture to disentangle actual real-time scheduling from dispatching and other kernel routines with a small API, suited for a variety of scheduling schemes as plug-ins. As the functionality of the plug-in is independent of the basic scheduling scheme and the interface is very small, we can insert and apply the aperiodic-plug-ins to both off-line and on-line scheduling methods.
Further research will go into extending the applicability to a wider range of systems and algorithms.

Bibliography

[1] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in Hard Real-Time Environment. *Journal of the ACM, 20, 1*, January 1973.

[2] V. Yodaiken. Rough notes on Priority Inheritance. Technical report, New Mexico Institute of Mining, 1998.

[3] S. R. Thuel and J.P. Lehoczky. On-Line Scheduling of Hard Deadline APeriodic Tasks in Fixed-Priority Systems. In *In Proceedings of the 14th Real-Time Systems Symposium*, Raleigh-Durham, USA, December 1993.

[4] S. R. Thuel and J.P. Lehoczky. Algorithms for Scheduling Hard Aperiodic Tasks in Fixed-Priority Systems using Slack Stealing. In *In Proceedings of the 15th Real-Time Systems Symposium*, San Juan, Puerto Rico, December 1994.

[5] M. Spuri and G. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. In *In Proceedings of the IEEE Real-Time Systems Symposium*, Washington D.C., USA, December 1996.

[6] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A New Kernel Approach for Modular Real-Time Systems Development. In *Proceedings of the 13th Euromicro Real-Time Systems Conference*, Delft, Netherlands, June 2001.

[7] D. Isovic and G. Fohler. Handling Sporadic Tasks in Statically Scheduled Distributed Real-Time Systems. In *Proceedings of the 11th Euromicro Real-Time Systems Conference*, York, England, June 1999.

[8] K. Ramamritham. Allocation and scheduling of complex periodic tasks. In *In the 10th International Conference on Distributed Computing Systems*, Paris, France, May 1990.

[9] H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger. Distributed Fault-Tolerant Real-Time Systems: The MARS Approach. *IEEE Micro*, February 1989.

[10] G. Fohler. Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems. In *In Proceedings of the 16th Real-Time Systems Symposium*, Pisa, Italy, December 1995.

[11] G. Fohler. *Flexibility in Statically Scheduled Hard Real-Time Systems*. PhD thesis, Technishe Universität, Wienna, Austria, 1994.

[12] 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, November 2000.

Chapter 4

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

Jan Carlson, Tomas Lennvall, and Gerhard Fohler
In the 6th International Symposium on Object-Oriented Real-Time Distributed Computing, Hakodate, Japan, May 2003

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.

4.1 Introduction

The time triggered approach has been shown to be suitable for critical real-time systems [1] [2]. 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 [3] 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 [4], 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 [5]. For very high overloads, scheduling based on value density outperforms EDF based methods. In [6], 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 [7], 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., [8], 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.

4.2 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 [9]. 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 [10]. 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.

4.2.1 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 [11], where an offline scheduler, e.g., [12] 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.

4.2.2 Handling the mixed task set

At runtime, local scheduling uses the slot shifting algorithm described in [3], 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.

4.2.3 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 [5], 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.

4.3 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 4.4. 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.
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.

[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.

4.3.1 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 4.1 and 4.2 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.

4.4 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.

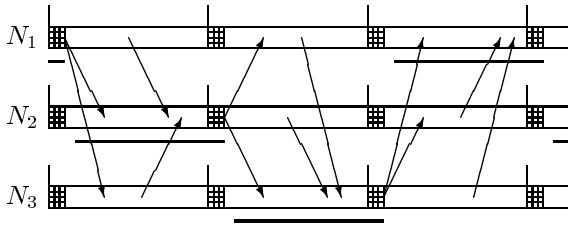


Figure 4.1: Node communication ($t = 1$).

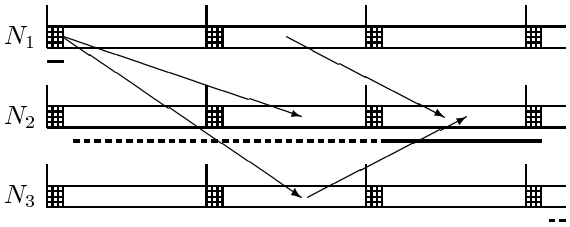


Figure 4.2: Node communication ($t = 3$).

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.

4.4.1 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 fin-

ishing 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 4.3.

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:

$$c_1 x_1 + c_2 x_2 + \ldots + c_i x_i \quad \geq \sigma_i$$

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

$$\begin{array}{ll} c_1 x_1 & \geq \sigma_1 \\ c_1 x_1 + c_2 x_2 & \geq \sigma_2 \\ \quad \vdots & \\ c_1 x_1 + c_2 x_2 + \ldots + c_n x_n & \geq \sigma_n \end{array}$$

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.

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

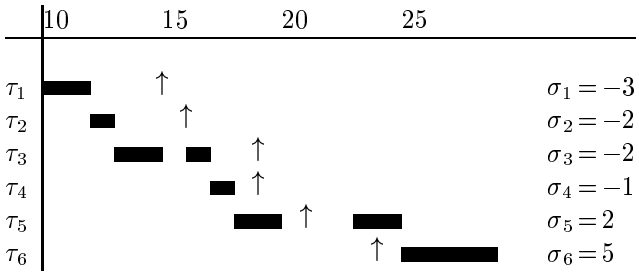
The modified definition of σ_i allows the same restrictions to be used as in the simplified setting, and the final representation of task rejection as a optimisation problem is:

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

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:

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

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.

4.4.2 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 [13]. 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 [13] 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.

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

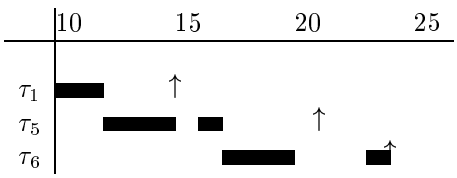
To find the best single candidate, we choose between x_1, x_3 and x_5. Since v_3 is smaller than v_1 and v_5, x_3 is chosen as single candidate. Constructing the collection, both x_4 and x_2 are added before the collection is large enough to solve the restriction. Comparing $v_3 = 10$ against $v_4 + v_2 = 15$ 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:



4.5 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 4.4.2. Each point in the figures represents some 300 simulations.
In the first experiment, all nodes in the system are subject to the same amount of load. The result is presented in Figure 5.1. 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.

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

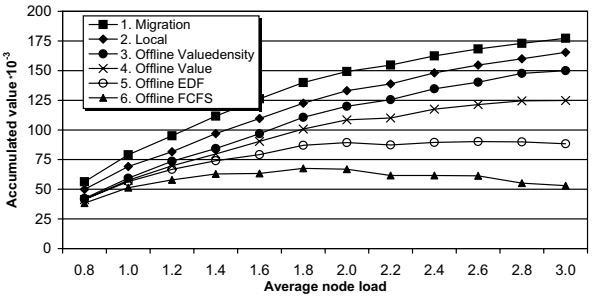


Figure 4.3: Even load distribution.

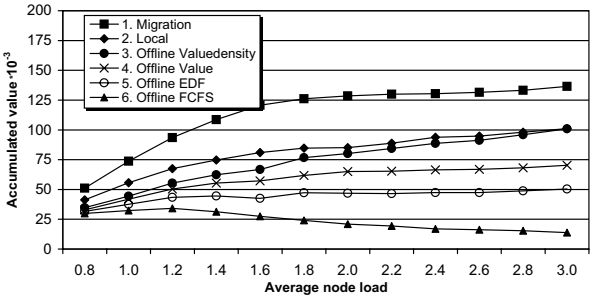


Figure 4.4: Uneven load distribution.

The second experiment, shown in Figure 5.2, 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.

4.6 Conclusions

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

Bibliography

[1] H. Kopetz. Time-Triggered Model of Computation. In *In Proceeedings 19th Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[2] H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger. Distributed Fault-Tolerant Real-Time Systems: The MARS Approach. *IEEE Micro*, February 1989.

[3] G. Fohler. Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems. In *In Proceedings of the 16th Real-Time Systems Symposium*, Pisa, Italy, December 1995.

[4] G. Buttazzo, M. Spuri, and F. Sensini. Value vs. Deadline Scheduling in Overload Conditions. In *In Proceedings of the 16th Real-Time Systems Symposium*, Pisa, Italy, December 1995.

[5] G. Buttazzo and J. Stankovic. RED: A Robust Earliest Deadline Scheduling Algorithm. In *In Proceedings of the 3rd International Workshop on Responsive Computing Systems*, September 1993.

[6] S. A. Aldarmi and A. Burns. Dynamic Value-Density for Scheduling Real-Time Systems. In *In Proceedings of the 11th Euromicro Conference on Real-Time Systems*, York, England, June 1999.

[7] S. Baruah and J. Haritsa. Scheduling for Overload in Real-Time Systems. *IEEE Transactions on Computers*, September 1997.

[8] K. Ramamritham, J.A. Stankovic, and W. Zhao. Distributed Scheduling of Tasks with Deadlines and Resource Requirements. *IEEE Transactions on Computers*, August 1989.

[9] J.A. Stankovic, K. Ramamritham, and C.-S. Cheng. Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems. *IEEE Transanctions on Computers*, December 1995.

[10] H. Kopetz. Sparse Time versus Dense Time in Distributed Real-Time Systems. In *In the 12th International Conference on Distributed Computing Systems*, Washington D.C., USA, December 1996.

[11] G. Fohler. *Flexibility in Statically Scheduled Hard Real-Time Systems*. PhD thesis, Technishe Universität, Wienna, Austria, 1994.

[12] K. Ramamritham. Allocation and scheduling of complex periodic tasks. In *In the 10th International Conference on Distributed Computing Systems*, Paris, France, May 1990.

[13] J. Carlson, T. Lennvall, and G. Fohler. Simulations Results and Algorithm Details for Value Based Overload Handling. Technical report, Mälaradlens Högskola, 2002.

Chapter 5

Paper D: Simulation Results and Algorithm Details for Value Based Overload Handling

Jan Carlson, Tomas Lennvall, and Gerhard Fohler
Technical Report, Mälardalen University, Sweden, 2002

Abstract

In this paper we present the simulation results for a proposed algorithm for value based task rejection in the presence of offline scheduled tasks for which a timely execution have to be guaranteed. We also describe in detail the algorithm for computing overload amounts.

5.1 Algorithm for computing overload amount

Given the deadlines and remaining execution times of the aperiodic tasks, and the spare capacity (slots not reserved for offline scheduled tasks) of consecutive intervals, this algorithm computes the overload amount of each aperiodic task. Let $\tau_1 \ldots \tau_n$ be a sequence of aperiodic tasks sorted by increasing deadline. Also, assume a sequence of consecutive, non-empty, time intervals, each associated to a number of offline scheduled tasks as defined by the slotshifting algorithm [1]. The following additional notation is used in the algorithm.

dl_x	the deadline of τ_x
c_x	the remaining execution time of τ_x
end_x	the end time of interval number x
sc_x	the spare capacity of interval number x
oa_x	will be assigned the overload amount of τ_x

Algorithm

Let ct be the current time, and ci the number of the interval that ct belongs to. Further, assign $oa_1 := c_1$. If the algorithm is called with compute-oa($ct, 1, ci, sc_{ci}$), then oa contains the overload values for τ_1 to τ_n, upon termination.

function compute-oa(t, d, i, c)
if $d \leq n$ **then**
  if $dl_d < end_i$ **then**
    $tmp := \min(c, dl_d - t)$
    $oa_d := oa_d - tmp$
    if $d < n$ **then** $oa_{d+1} := oa_d + c_{d+1}$
    compute-oa($dl_d, d + 1, i, c - tmp$)
  else
    $oa_d := oa_d - c$
    compute-oa($end_i, d, i + 1, sc_{i+1}$)

Note that the function is tail-recursive and thus can be implemented with bounded memory, e.g., as a standard imperative loop.

Complexity

Before considering the complexity of the algorithm, we formulate an invariant, i.e., a proposition that is true every time the function is called. For this, we

define $in(x)$ to be the number of the interval containing the time x. This allow us to formulate the invariant as $i \leq in(dl_d)$.
The correctness of the invariant is proven as follows. For the initial call to the function, we have $i = ci \leq in(dl_1)$ since no task in the sequens has already violated its deadline. Next, we assume that the invariant holds for one call, and show that this implies that it must hold for the next recursive call as well.
If the first branch of the if-then-else statement is selected, i is unchanged and $d$is increased by one in the next recursive call. Since $in(dl_d) < in(dl_{d+1})$, and since $i \leq in(dl_d)$ by assumption, we have $i \leq in(dl_{d+1})$ so the invariant holds for the next call as well.
If, instead, the else branch is selected, we must have $dl_d \geq end_i$. Assume further that the invariant does not hold for the next call. Than, since it holds for the current call, we must have $i = in(dl_d)$. This implies that $end_i \geq dl_i$, which leads to a contradiction and thus proves that the invariant must hold for the next call.
By induction, we have now shown that the invariant holds each time the function is called.
Since we have $d \leq n$, the invariant implies $i \leq in(dl_n)$. Also, we know that d and i are never decreased, that one of them is increased in each recursive call, and that they are initalised to 1 and ci respectivly. This implies that the total number of calls to the function can be no more than $n + m$, where m is the number of intervals between the current time, and the deadline of τ_n. Thus, the worst case time complexity of the algorithm is in $O(n + m)$.

5.2 Simulations

We have implemented the algorithms described in [2], and have simulated 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 randomly created offline schedules have a load of 0.4, evenly distributed over the nodes, and their length varies 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 total system 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[1], since execution time is less than wcet.

Experiment 1: Method comparison

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 (*Migration*).
2. The overload handling algorithm, without task migration (*Local*).
3. A basic algorithm that uses the offline schedule, assigning idle slots to aperiodic tasks based on value density (*Offline Valuedensity*).
4. Same as 3, but aperiodic tasks are ordered by value (*Offline Value*).
5. Same as 3, but aperiodic tasks are ordered EDF (*Offline EDF*).
6. Same as 3, but aperiodic tasks are serviced in order of arrival. (*Offline FCFS*).

Methods 1 and 2 implement the efficiency improvements suggested in [2]. Each point in the figures represents some 300 simulations.
In the first part of the experiment, all nodes in the system are subject to the same amount of load. The result is presented in Figure 5.1. Here, the possibility of task migration does not provide any significant improvement. Compared to the basic method, the performance of the proposed method is significantly higher.
The second part of the experiment, shown in Figure 5.2, 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.

[1] The actual system load varies approximately between 0.7 and 2.35 in the experiments, calculated from the distribution of actual execution times

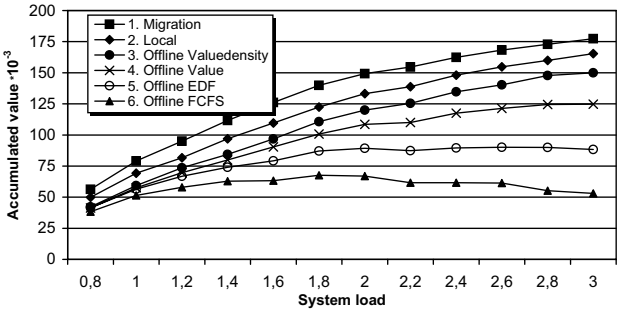


Figure 5.1: Accumulated value for even load distribution.

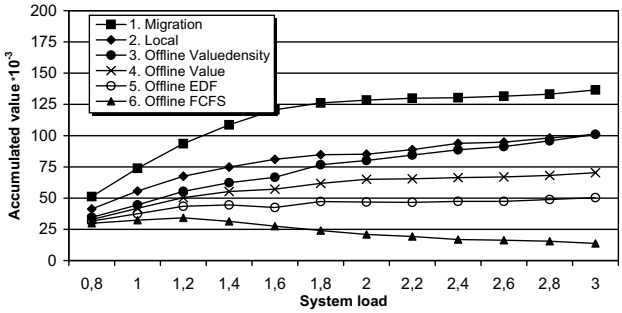


Figure 5.2: Accumulated value for uneven load distribution.

Experiment 2: Restrictions

The theoretical worst case time complexity of the overload algorithm, for a ready queue of length n, is $O(n^2)$. This experiment shows how the execution time is affected by system load, and the impact on performance from restricting the algorithm as suggested in [2] to deal with complexity issues.
The parameter *cutoff* denotes the maximum length of the ready queue. I.e., tasks that are inserted at a position greater than *cutoff* are automatically rejected, which means that they are placed in the maybe-later queue (if they just arrived, or if they were in the ready queue during the previous slot), or not stolen (if they were from a maybe-later queue).
We have measured the total accumulated value of aperiodic tasks that finished in time (similar to experiment 1) for different *cutoff* values. Execution time has been approximated by the number of arithmetic, comparison and assignment operation performed in the overload algorithm, including the computation of σ-values.
The parameters are the same as in experiment 1, with the load evenly distributed over the nodes, and using the full method from the paper (*Migration*). Figure 5.3 shows the accumulated value for different *cutoff* values. In Figure 5.4, the average number of operations for a single call to the overload algorithm is presented. Figure 5.5 gives the maximum number of operations performed during a single call to the overload algorithm. Each point in the figures represents some 300 simulations. Thus, in figures 5.4 and 5.5, each point represents over 4 million calls to the overload algorithm (8 nodes, and a simulation length of 2000).

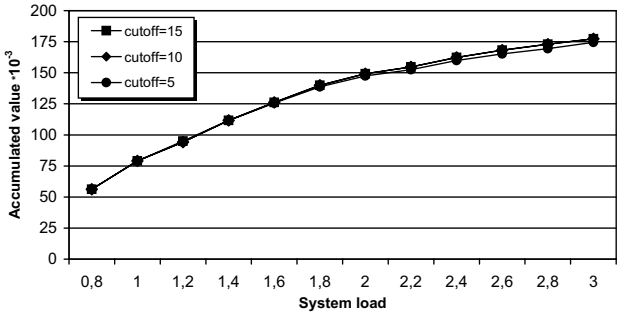


Figure 5.3: Accumulated value for different *cutoff* values.

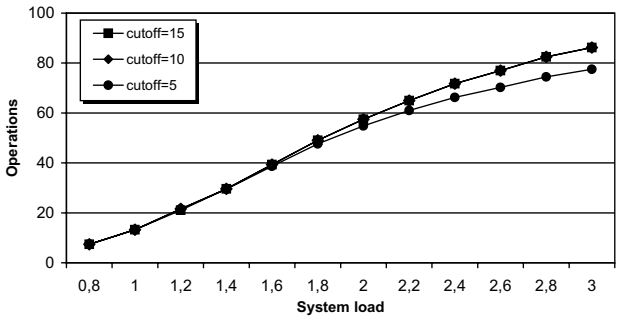


Figure 5.4: Average number of operations for different *cutoff* values.

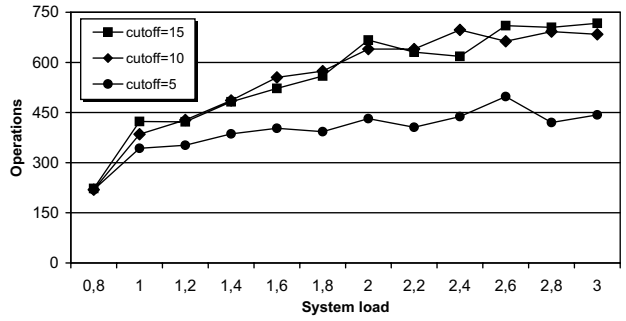


Figure 5.5: Maximum number of operations for different *cutoff* values.

In practice, the execution time is not as big an issue as the theoretical complexity suggests. None of the 57 million calls to the overload algorithm made during simulations needed more than 720 operations to be performed.
This is partly because the ready queue size (which is the parameter used in the complexity analysis) is not proportional to system load. Also, the worst case assumes that none of the restrictions are trivially solved by the solution to the previous ones, which is highly unlikely when the queue is long.
The simulations show that restricting the length of the ready queue significantly reduces worst case execution time, with only a moderate performance decrease.

Bibliography

[1] G. Fohler. Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems. In *In Proceedings of the 16th Real-Time Systems Symposium*, Pisa, Italy, December 1995.

[2] T. Lennvall, J Carlson, and G. Fohler. Enhancing Time Triggered Scheduling with Value Based Overload Handling and Task Migration. In *Proceedings of the 6th IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC 2003)*, Hakodate, Japan, May 2003.