Handling Sporadic Tasks in Real-time Systems
- Combined Offline and Online Approach -

Damir Isović
Department of Computer Engineering
Mälardalen University, Sweden
damir.isovic@mdh.se

ZA MOJE RODITELJE
(to my parents)

Abstract

Many industrial applications with real-time demands are composed of mixed sets of tasks with a variety of requirements. These can be in the form of standard timing constraints, such as period and deadline, or complex, e.g., to express application specific or non temporal constraints, reliability, performance, etc. Arrival patterns determine whether tasks will be treated as periodic, sporadic, or aperiodic. As many algorithms focus on specific sets of task types and constraints only, system design has to focus on those supported by a particular algorithm, at the expense of the rest.

In this work, we present a set of algorithms to deal with a combination of mixed sets of tasks and constraints: periodic tasks with complex and simple constraints, soft and firm aperiodic, and in particular sporadic tasks. Instead of providing algorithms tailored for a specific set of constraints, we propose an EDF based runtime algorithm, and the use of an offline scheduler for complexity reduction to transform complex constraints into the EDF model. At runtime, an extension to EDF, two level EDF, ensures feasible execution of tasks with complex constraints in the presence of additional tasks or overloads.

We present a combined offline and online approach for handling sporadic tasks. First, the sporadic tasks are guaranteed offline, during design time, which allows rescheduling or redesign in the case of failure. Since we do not know the arrival time of sporadics we need to perform the worst-case offline guarantee. Second, we try online to reduce the pessimism intoduced by worst-case assumption. An online algorithm keeps track of arrivals of instances of sporadic tasks to reduce pessimism about future sporadic arrivals and improve response times and acceptance of firm aperiodic tasks.

A simulation study underlines the effectiveness of the proposed approach.

Acknowledgements

Without support of some people, this work would not have been possible. I am grateful to them all, not only for their technical support, but also good time I have shared with them.

I would like first to thank my supervisor Gerhard Fohler (if I put the title professor before his name, I'm afraid he will freak out so I won't :) Thank you Gerhard for beliving in me and taking me as your student. I know that I need to be "pushed" sometimes and you certainly know how to do that, always pointing me into the right direction. Thank you for not only being my supervisor, but also my friend. Prost!

I would like to express my gratitude to Radu Dobrin and Tomas Lenvall for useful discussions and reviews of my work. I also want to thank my colleagues at the Department of Computer Engineering at Mälardalen University, specially the members of the System Design Lab.

I am grateful to my parents and family for always encouraging me to pursue my ideas and desires and for their unfailing support and love throughout my entire education. Lastly, my greatest thanks go to my beloved fiancé Emina for her love and support during the long night hours when completing this thesis.

Västerås, May 2001
Damir Isović

Contents

List of Figures

List of Publications

The following articles are included in this licentiate[1] thesis:

A *Handling Sporadic Tasks in Off-line Scheduled Distributed Real-Time Systems*, Damir Isovic and Gerhard Fohler, In Proceedings of 11th EUROMICRO Conference on Real-Time Systems York, England , July 1999.

B *Online Handling of Firm Aperiodic Tasks in Time Triggered Systems*, Damir Isovic and Gerhard Fohler, In WiP proceedings of 12th EUROMICRO Conference on Real-Time Systems, Stockholm, Sweden , June 2000.

C *Efficient Scheduling of Sporadic, Aperiodic, and Periodic Tasks with Complex Constraints*, Damir Isovic and Gerhard Fohler, In Proceedings of the 21st IEEE Real-Time Systems Symposium, Walt Disney World, Orlando, Florida, USA , November 2000.

D *Simulation Analysis of Sporadic and Aperiodic Task Handling*, Damir Isovic and Gerhard Fohler, Technical Report , Mälardalen Real-Time Research Centre, Mälardalen University, May 2001.

Besides the above articles, I have written and published the following scientific papers:

I *System Development with Real-Time Components*, Damir Isovic, Markus Lindgren and Ivica Crnkovic, In Proc. of ECOOP2000 Workshop 22 - Pervasive Compont Sophia Antipolis and Cannes, France , June 2000.

[1] A licenciate degree is a Swedish graduate degree halfway between MSc and PhD.

Chapter 1

Introduction

1.1 What are real-time systems?

Real-time systems are computing systems in which meeting timing constraints is essential to correctness. Usually, real-time systems are used to control or interact with a physical system, where timing constraints are imposed by the environment. As a consequence, the correct behaviour of these systems depends not only on the result of the computation but also at which time the results are produced [1]. If the system delivers the correct answer, but after a certain deadline, it could be regarded as having failed.

Many application are inherently of real-time nature; examples include aircraft and car control systems, chemical plants, automated factories, medical intensive care devices and numerous others. Most of these systems interact directly or indirectly with electronical and mechanical devices. Sensors provide information to the system about the state of its external environment. For example, medical monitoring devices, such as ECG, use sensors to monitor patient and machine status. Air speed, attitude and altitude sensors provide aircraft information for proper execution of flight control plans etc.

The design of safety-critical real-time systems has to put focus on demands on predictability, flexibility, and reliability. If we have an application with completely known characteristics, we can achieve predictable behaviour of the system, e.g., linear and angular position sensors that read a robot's arm position every 20 ms and adjust it via stepper motors. On the other hand, many external events are not predictable, for example, an external stimulus such as pressing a button. Systems must react to these sporadic events when they occur rather than

when it might be convinient. By taking care of them we introduce flexibility to the systems.

In this work, we provide mechanisms to handle unpredictable sporadic events together with predictable ones.

1.2 Dealing with time

Design of real-time systems must make sure that the system reacts to external events in a timely way. The reaction may be a simple state change, such as switching from red to green light, or a complicated controll loop controling many actuators simultaneously.

Real-time systems can be constructed out of sequential programs, but are typically built of concurrent programs, called *tasks*. A typical timing constraint on a real-time task is the deadline, i.e., the maximum time interval within which the task must complete its execution. Depending on the consequences that may occur due to a missed deadline, real-time systems are distinguished into two classes, *hard* and *soft*. In hard real-time systems all task deadlines must be met, while in soft real-time systems the deadlines are desirable but not necessary. In hard real-time systems, late data is bad data. Soft real-time systems are constrained only by average time constraints, e.g. handling input data from the keyboard. In these systems, late data is still good data. Many systems consist of both hard and soft real-time subsystems, and from now on we will refer to them as *mixed* real-time systems.

Real-time systems span a large part of computer industry. So far most of the real-time systems research has been mostly confined to single node systems and mainly for processor scheduling. This needs to be extended for multiple resources and distributed nodes. In our work, we consider a distributed system, i.e., one that consist of several processing and communication nodes [2].

1.3 Real-time scheduling policies

When a processor has to execute a set of concurrent tasks, the CPU has to be assigned to the various tasks according to a predefined criterion, called a *scheduling policy*. There is a great variety of algorithms proposed for scheduling of real-time systems today. In this subsection we will give a brief introduction to some most common classifications, which have been adopted in our research.

1.3.1 Offline vs Online

Real-time scheduling algorithms fall into two categories [3]: *offline* and *online* scheduling. In offline scheduling, the scheduler has complete knowledge of the task set and its constraints, such as deadlines, computation times, precedence constaints etc. Scheduling decisions are based on fixed parameters, assigned to tasks before their activation. The offline guaranteed schedule is stored and dispached later during runtime of the system. Offline scheduling is also refered as *static* or *pre-runtime* scheduling.

On the other hand, online scheduling algorithms make their scheduling decisions at runtime. Online schedulers are flexible and adaptive, but they can incur significant overheads because of runtime processing. Besides, online scheduling algorithms do not need to have the complete knowledge of the task set or its timing constraints. For example, an external event that arrives at the runtime of the system: we need to deal with it upon its arrival. Scheduling decisions are based on dynamic parameters that may change during system evolution. Online scheduling is often refered to as *dynamic* or *runtime* scheduling.

1.3.2 Resource sufficient vs resource insufficient

Online scheduling can be further divided into scheduling algorithms that work in *resource sufficient* and those that work into *resource insufficient* environments [4], i.e., in overload situations. In resource sufficient environments, even though tasks arrive dynamically, at any given time all tasks are schedulable. Earliest Deadline First (EDF) [5] is proven to be optimal dynamic scheduling policy in resource sufficient environments.

1.3.3 Event-trigged vs Time-trigged

There are two fundamentally different principles of how to control the activity of a real-time system, *event-trigged* and *time-trigged*. In *event-trigged* systems all activities are carried out in response to relevant events external to the system. When a significant event in the outside world happens, it is detected by some sensor, which then causes the attached device (CPU) to get an interrupt. For soft real-time systems with lots of computing power to spare, this approach is simple, and works well. The main problem with event-trigged systems is that they can fail under conditions of heavy load, i.e., when many events are happening at once. As an example of an event-trigged system we can mention the

SPRING system [6], which applies an online guarantee algorithm with complex task models in distributed environments.

In a time-trigged system, all activities are carried out at certain points in time known a priori. Accordingly, all nodes in time-trigged systems have a common notion of time, based on approximately synchronized clocks. One of the most important advantages of time-trigged control are predictable temporal behaviour of the system, which eases system validation and verification considerably. An example of a time-trigged system is the MARS system [7].

In summary, event-triggered designs give faster response at low load but more overhead and chance of failure at high load. This approach is most suitable for dynamic environments, where dynamic activites can arrive at any time. Time triggered systems have the opposite properties and are suitable in relatively static environment in which a great deal is known about the system behaviour in advance.

We will show in this work how event-triggered methods can be combined with time-triggered systems to provide for efficient inclusion of dynamic activities, in particular sporadic ones.

1.4 Task model

Our methods deal with mixed sets of tasks and constraints.

1.4.1 Periodic Tasks

There are several definitions of periodic tasks. The most common one, that has also been adopted in our work, is that a periodic tasks consist of an infinite sequence of identical activities, called *instances*, that are invoked within regular time periods. Periodic tasks are commonly found in applications such as avionics and process control accurate control requires continual sampling and processing data. Periodic tasks have usually explicit deadlines that must be met, i.e., they are usaually *hard*. We also refer to the periodic tasks as *static*, which indicates their exclusive treatment by the offline scheduler.

1.4.2 Dynamic Arrivals

A dynamic task is a sequential program that is invoked by the occurrence of an event. An event is a stimulus that may be generated by processes external to the system (e.g., an interrupt from a device) or by processes internal to the system

(e.g., the arrival of a message). Dynamically arriving tasks can be categorised on their criticality and knowledge about their occurence times.

Aperiodic Tasks usually arise from asynchronous events outside the system, such as operator request. Those events have specified response times associated with them. In general, aperiodic tasks are viewed as being activated randomly, following a certain statistical distribution (for example, a Poisson distribution). Aperiodic tasks cannot be fitted into a fixed periodic framework, i.e., their handling has to be prepared explicitly for unknown arrival times.

Furthermore, aperiodic tasks can be hard, soft and *firm*. Hard aperiodic tasks have stringent timing constraint that must be met, while soft aperiodic do not have deadlines at all. A firm aperiodic task has a deadline that must be met once the task is guaranteed online. The difference between firm and hard aperiodic tasks is that hard tasks are guaranteed offline, while firm tasks are guaranteed online, upon their arrival. Methods presented in this work provide for inclusion of both firm and soft aperiodic tasks. Firm tasks must be guaranteed while the soft ones do nor require any acceptance test: they are executed if there are no ready scheduled or guaranteed tasks.

Sporadic Tasks handle events that arrive at the system at arbitrary points in time, but with defined maximum frequency. They are invoked repeatedly with a (non-zero) lower bound on the duration between consecutive occurrences of the same event. Therefore, each sporadic task will be invoked repeatedly with a lower bound on the interval between consecutive invocations i.e., *minimum inter-arrival time* between two consecutive invocations. For example, a sporadic task with a minimum inter-arrival time of 50 ms will not invoke its instances more often than each 50 ms. In reality, it may arrive much less frequent than once every 50 ms, but if we want to schedule such a task we need to assume that it will invoke its instances with minimum inter-arrival time, i.e., maximum frequency.

1.4.3 Simple and Complex Constraints

In this work, we distinguish between *simple* constraints, i.e., period, start-time, and deadline, for the earliest deadline first scheduling model, and *complex* constraints. We refer to such relations or attributes of tasks as *complex constraints*, which cannot be expressed directly in the earliest deadline first scheduling model using period, start-time, and deadline. In most of the cases, offline transformations are needed to schedule these at runtime (some can be resolved

online at the cost of the higher overhead). Here are some examples of complex constraints:

Synchronization – Many execution sequences require a precedence order of task executions. An algoritm for the transformation of precedence constraints on single processor to suit the EDF scheduling model has been presented in [8]. However, many industrial applications require allocation of tasks with precedence constraints on different nodes, i.e., a distributed system with internode communication. The transformation of precedence constraints with an end-to-end deadline in this case requires subtask deadline assignment to create execution windows on the individual nodes so that precedence is fulfilled, e.g., [9]. A schedulability analysis for pairs of tasks communicating via a network instead of decomposition has been presented in [10].

Non-periodic execution – Constraints such as some forms of jitter, e.g., for feedback loop delay in control systems [11], require instances of tasks to be separated by *non-constant* length intervals. In order to fit these constraints into the periodic task model, it can easily happen that we end up with an over-constrained specification.

Non-temporal constraints – Usually come from the system demands, e.g., not to allocate two tasks to the same node, or to have minimum separation times etc.

Jitter – The execution start or end of a certain task is constrained by maximum variations. Strictly periodic execution can solve some instances of this problem, but over-constrains the specification. Algorithms are computationaly expensive [12].

Application specific constraints – Come from application demands, e.g. dupplicated messages on a bus may need to follow a certain pattern. They are usually imposed by engineering practice: an enginer may want to improve schedules, creating new constraints that reflect his/her practical experience.

Our method uses a general technique, capable of incorporating various types of constraints and their combinations. Those are resolved in the offline part of the method, without degrading the system performance under runtime.

1.5 This thesis

Dynamically arriving tasks cannot be fitted into a fixed periodic framework, i.e., their handling has to be prepared explicitly for unknown occurence times. Offline schedules will generally not be tight, i.e., there will be times where resources are unused. In this work we try to efficiently reclaim those resources, and use it for dynamic arrivals, i.e., aperiodic and sporadic tasks.

1.5.1 Motivation and Approach

A variety of algorithms have been presented to handle periodic and aperiodic tasks, e.g., [13], [14], [15]. An online algorithm for scheduling sporadic tasks with shared resources in hard real-time systems has been presented in [16]. Scheduling of sporadic requests with periodic tasks on an earliest-deadline-first (EDF) basis has been presented in [17]. An offline guarantee algorithm for sporadic tasks based on bandwidth reservation has been presented in [18] for single processor systems. Handling of firm aperiodic requests using a Total Bandwidth Server has been presented in [13]. Online guarantees of aperiodic tasks in firm periodic environments, where tasks can skip some instances, have been described in [19].

The slot shifting algorithm to combine offline and online scheduling was presented in [20]. It focuses on inserting aperiodic tasks into offline schedules by modifying the runtime representation of available resources. While appropriate for including sequences of aperiodic tasks, the overhead for sporadic task handling becomes too high. The use of information about amount and distribution of unused resources for non periodic activities is similar to the basic idea of slack stealing [15], [21] which applies to fixed priority scheduling. The work presented in this thesis is based on slot shifting.

These methods can generally be classified as *latest start time* methods, since they all share the characteristic of postponing the execution of hard tasks in order to give more resources to the soft tasks. Normally, as long as all guaranteed tasks meet their deadlines, it doesn't matter if they complete their execution in advance of their deadline or just before it.

However, the methods described above concentrate most on particular types of constraints. As mentione before, a real-time system might need to fullfill some complex constraints, in addition to basic temporal constraints of tasks, such as periods, start-times and deadlines. Those complex constraints can not be expressed as generally as the simple ones. Adding complex constraints to a chosen scheduling strategy increases scheduling overhead [22] or requires new,

specific schedulability tests which may have to be developed.

Besides, most of the existing methods for handling sporadic tasks perform only an online acceptance test, which introduces extra overhead to the system. When a set of sporadic tasks arrives at runtime, a scheduler performs an acceptance test. The test succeeds if each sporadic task in the set can be scheduled to meet its deadline, without causing any previously guaranteed tasks to miss its deadline, else it is rejected. A disadvantage with this approach is that if the set has been rejected, it is too late for countermeasures.

Our method provides an *offline* schedulability test for sporadic tasks, based on slot shifting. It constructs a worst case scenario for the arrival of the sporadic task set and tries to guarantee it on the top of the offline schedule. The guarantee algorithm is applied at selected slots only. At runtime, it uses the slot shifting mechanisms to feasibly schedule sporadic tasks in union with the offline scheduled periodic tasks, while allowing resources to be reclaimed for aperiodic tasks. Since the major part of preparations is performed offline, the involved online mechanisms are simple. Furthermore, the reuse of resources allows for high resource utilization.

1.5.2 Combined Offline and Online Approach

Offline scheduling methods can accomodate many specific constraints but at the expense of runtime flexibility, in particular inability to handle dynamic activities such as aperiodic and sporadic tasks. On the other hand, if we only use online scheduling, then we might introduce high overhead for resolving complex constraints, or, in the worst case, we cannot resolve them at all.

Our method is a combined offline and online approach: it integrates offline, time-trigged scheduling and dynamic, event-trigged scheduling. We use slot shifting to eliminate all types of complex constraints before the runtime of the system. They are transformed into a simple EDF model, i.e., periodic tasks with start times and deadlines. Dynamic activities are incorporated into offline schedule by making use of the unused resources and leeways in the schedule. We assume a resurce insufficient environment, where dynamic activities have to be guaranteed to fit into the offline schedule, without affecting any of previously scheduled or guaranteed activities. We provide both offline and online mechanisms for dealing with such a priori unknown activities.

The method requires a small runtime data structure, simple runtime mechanisms, going through a list with increments and decrements, provides $O(N)$ acceptance tests and facilitates changes in the set of tasks, for example to handle overloads. Furthermore, our method provides for handling of slack of non

periodic tasks as well, e.g., instances of tasks can be separated by intervals other than periods.

As a final result of this work, we provide an algorithms to deal with a combination of mixed sets of tasks and constraints: periodic tasks with complex and simple constraints, soft and firm aperiodic, and in particular sporadic tasks. Instead of providing algorithms tailored for a specific set of constraints, we propose an EDF based runtime algorithm, and the use of an offline scheduler for complexity reduction to transform complex constraints into the EDF model. At runtime, an extension to EDF, two level EDF, ensures feasible execution of tasks with complex constraints in the presence of additional tasks or overloads.

1.5.3 Handling sporadic tasks

Offline scheduling is not suitable for handling sporadic tasks due to the unknown arrival times. One approach could be to transform sporadic tasks into equvalent pseudo-periodic tasks [23] offline, which can be scheduled simply at runtime. However, this may lead to significant under-utilization of the processor time, especially when the deadline of the pseudo-periodic task is small compared to the minimum inter-arrival time of sporadic task. That because a great amount of time has to be reserved offline, before the runtime of the system, for servicing dynamic request from sporadic tasks. In extreme cases, a task handling an event which is rare, but has a tight deadline, may require reservation of all resources.

We present a combined offline and online approach for handling sporadic tasks. The offline transformation determines resource usage and distribution as well, which we use to handle sporadic tasks. Offline we assume the worst case scenario for arrival patterns for sporadic tasks, and online we try to reduce this pessimism by using the current information about the system. Dynamic activites are accomodated without affecting the feasible execution of statically scheduled tasks.

Offline part – The offline transformation determines resource usage and distribution as well, which we use to handle sporadic tasks. The sporadic tasks are guaranteed offline, during design time, which allows rescheduling or redesign in the case of failure. An offline test determines and allocates resources for sporadic tasks such that worst case arrivals can be accommodated at any time.

Online part – Since we do not know when sporadic tasks will arrive at the system, we need to assume the worst case schenario when guaranteeing them offline, i.e., we need to assume that sporadic tasks will invoke their instances at the maximum frequency. This is a too pesimistic but neccessary assumption and we try online to reduce this pesimism. At runtime, an online algorithm keeps track of arrivals of instances of sporadic tasks to reduce pessimism about future sporadic arrivals and improve response times and acceptance of firm aperiodic tasks. If a sporadic tasks invokes its instances with less frequency than the wors case one, then we can easily reclaim its reserved resources for other dynamic activities, i.e., firm and soft aperiodic tasks. When a sporadic task arrives, we do not need to account for the invocation of its next instance at least for the period of its minimum inter-arrival time and reuse its allocated resources for aperiodic tasks.

1.5.4 Conclusions

We present a method for integrated offline and online scheduling of mixed sets of tasks and constraints. A complete offline schedule can be constructed, transformed into EDF tasks, and scheduled at runtime together with other EDF tasks. The transformation is performed to maximize flexibility of task executions.

During offline analysis our algorithm determines the amount and location of unused resources, which we use to include dynamic activities during the runtime of the system. In particular, we presented an efficient method to handle sporadic tasks, providing for $O(N)$ online acceptance test for firm aperiodic tasks.

The sporadic tasks are guaranteed during design time, allowing rescheduling or redesign in the case of failure. At runtime, resources reserved for sporadic tasks can be reclaimed and used for efficient aperiodic task handling.

Thus, our method combines handling of complex constraints, efficient and flexible runtime scheduling, as well as offline and online scheduling, providing a basis for predictably flexible real-time systems

1.6 Results

This section sumarizes the main contribution of each paper in the thesis.

1.6.1 Paper A

Damir Isovic and Gerhard Fohler, *Handling Sporadic Tasks in Offline Scheduled Distributed Real-Time Systems*, In Proceedings of 11th EUROMICRO Conference on Real-Time Systems York, England , July 1999.

Summary In this paper, we presented an algorithm to handle event-triggered sporadic tasks, i.e., with unknown arrival times, but known maximum arrival frequencies, in the context of time-triggered, distributed schedules with general constraints. The sporadic tasks are guaranteed during design time, allowing rescheduling or redesign in the case of failure. At runtime, resources reserved for sporadic tasks can be reclaimed and used for efficient aperiodic task handling.

Our algorithm is based on the slot shifting method, which provides for the combination of time-triggered offline schedule construction and online scheduling of aperiodic activities. It analyzes constructed schedules for unused resources and leeway in task executions first. The runtime scheduler uses this information to include aperiodic tasks, shifting other task executions (”slots”) to reduce response times without affecting feasibility. It can also be used to perform online guarantees.

We provided an offline schedulability test for sporadic tasks based on slot shifting. It constructs a worst case scenario for the arrival of the sporadic task set and tries to guarantee it in the offline schedule. The guarantee algorithm is applied at selected slots only. At runtime, it uses the slot shifting mechanisms to feasibly schedule sporadic tasks in union with the offline scheduled periodic tasks, while allowing resources to be reclaimed for aperiodic tasks.

Since the major part of preparations is performed offline, the involved online mechanisms are simple. Furthermore, the reuse of resources allows for high resource utilization.

1.6.2 Paper B

Damir Isovic and Gerhard Fohler, *Online Handling of Firm Aperiodic Tasks in Time Triggered Systems*, In WiP proceedings of 12th EUROMICRO Conference on Real-Time Systems, Stockholm, Sweden , June 2000.

Summary In this paper we presented an algorithm for the flexible handling of firm aperiodic tasks in offline scheduled systems. It is based on slot shifting, a method to combine offline and online scheduling methods.

First, a standard offline scheduler constructs a schedule, resolving complex task constraints such as precedence, distribution, and end-to-end deadlines. This is then analyzed for unused resources and leeway in task executions. The runtime scheduler uses this information to handle aperiodic tasks, shifting other task executions (”slots”) to reduce response times without affecting feasibility. We provided an $O(N)$ acceptance test for a set of aperiodic tasks on the offline schedule and guarantee tasks without explicit reservation of resources. Our method supports flexible, value based selections of tasks to reject or remove in overload situations, and simple resource reclaiming.

While the current algorithm enables the rejection and removal of tasks, it does not address the issue of selection. We are investigating into providing a number of overload handling strategies, e.g., [24], [25].

In a previous paper [26], we presented an offline test for sporadic tasks based on worst case arrival assumptions. It cannot utilize less frequent arrivals for firm aperiodic tasks since the runtime overheads to reflect the continuous changes in resource availability are prohibitively high. We are looking into applying the algorithm presented here to handle sporadic tasks at runtime.

1.6.3 Paper C

Damir Isovic and Gerhard Fohler, *Efficient Scheduling of Sporadic, Aperiodic, and Periodic Tasks with Complex Constraints*, In Proceedings of the 21st IEEE Real-Time Systems Symposium, Walt Disney World, Orlando, Florida, USA , November 2000.

Summary In this paper we have presented methods to schedule sets of mixed types of tasks with complex constraints, by using earliest deadline first scheduling and offline complexity reduction. In particular, we have proposed an algorithm to handle sporadic tasks to improve response times and acceptance of firm aperiodic tasks.

We have presented the use of an offline scheduler to transform complex constraints of tasks into starttimes and deadlines of tasks for simple EDF runtime scheduling. We provided an extension to EDF, two level EDF, to ensure feasible execution of these tasks in the presence of additional tasks or overloads. During offline analysis our algorithm determines the amount and location of unused resources, which we use to provide $O(N)$ online acceptance tests for firm aperiodic tasks. We presented an algorithm for handling offline guaranteed sporadic tasks, which keeps track of arrivals of instances of sporadic tasks at runtime. It uses this updated information to reduce pessimism

about future sporadic arrivals and improve response times and acceptance of firm aperiodic tasks. Results of simulation study show the effectiveness of the algorithms.

Future research will deal with extending the algorithm to include interrupts, overload handling, and aperiodic and sporadic tasks with complex constraints as well. We are studying the inclusion of server algorithms, e.g., [27] into our scheduling model by including bandwidth as additional requirement in the offline transformation.

1.6.4 Paper D

Damir Isovic and Gerhard Fohler, *Efficient Scheduling of Sporadic, Aperiodic, and Periodic Tasks with Complex Constraints – Simulation Analysis*, Technical Report , Mälardalen Real-Time Research Centre, Mälardalen University, May 2001.

Summary In this paper we present the simulation results for proposed algorithms to handle mixed sets of tasks and sonstraints: periodic tasks with complex and simple constraints, soft and firm aperiodics, and sporadic tasks.

In particular, we simulated the algorithms to efficently handle sporadic tasks in order to increase acceptance ratio for online arriving firm aperiodic tasks. We have simulated the proposed guarantee algorithms and the results underlines the effectiveness of the proposed approach.

Bibliography

[1] John Stankovic and Krithi Ramamritham. Tutorial on hard real-time systems. *IEEE Computer Society Press*, 1988.

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

[3] J.A. Stankovic et. al. Implications of classical scheduling results for real-time systems. *IEEE Computer, Vol. 28, No. 6, pp. 16-25*, June 1995.

[4] J.A. Stankovic, C. Lu, and S.H. Son. The case for feedback control in real-time scheduling. In *Proc. 11th IEEE Euromicro Conference on Real-Time*, York, England, 1998.

[5] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in hard real-time environment. *Journ. of the ACM, 20, 1*, Jan. 1973.

[6] J. A. Stankovic and K. Ramamritham. The Spring kernel: A new paradigm for real-time operating systems. *IEEE Software*, pages 62–72, May 1991.

[7] A. Damm, J. Reisinger, W. Schwabl, and H. Kopetz. The real-time operating system of MARS. *ACM Operating Systems Review, SIGOPS*, 23(3):141–157, 1989.

[8] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems Journal*, 2(3):181–194, Sept. 1990.

[9] M. DiNatale and J.A. Stankovic. Applicability of simulated annealing methods to real-time scheduling and jitter control. In *Proceedings of Real-Time Systems Symposium*, Dec. 1995.

[10] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 50(2-3), 1994.

[11] M. Törngren. Fundamentals of implementing real-time control applications in distributed computer systems. *Real-Time Systems*, 1997.

[12] S. Baruah, G. Buttazzo, S. Gorinsky, and G. Lipari. Scheduling periodic task systems to minimize output jitter. In *Sixth International Conference on Real-Time Computing Systems and Applications*, Dec. 1999.

[13] M. Spuri, Giorgio C. Buttazzo, and F. Sensini. Robust aperiodic scheduling under dynamic priority systems. *In Proc. of the IEEE RTSS*, Dec. 1995.

[14] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems Journal*, 1(1):27–60, June 1989.

[15] S. R. Thuel and J.P. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *Proceddings of the Real-Time Symposium*, pages 22–33, San Juan, Puerto Rico, Dec. 1994.

[16] K. Jeffay. Scheduling sporadic tasks with shared resources in hard real-time systems. *Dept. of Comp. Sci., Univ. of North Carolina at Chapel Hill*, 1992.

[17] T. Tia, W.S. Liu, J. Sun, and R. Ha. A linear-time optimal acceptance test for scheduling of hard real-time tasks. *Dept. of Comp. Sc., Univ. of Illinois at Urbana-Champaign*, 1994.

[18] G.C. Buttazzo, G. Lipari, and L. Abeni. A bandwidth reservation algorithm for multi-application systems. *Proc. of the Intl. Conf. on Real-time Computing Systems and Applications, Japan*, 1998.

[19] M. Caccamo and Giorgio C. Buttazzo. Exploiting skips in periodic tasks for enhancing aperiodic responsiveness. *Proc. of the 18th Real-Time Systems Symposium, USA*, Dec. 1997.

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

[21] R.I. Davis, K.W. Tindell, and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In *Proceddings of the Real-Time Symposium*, pages 222–231, Dec. 1993.

[22] V. Yodaiken. Rough notes on priority inheritance. Technical report, New Mexico Institut of Mining, 1998.

[23] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, MIT, 1983. Report MIT/LCS/TR-297.

[24] G. Buttazzo and J. Stankovic. *Adding Robustness in Dynamic Preemptive Scheduling*. Kluwer Academic Publishers, 1995.

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

[26] Damir Isovic and Gerhard Fohler. Handling sporadic tasks in off-line scheduled distributed hard real-time systems. *Proc. of 11th EUROMICRO conf. on RT systems, York, UK*, June 1999.

[27] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of Real-Time Systems Symposium*, Dec. 1998.

Chapter 2

Paper A: Handling Sporadic Tasks in Offline Scheduled Distributed Real-Time Systems

Damir Isovic and Gerhard Fohler
In Proceedings of 11th EUROMICRO Conference on Real-Time Systems York, England, July 1999

Abstract

Many industrial applications mandate the use of a time-triggered paradigm and consequently the use of off-line scheduling for reasons such as predictability, certification, cost, or product reuse. The construction of an off-line schedule requires complete knowledge about all temporal aspects of the application. The acquisition of this information may involve unacceptable cost or be impossible. Often, only partial information is available from the controlled environment.

In this paper, we present an algorithm to handle event-triggered sporadic tasks, i.e., with unknown arrival times, but known maximum arrival frequencies, in the context of distributed, off-line scheduled systems. Sporadic tasks are guaranteed during design time, allowing rescheduling or redesign in the failure case. At run-time, the sporadic tasks are scheduled dynamically, allowing the reuse of resources reserved for, but not consumed by the sporadic tasks. We provide an off-line schedulability test for sporadic tasks and apply the method to perform on-line scheduling on top of off-line schedules. Since the major part of preparations is performed off-line, the involved on-line mechanisms are simple. The on-line reuse of resources allows for high resource utilization.

2.1 Introduction

Off-line scheduling is mandated by a number of industrial applications. Predictability, cost, product reuse, and maintenance are examples for reasons advocating a time-triggered approach. The off-line construction of schedules, however, requires complete knowledge about application characteristics before the run-time of the system. Often, such comprehensive information is inaccessible due to high cost of acquisition or unavailability. Rather, incomplete characteristics are available. One typical, partially known property is the arrival time of activities. Instead of exact arrival patterns, bounds on the arrival frequencies are known, e.g., derived from a model of a physical process. One such limit is the *minimum inter-arrival time* between subsequent instances of tasks. Tasks for which it is known are called *sporadic tasks* [1].

The use of off-line scheduling for sporadic tasks is problematic; they can be transformed into pseudo-periodic tasks [1], but with potentially prohibitive overhead.

An on-line algorithm for scheduling sporadic tasks with shared resources in hard real-time systems has been presented in [2]. Scheduling of sporadic requests with periodic tasks on an *earliest-deadline-first (EDF)* basis [3] has been presented in [4]. Handling of firm aperiodic requests using a Total Bandwidth Server has been presented in [5]. On-line guarantees of aperiodic tasks in firm periodic environments, where tasks can skip some instances, have been described in [6]. Systems containing hard real-time sporadic tasks have been analyzed for their worst case behavior in [7].

These algorithms above perform on-line guarantees. When a set of sporadic tasks arrives at run-time to the system, a scheduler performs an acceptance test. The test succeeds if each sporadic task in the set can be scheduled to meet its deadline, without causing any off-line guaranteed periodic tasks or previously accepted non-periodic tasks to miss its deadline, else it is rejected. A disadvantage with this approach is that if the set has been rejected, it is too late for countermeasures.

An off-line guarantee algorithm for sporadic tasks based on bandwidth reservation has been presented in [8] for single processor systems.

In this paper, we present a method providing an off-line feasibility test for sporadic tasks on top of an off-line scheduled, distributed periodic task set with general constraints, e.g., precedence; rescheduling or redesign can be performed, should the test fail. Given the deadline, maximum frequency and execution time for each task in the sporadic task set, we can create a worst case load pattern. Then, we try to guarantee this worst case sporadic demand within

the periodic schedule before the system starts its execution. We have to account for arrivals at any time; it is, however, sufficient to investigate only some selected points in time. Our method is based on the *slot shifting* [9] method, which provides for the combination of off-line and on-line scheduling. After a static schedule for the distributed periodic tasks has been created off-line in a first step, the amount and distribution of unused resources and leeways in it is determined. These are then used to incorporate aperiodic tasks into the schedule by shifting the off-line scheduled tasks' execution, without violating their feasibility. An on-line mechanism is used to guarantee and schedule aperiodic tasks. Using the on-line scheduling algorithm of slot shifting, the resources reserved for, but not consumed by sporadic tasks can be reused. Should sporadic tasks arrive at less than their guaranteed maximum frequency, their resources can be reclaimed, e.g., for aperiodic tasks. Furthermore, the on-line algorithm of slot shifting has been modified to schedule guaranteed sporadic requests.

The method presented in this paper allows for the handling of distributed periodic tasks with general constraints, such as precedence, based on the time-triggered paradigm, together with the event-triggered scheduling of guaranteed sporadic tasks and online aperiodic tasks, possibly reclaiming resources.

The rest of this paper is organized as follows: First, a description of system and task model is given in section 2.2, and a brief summary of the slot shifting method in section 2.3. The off-line guarantee test for the sporadic tasks is presented in section 2.4, followed by the online mechanisms in section 2.5. An example in section 2.6 illustrates the discussed mechanisms. Finally, section 2.7 concludes the paper.

2.2 System description and task model

The system is considered to be *distributed*, i.e., one that consists of several processing and communication nodes [10].

2.2.1 Time model

We assume a discrete time model [11]. Time ticks are counted globally, by a synchronized clock with granularity of slot length, and assigned numbers from 0 to ∞. The time between the start and the end of a slot *i* is defined by the interval $[slotlength * i, slotlength * (i + 1)]$. Slots have uniform length and start and end at the same time for all nodes in the system. Task periods and deadlines must be multiples of the slot length.

2.2.2 Off-line periodic schedule

A schedule is a sequence of *n* slots. For static schedules the number of slots is typically equal to the *least common multiple (LCM)* of all involved periods.

2.2.3 Task model

All tasks in the system are fully preemptive and communicate with the rest of the system via data read at the beginning and written at the end of their executions.

Periodic tasks execute their invocations within regular time intervals. A periodic task T_P is characterized by its maximum execution time ($MAXT$) [12], period (P) and relative deadline of (Dl).
The k^{th} invocation of T_P is denoted T_P^k and is characterized by its earliest start time (est) and absolute deadline (dl). The absolute deadline of the k^{th} invocation of T_P is equal to the sum of the earliest start time of its preceding invocation and the relative deadline.

Aperiodic tasks are invoked only once. Their arrival times are unknown at design time. A hard aperiodic task T_A has the following set of parameters: the arrival time (a), maximum execution time and relative deadline. Soft aperiodic tasks have no deadline constraints.

Sporadic tasks arrive to the system at random points in time, but with defined minimum inter-arrival times between two consecutive invocations. We do not know when they arrive to the system, but we do know their maximum frequency. A sporadic task T_S is characterized by its relative deadline, minimum inter-arrival time (λ) and maximum execution time.
The attributes above are known before the run-time of the system. The additional information that becomes available on-line, upon the arrival time of the k^{th} invocation is its arrival time and its absolute deadline.

2.3 Integrated off-line and on-line Scheduling

In this section, we briefly describe the slot shifting method which we use as a basis to combine off-line and on-line scheduling. It provides for the efficient handling and possibly on-line guarantee of aperiodic tasks on top of a

distributed schedule with general task constraints. Slot shifting extracts information about unused resources and leeway in an off-line schedule and uses this information to add tasks feasibly, i.e., without violating requirements on the already scheduled tasks. A detailed description can be found in [9].

2.3.1 Off-line preparations

First, an off-line scheduler [13] creates scheduling tables for the periodic tasks. It allocates tasks to nodes and resolves precedence constraints by ordering task executions.

Start-times and deadlines The scheduling tables list fixed start- and end times of task executions, that are less flexible than possible. The only assignments fixed by specification, however, are the initiating and concluding tasks in the precedence graph, and, as we assume message transmission times to be fixed here[1], tasks sending or receiving inter-node messages. These are the only fixed start-times and deadlines, all others are calculated recursively, as the execution of all other tasks may vary within the precedence order, i.e., they can be shifted.

Intervals and spare capacities The deadlines of tasks are then sorted for each node and the schedule is divided into a set of *disjoint execution intervals* for each node. Spare capacities 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.

The spare capacities of an interval I_i are calculated as given in formula 2.1:

$$sc(I_i) \;=\; |I_i| - \sum_{T \in I_i} MAXT(T) + min(sc(I_{i+1}), 0) \qquad (2.1)$$

The length of I_i minus the sum of the activities assigned to it is the amount of idle times 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.

[1] We apply the same mechanisms to the network as well, i.e., shifting messages, as detailed in [9].

2.3.2 On-Line mechanisms

During system operation, the on-line scheduler is invoked after each slot. It checks whether aperiodic tasks have arrived, performs the guarantee algorithm, and selects a task for execution. This decision is then used to update the spare capacities. Finally the scheduling decision is executed in the next slot.

Guarantee Algorithm 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 spare capacity 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 $dl(T_A)$, and
- $min(sc(I_{l+1}), dl(T_A) - start(I_{l+1}))$, 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 $MAXT(T_A)$, 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. This guarantee algorithm is $O(N)$, N being the number of intervals. It is shown in [14], that this acceptance test has equivalent results – but with simpler run-time handling – as to the ones presented in [15] and [16], which are optimal for single processors.

On-line scheduling On-line 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. Soft aperiodic tasks, i.e., without deadline, can be executed immediately if $sc(I_c) > 0$. After each scheduling decision, the spare capacities of the affected intervals are updated.

2.4 Acceptance test for a set of sporadic tasks

In this section we will introduce an off-line guarantee algorithm for a set of sporadic tasks. The set is said to be *feasible* with the already scheduled task set if it is possible to schedule all tasks in the sporadic set such that no scheduled

periodic task misses its deadline.
Firstly, the off-line periodic schedule is created and analyzed for slot shifting. Secondly, the set of sporadic tasks is tried to fit into the periodic schedule, by investigating only selected time slots. If the sporadic set is not accepted, it is up to designer to redesign the system, i.e., reschedule periodic tasks or change the sporadic set.

2.4.1 Sporadic set

All tasks in the sporadic set are assumed to be invoked with their maximum frequency, creating the worst case scenario for the scheduler. If the deadline of a sporadic task can be guaranteed for the release with their maximum frequency, then all subsequent deadlines are guaranteed. Examples of this approach are given in [17]. The minimum time difference between successive releases of a sporadic task is its minimum inter-arrival time. It has been shown [7] that a sporadic task which is released with its maximum frequency behaves exactly like a periodic task with period equal to its minimum inter-arrival time. Now we know the deadline, the maximum execution time and the 'period' of each sporadic task in the set and we can use that information to try to guarantee the set for its worst load pattern.

2.4.2 Critical slots

One way of investigating if the sporadic set fits into the periodic schedule is to investigate if it fits at each time slot of the periodic schedule, but this is impractical. It is sufficient to investigate only some selected points in time, called *critical slots (t_c)*. There is only one critical slot per interval[2] , and if the sporadic set can be guaranteed at the critical slot, it will be guaranteed at every other slot within the same interval.
The worst case for arrival of the sporadic set to an interval I is the slot where the execution of the sporadic tasks can be delayed maximally by the execution of the off-line scheduled tasks. This gives:

Proposition 1 Critical slot t_c for an interval I is calculated as:

$$t_c(I) = start(I) + sc(I)$$

[2]Intervals are calculated as described in section 2.3.

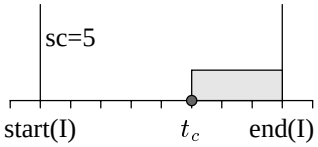


Figure 2.1: Example of a critical slot.

as depicted in figure 2.1. If a sporadic task T_S can be guaranteed at the critical slot, it will be guaranteed at each other slot within the same interval I:

$$\forall t \in I, T_S\ guaranteed\, at\, t_c \Rightarrow T_S\ guaranteed\, at\, t$$

Proof. Let:

$t_c(I) = start(I) + sc(I)$ - the critical slot of I.
$t \in I, t \neq t_c$ - some other slot in I.

Assume the following is correct:

Assumption 1. *There is a slot t in interval I such that a sporadic task T_S can be guaranteed at the critical slot t_c, but not at t:*

$$\exists t \in I, (T_S\ guaranteed\, at\, t_c) \wedge (T_S \neg guaranteed\, at\, t)$$

Let δ denote the difference between spare capacities available for T_S at t_c and t, i.e., the amount of spare capacity that we may get or lose by shifting the arrival time of T_S from t_c to t. Assumption 1 states that T_S can be guaranteed at t_c but not at t, which means that there is more spare capacity available at t_c than at t and we lose spare capacity if we shift. This implies that δ is negative:

$$\delta < 0 \tag{2.2}$$

There are two possibilities for the arrival of the sporadic task T_S, $t = A(T_S)$, before or after the critical slot t_c:

Case 1: $t > t_c$, T_S arrives after t_c, as depicted in figure 2.2.
The requirement for T_S to be accepted is that the spare capacity available for T_S at its arrival time has to be greater or equal to the maximum execution time of T_S.
Let α and β denote the change of spare capacity caused by shifting in the arrival and deadline, resp., interval as depicted in figure 2.3:

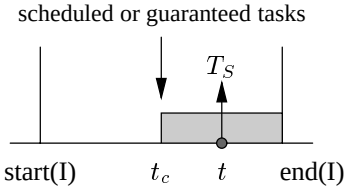


Figure 2.2: Sporadic arival after critical slot.

- α - the difference in spare capacity of the arrival interval caused by shifting the arrival time of T_S from t_c to t.
- β - the difference in spare capacity of the deadline interval caused by shifting the deadline of T_S.

This gives:

$$\delta = \alpha + \beta \tag{2.3}$$

Shifting the arrival time of T_S from t_c to t means that the deadline of T_S is shifted to the right. In the arrival interval, $I_{arrival}$, slots from t_c to t are reserved for the execution of the scheduled periodic tasks, giving $\alpha = 0$. In the deadline interval, $I_{deadline}$, shifting the deadline of T_S may only increase the portion of available spare capacities in that interval. This gives that β has to be greater or equal to zero ($\beta \geq 0$).

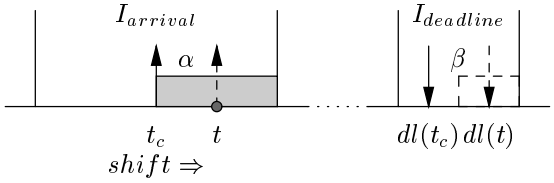


Figure 2.3: Sporadic arrival shifted to the right.

The maximum value of δ occurs when the deadline of T_S does not intersect with any other activity, that is, execution of some other task. In other words, $\beta = t - t_c > 0$. If so, then:

$$\delta = \alpha + \beta > 0, (\alpha = 0, \beta > 0) \tag{2.4}$$

Otherwise, if $dl(T_S)$ occurs during the execution of some other task, the worst case scenario is that we do not get any new resources for T_S, that is:

$$\delta = \alpha + \beta = 0, (\alpha = 0, \beta = 0) \tag{2.5}$$

$(4) \wedge (5)$ gives:

$$\delta \geq 0$$

which is contradictory to (2), making assumption 1 false.

Case 2: $t < t_c$, T_S arrives before t_c, as depicted in figure 2.4.

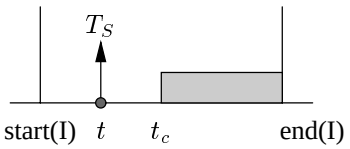


Figure 2.4: Sporadic arrival before critical slot.

Now we shift the arrival time of T_S to the left, that is before the critical point t_c. This is shown in figure 2.5.

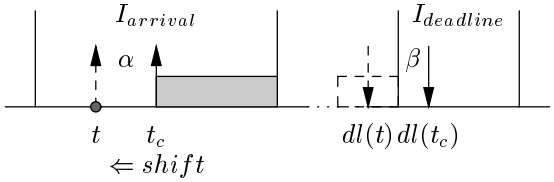


Figure 2.5: Sporadic arrival shifted to the left.

Let α and β denote the same as in case 1. Shifting the arrival time of T_S results in a positive α that is equal to the difference between t_c and t, i.e., $\alpha = t_c - t > 0$. In the deadline interval, the amount of lost spare capacities caused by shifting can maximally be the same as the amount of gained spare capacities in the arrival interval, giving $\beta_{worst} = -\alpha$. This implies:

$$\delta = \alpha + \beta = \alpha + (-\alpha) = 0, (\beta = \beta_{worst}) \tag{2.6}$$

In a more optimistic scenario, we can even lose less spare capacities in the deadline interval than we get in the arrival interval, that is $\beta < \alpha$. In that case, we get:

$$\delta = \alpha + \beta > 0, (|\beta| < \alpha) \tag{2.7}$$

$(6) \wedge (7)$ implies:

$$\delta \geq 0$$

which is contradictory to (2). This implies the assumption 1 doesn't hold for case 2.
Assumption 1 doesn't hold either for case 1 or case 2. Therefore proposition 1 is true. This concludes the proof.

□

Critical points are calculated off-line for each interval, and only those points are checked for the feasibility of the sporadic task set.

2.4.3 Off-line feasibility test for sporadic tasks

The feasibility test for the set of sporadic tasks works by creating a worst case load demand of the sporadic tasks as described in section 2.4.1. We assume that all sporadic tasks arrive with their maximum frequency and test if the demand created can be accommodated into the static schedule at all critical slots. Here follows the off-line guarantee algorithm for a set of sporadic tasks $\mathcal{S}$:

Let:

i	= index of $I_{arrival}$
k	= index of $I_{deadline}$
sc_a	= available sc for a sporadic task T_S from $a(T_S)$ to $dl(T_S)$
R	= an array containing slots reserved for previously guaranteed sporadic tasks
$initR()$	= initiates R to empty set
$countR(x, y)$	= returns number of reserved slots between slot x and slot y.
$reserveR(n, d)$	= reserves n slots as close to d as possible (as late as possible)

1: $\forall t_c$
2: $initR()$
3: $\forall T_S \in \mathcal{S}$

4: $\forall T_S^n \in LCM_S$
5: $sc_a(T_S^n) = \sum_{j=i+1}^{k-1} sc(I_j)$
6: $+min(sc(I_k), dl(T_S^n) - start(I_k))$
7: $-countR(a(T_S^n), dl(T_S^n))$
8: $if \quad (sc_a(T_S^n) \geq MAXT(T_S))$
9: $then \quad reserveR(MAXT(T_S), dl(T_S^n))$
10: $else$ abort (set rejected)

Comments:
1: Investigate every critical slot.
2: No slots reserved yet.
3: Guarantee every sporadic task T_S in the set.
4: Guarantee every invocation T_S^n of T_S.
5: Calculate sc available for T_S from its arrival until its deadline. It is equal to the sum of sc for all full intervals between $I_{arrival}$ and the $I_{deadline}$ of T_S^n, increased by
6: the remaining sc of the $I_{deadline}$ available until $dl(T_S^n)$, decreased by
7: the amount of sc reserved for other, previously guaranteed sporadics that intersect with T_S^n.
8: If the available sc is greater or equal to the maximum execution time of T_S, then
9: reserve slots needed for T_S^n as close to its dl as possible, and continue.
10: If not enough spare capacity, abort the guarantee algorithm and report that the guaranteeing failed.

2.5 On-line mechanism

During the system operation, the on-line scheduler is invoked after each slot. It checks whether new dynamic tasks have arrived during the last slot. When a set of sporadic tasks arrives to the system, the ready set of slot shifting is expanded by sporadic tasks that are ready to execute. Soft aperiodic tasks can be executed if the spare capacity of the current interval is greater than zero, and there are no ready sporadic tasks.

Let:
t = current time
$sc(I)_t$ = spare capacity of the current interval at time t.
$\mathcal{R}(t)$ = the ready set that consists of all periodic

and guaranteed sporadic tasks that have earliest start time less or equal to the current time.

We identify the following cases:

1. $\mathcal{R}(t) = \{\}$: There are no tasks ready to be executed, the CPU remains idle.
2. $\mathcal{R}(t) \neq \{\} \wedge\ \exists T_A,\ T_A$ soft aperiodic:
   (a) $sc(I)_t > 0 \wedge \exists T_S \in \mathcal{R}(t), T_S$ sporadic $\Rightarrow$ execute T_S.
   (b) $sc(I)_t > 0 \wedge \neg\exists T_S \in \mathcal{R}(t) \Rightarrow$ execute T_A.
   (c) $sc(I)_t = 0$: a periodic task from ready set has to be executed. Zero spare capacities indicate that adding further activities will result in a deadline violation of the guaranteed task set.
3. $\mathcal{R}(t) \neq \{\} \wedge \neg\exists T_A, T_A$ soft aperiodic: The task of ready set with the shortest deadline is executed.

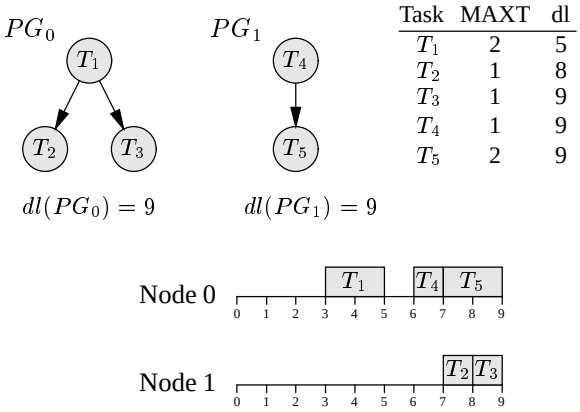
2.5.1 Maintenance of spare capacities

The decision of the scheduler is now used to update spare capacities, depending on which type of task was selected for execution:

- *Aperiodic execution*: one slot of the spare capacities is used to execute a slot of dynamic task. The spare capacity of the current interval has to be decremented by one.
- *Periodic execution*: Executing a static task only swaps spare capacities. Depending on the interval to which the executed task belongs to, the current interval I_i, or a subsequent one $I_j, j > i$ is affected. The amount of total spare capacities is unchanged.
- *Sporadic execution*: The spare capacity of the current interval is decremented by one.
- *No execution*: One slot of spare capacity is used without dynamic processing. Spare capacity has to be decremented by one.

2.6 Example

Assume the following periodic tasks with execution times and precedence constraints as described in figure 2.6.



Task	MAXT	dl
T_1	2	5
T_2	1	8
T_3	1	9
T_4	1	9
T_5	2	9

Figure 2.6: Example offline sporadic guarantee – periodic tasks and offline schedule.

We calculate intervals and spare capacities, as described in 2.3.1, and critical slots as described in 2.4.2:

$Interval$	$Node$	$start$	end	sc	t_c
I_0	0	0	5	3	3
I_1	0	5	9	1	6
I_2	1	6	8	1	7
I_3	1	8	9	0	8

Intervals with their assigned tasks and critical slots are depicted in figure 2.7. Now assume a sporadic set $\mathcal{S} = \{S_1(1,5), S_2(3,10)\}$ where the first parameter is maximum execution time and the other one the minimum inter-arrival time at node 0. If we assume that sporadic tasks arrive with their maximum frequencies, then the deadline of each invocation is equal to the release of the next invocation. Now we apply the off-line guarantee algorithm on each task in the sporadic set $\mathcal{S}$. First, we try to guarantee S_1 and S_2 at critical slot 3, and if they can be guaranteed, we proceed with investigation of slot 6. The LCM of $\mathcal{S}$ is 10, which means that S_1 is invoked twice and S_2 once before the pattern is repeated. We now illustrate the guarantee test for S_1 and S_2. Numbers

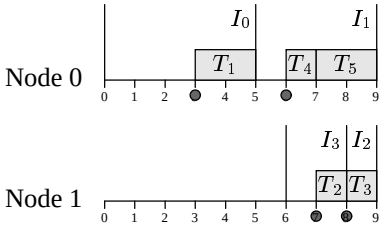


Figure 2.7: Example offline sporadic guarantee – schedule with intervals.

above columns represent steps in the guarantee algorithm described in 2.4.3 (see figure 2.8 in parallel):

1	3	4	5	8	9	10
$\mathbf{t_c}$	$\mathbf{T_S}$	**Inv.**	$\mathbf{sc_a}$	$\geq \mathbf{MAXT}(?)$	$\mathbf{R}$	
3	S_1	1	1	$\geq 1 \Rightarrow \top$	{5}	
		2	3	$\geq 1 \Rightarrow \top$	{5,11}	
	S_2	1	2	$\geq 3 \Rightarrow \bot$		abort

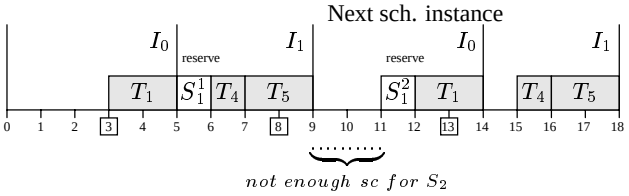


Figure 2.8: Example offline sporadic guarantee – steps in guarantee algorithm.

The sporadic set cannot be guaranteed at critical slot 3. What we can do now is to redesign the system and try again. Since we support distributed systems, we could reallocate some of the periodic tasks from node 0 to node 1, or allocate some of sporadics on node 1. In this example, we decide to schedule the periodic task T_4 on node 1 instead of node 0. The new periodic schedule is depicted in figure 2.9. Intervals remain the same, spare capacities and critical slots have to be recalculated for I_1 and I_2:

I_1 : $sc(I_1)$ = 1+1=2 $t_c(I_1)$ = 5+2=7

I_2 : $sc(I_2)$ = 1-1=0 $t_c(I_2)$ = 6+0=6

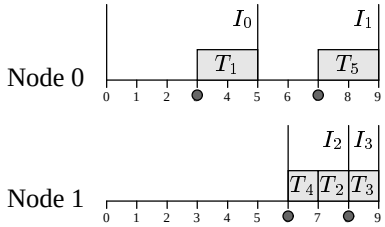
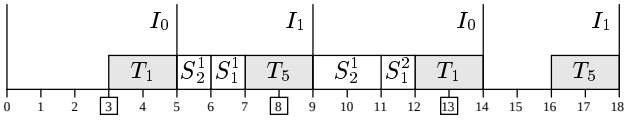


Figure 2.9: Example offline sporadic guarantee – offline schedule after redesign.

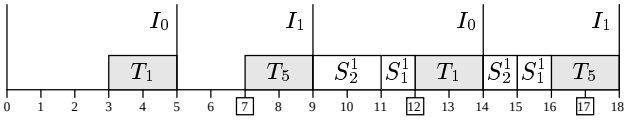
We try to guarantee $\mathcal{S}$ on node 0 again, but this time one critical point has changed; we got 7 instead for 6:

1	3	4	5	8	9
$\mathbf{t_c}$	**Task**	**Inv.**	$\mathbf{sc_a}$	$\geq (?)$	$\mathbf{R}$
3	S_1	1	2	$\geq 1 \Rightarrow \top$	{6}
		2	3	$\geq 1 \Rightarrow \top$	{6,11}
	S_2	1	3	$\geq 3 \Rightarrow \top$	{5,6,9,10,11}
7	S_1	1	3	$\geq 1 \Rightarrow \top$	{11}
		2	2	$\geq 1 \Rightarrow \top$	{11,15}
	S_2	1	3	$\geq 3 \Rightarrow \top$	{9,10,11,14,15}

The sporadic set is guaranteed at both critical slots and we can accept it.
Next follows the description of on-line execution (as described in section 2.5) on node 0 for the periodic and sporadic tasks described above, extended with one soft aperiodic task, $A_1(2, 2)$, where the first parameter is arrival time, and the other one is execution time. Assume $\mathcal{S}$ arrives at slot 3. $\mathcal{R}(t)$ containts tasks that are ready to execute in slot t. The execution trace is depicted in figure 2.11 ("case" refers to the cases described in 2.5):



a) Critical slot 3



b) Critical slot 7

Figure 2.10: Example offline sporadic guarantee – guaranteeing after redesign.

t	$\mathcal{R}(t)$	case	exe.	sc
0	$\{T_1, T_5\}$	3	T_1	unchanged
1	$\{T_1, T_5\}$	3	T_1	unchanged
2	$\{T_5, A_1\}$,	2b	A_1	$sc(I_0)$ decreased
3	$\{T_5, S_1^1, S_2, A_1\}$	2a	S_1^1	$sc(I_0)$ decreased
4	$\{T_5, S_2, A_1\}$	2a	S_2	$sc(I_0)$ decreased
5	$\{T_5, S_2, A_1\}$	2a	S_2	$sc(I_1)$ decreased
6	$\{T_5, A_1\}$	2b	A_1	$sc(I_1)$ decreased
7	$\{T_5\}$	3	T_5	unchanged
8	$\{T_5, S_1^2\}$	2c	T_5	unchanged

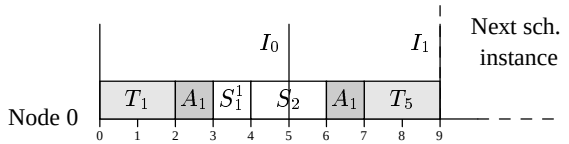


Figure 2.11: Example offline sporadic guarantee – online execution.

2.7 Conclusion

In this paper, we presented an algorithm to handle event-triggered sporadic tasks, i.e., with unknown arrival times, but known maximum arrival frequencies, in the context of time-triggered, distributed schedules with general constraints. The sporadic tasks are guaranteed during design time, allowing rescheduling or redesign in the case of failure. At run-time, resources reserved for sporadic tasks can be reclaimed and used for efficient aperiodic task handling.

Our algorithm is based on the slot shifting method, which provides for the combination of time-triggered off-line schedule construction and on-line scheduling of aperiodic activities. It analyzes constructed schedules for unused resources and leeway in task executions first. The run-time scheduler uses this information to include aperiodic tasks, shifting other task executions ("slots") to reduce response times without affecting feasibility. It can also be used to perform online guarantees.

We provided an off-line schedulability test for sporadic tasks based on slot shifting. It constructs a worst case scenario for the arrival of the sporadic task set and tries to guarantee it in the off-line schedule. The guarantee algorithm is applied at selected slots only. At run-time, it uses the slot shifting mechanisms to feasibly schedule sporadic tasks in union with the off-line scheduled periodic tasks, while allowing resources to be reclaimed for aperiodic tasks.

Since the major part of preparations is performed off-line, the involved on-line mechanisms are simple. Furthermore, the reuse of resources allows for high resoure utilization.

Acknowledgements

The authors wish to thank the reviewers for their fruitful comments which helped to improve the quality of the paper. Further thanks go to Jukka Mäki-Turja and Björn Lindberg for their careful reviewing and stimulating discussions.

Bibliography

[1] A.K. Mok. *Fundamental Design Problems for the Hard Real-Time Envs.* PhD thesis, MIT, May 1983.

[2] K. Jeffay. Scheduling sporadic tasks with shared resources in hard real-time systems. *Dept. of Comp. Sci., Univ. of North Carolina at Chapel Hill*, 1992.

[3] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in hard real-time environment. *Journ. of the ACM, 20, 1*, Jan. 1973.

[4] T. Tia, W.S. Liu, J. Sun, and R. Ha. A linear-time optimal acceptance test for scheduling of hard real-time tasks. *Dept. of Comp. Sc., Univ. of Illinois at Urbana-Champaign*, 1994.

[5] M. Spuri, Giorgio C. Buttazzo, and F. Sensini. Robust aperiodic scheduling under dynamic priority systems. *In Proc. of the IEEE RTSS*, Dec. 1995.

[6] M. Caccamo and Giorgio C. Buttazzo. Exploiting skips in periodic tasks for enhancing aperiodic responsiveness. *Proc. of the 18th Real-Time Systems Symposium, USA*, Dec. 1997.

[7] A. Burns, N.C. Audsley, M.F. Richardson, and A.J. Wellings. Hard real-time scheduling: the deadline monotonic approach. *Proc. of the IFAC/IFIP Workshop, UK*, 1992.

[8] G.C. Buttazzo, G. Lipari, and L. Abeni. A bandwidth reservation algorithm for multi-application systems. *Proc. of the Intl. Conf. on Real-time Computing Systems and Applications, Japan*, 1998.

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

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

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

[12] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *RT Systems Journal*, 1989.

[13] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

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

[15] M.R. Garey., D.S. Johnson, B.B. Simons, and R.E. Tarjan. Scheduling unit-time tasks with arbitrary release times and deadlines. *IEEE Trans. on Soft. Eng.*, May 1981.

[16] M. Chetto and H. Chetto. Scheduling periodic and sporadic tasks in a real-time system. *Inf. Proc. Letters*, Feb. 1989.

[17] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Deadline monotonic scheduling theory. *WRTP'92. Preprints of the IFAC Workshop. Pergamon Press, U.K*, 1992.

Chapter 3

Paper B: Online Handling of Firm Aperiodic Tasks in Time Triggered Systems

Damir Isovic and Gerhard Fohler
In WiP Proceedings of 12th EUROMICRO Conference on Real-Time Systems,
Stockholm, Sweden, June 2000

Abstract

A number of industrial applications advocate the use of time triggered approaches for reasons of predictability, cost, product reuse, and maintenance. The rigid offline scheduling schemes used for time triggered systems, however, do not provide for flexibility. At runtime, aperiodic tasks can only be included into the unused resources of the offline schedule, supporting neither guarantees nor fast response times.

In this paper we present an algorithm for flexible hard aperiodic task handling in offline scheduled systems: it provides an O(N) acceptance test to determine if a set of aperiodics can be feasibly included into the offline scheduled tasks, and does not require runtime handling of resource reservation for guaranteed tasks. Thus, it supports flexible schemes for rejection and removal of aperiodic tasks, overload handling, and simple reclaiming of resources.

As a result, our algorithm provides for a combination of offline scheduling and online hard aperiodic task handling.

3.1 Introduction

Time triggered real-time systems have been shown to be appropriate for a variety of critical applications: they provide verifiable timing behavior and allow for distribution, complex application structures, and general constraints, such as precedence or end-to-end deadlines. Their benefits are, however, limited for applications with not completely known characteristics, such as arrival times. Including such non periodic tasks in the rigid offline schedules, as used for time triggered systems, can be resource inefficient at best for sporadic tasks, and impossible for aperiodics, i.e., without knowledge of arrival times.

Real-world applications demand flexible handling of aperiodic tasks: efficient acceptance tests, resource reservation if a task can be guaranteed, but also specific rejection strategies for the negative case. Often, aperiodics are guaranteed on a first-come-first-serve basis, i.e., already guaranteed tasks will be executed, and only newly arrived ones rejected, implying that earlier arriving tasks are more important. Instead, the selection of which tasks to reject or remove should be left to the designer, allowing values and importance assigned to tasks, not on their arrival times.

In this paper, we present methods for the flexible online handling of firm aperiodic tasks based on offline constructed schedules for time triggered systems. It is based on slot shifting, [1], a method to combine offline and online scheduling by utilizing unused resources. We provide an EDF based acceptance test of $O(N)$ to determine the feasible inclusion of aperiodic tasks into the offline scheduled tasks. The algorithm avoids explicit runtime handling of resource reservations for the guaranteed tasks and their impact on the offline scheduled tasks. Therefore, flexible schemes for rejection and removal of tasks and aperiodic overload handling can be applied. The resources of tasks completed earlier can be reclaimed for other aperiodic tasks without further provisions or explicitly freeing them.

Aperiodic task handling has been studied extensively for many scheduling schemes. Server based algorithms, for example, have been presented for earliest deadline, e.g., [2], and fixed priority systems, e.g., [3]. Example algorithms for the selection of tasks to reject in overload situations have been discussed in [4], [5],[6], [7]. These algorithms assume control over all tasks in the system and do not take into account the impact of offline scheduled tasks. The algorithm presented in [1] for guarantees of single firm aperiodic tasks on offline schedules does not provide for removal of guaranteed tasks.

The rest of the paper is organized as follows: section 3.2 presents slot shifting, the method we use to combine offline and online scheduling. We introduce

the motivation and basic idea of our algorithm in section 3.3, and the detailed description in section 3.4. An example in section 3.5 illustrates the algorithm. Section 3.6 concludes the paper and gives an outlook for further work.

3.2 Slot Shifting: Flexibility for Time Triggered Systems

The rigid offline scheduling schemes used for time triggered systems do not provide for flexibility. Including non periodic tasks in the offline schedule can be impossible for aperiodics, i.e., without any knowledge or restriction on arrival times. At runtime, aperiodic tasks can only be included into the unused resources of the offline schedule, supporting neither guarantees nor short response times.

In this section, we briefly describe the slot shifting method [1] which we use as a basis to combine offline and online scheduling. It provides for the efficient handling of aperiodic tasks on top of a distributed schedule with general task constraints. Slot shifting extracts information about unused resources and leeway in an offline schedule and uses this information to add tasks feasibly, i.e., without violating requirements on the already scheduled tasks. A detailed description can be found in [1].

First, a standard offline scheduler, e.g., [8], or [9] creates scheduling tables for the periodic tasks. It allocates tasks to nodes and resolves precedence constraints by ordering task executions.

Start-times and deadlines The scheduling tables list fixed start- and end times of task executions, eliminating all flexibility. The only assignments fixed by the specification of the tasks' feasibility, however, are the initiating and concluding tasks in the precedence graph, and, as we assume message transmission times to be fixed here[1], tasks sending or receiving inter-node messages. These are the only fixed start-times and deadlines, all others are calculated recursively during offline preparations. The execution of all other tasks may vary within the precedence order, i.e., they can be shifted.

Intervals and spare capacities The deadlines of tasks are then sorted for each node and the schedule is divided into a set of *disjoint execution intervals*

[1] We apply the same mechanisms to the network as well, i.e., shifting messages, as detailed in [1].

for each node. Spare capacities 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.

The spare capacities of an interval I_i are calculated as given in formula 3.1:

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

The length of I_i minus the sum of the activities assigned to it is the amount of idle times 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.

After determination of intervals and spare capacities, the offline preparations are completed and the amount and location of unused resources is available for online use.

Online scheduling Online 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. Soft aperiodic tasks, i.e., without deadline, can be executed immediately if $sc(I_c) > 0$. After each scheduling decision, the spare capacities of the affected intervals are updated.

3.3 Motivation and Approach

Guaranteeing and handling of firm aperiodic tasks involves three steps:

Acceptance test: Upon arrival of a firm aperiodic task, a test determines whether there are enough resources available to include it feasibly in the set of previously guaranteed tasks and if the scheduling strategy will ensure timely completion.

Reservation of resources: If the task can be accepted, it is guaranteed by providing a mechanism which ensures that the resources it requires will be available for its execution. This can be achieved, e.g., by removing these resources from the available ones, or by ensuring that subsequent guarantees will not remove them. Note that acceptance test and guarantee can be separated.

Rejection strategy: A failed acceptance test indicates an overload situation. The common response, not to guarantee the task under consideration, assumes that already guaranteed tasks are more important than newly arriving ones. This is, however, not generally the case. Rather, the importance order of the tasks is independent of their arrival time. Consequently, a rejection strategy is required, which determines which task or tasks – out of all guaranteed or newly arrived tasks – to reject or abort.

3.3.1 Shortcommings of Previous Version

The original version of slot shifting provides an online guarantee algorithm of $O(N)$ as well: upon arrival of a firm aperiodic task A, the spare capacities up to its deadline are summed up and compared to the execution time demand. If A is accepted, the spare capacities and intervals are recalculated taking into account that resources needed for A are not available for other tasks. If the deadline of task A does not overlap with one of offline calculated intervals, then the interval that contains $dl(A)$ needs to be split. The details how the online slot shifting guarantee algorithm works can be found in [1].

However, we want to avoid the creation of new intervals, in order to keep the online mechanism as simple as possible. While this algorithm guarantees single aperiodics, it has limited flexibility: only the task currently tested is possibly rejected; once guaranteed, aperiodics will execute. Changes in the set of guaranteed tasks require costly deletion of intervals, recalculation of spare capacities, and new guarantees. Thus, flexible schemes for rejections, removal of guaranteed tasks, and overload handling induce prohibitively high overhead.

3.3.2 Basic Idea

The new method presented here separates acceptance and guarantee. It eliminates the online modficiation of intervals and spare capacities and thus allows rejection strategies over the entire aperiodic task set.

The basic idea behind the method is based on standard earliest deadline first guarantee, but sets it to work on top of the offline schedule: EDF is based on having full availability of the CPU; we have to consider interference from offline scheduled tasks and pertain their feasibility.

Assume, at time t_1, we have a set of guaranteed aperiodic tasks $\mathcal{G}_{t_1}$ and an offline schedule represented by offline tasks, intervals, and spare capacities. At time $t_2, t_1 < t_2$, a new aperiodic A arrives. Meanwhile, a number of tasks of $\mathcal{G}_{t_1}$ may have executed; the remanining task set at t_2 is denoted $\mathcal{G}_{t_2}$. We test if

$A \cup \mathcal{G}_{t_2}$ can be accepted, considering offline tasks. If so, we add A to the set of guaranteed aperiodics. No explicit reservation of resources is done, which would require changes in the intervals and spare capacities. Rather, resources are guaranteed by accepting the task only if it can be accepted *together* with the previous guaranteed and offline scheduled ones. This enables the efficient use of rejection strategies.

3.4 Algorithm Description

Let $\mathcal{G}_{t_1}$ denote a set of guaranteed firm aperiodic tasks at time t_1.

$$\mathcal{G}_{t_1} = \{G_i \mid c_{t_1}(G_i) > 0 \wedge t_1 < dl(G_i) \leq dl(G_{i+1})\}$$

where $c_{t_1}(G_i)$ denotes the remaining execution time of task G_i at time t_1, and $dl(G_i)$ is its absolute deadline. We keep track of how much each task has executed, which means we know the remaining execution times of each task at any time. If a guaranteed task has not yet started to execute, the remaining execution time is equal to its actual execution time, i.e., $c_{t_1}(G_i) = c(G_i)$. Tasks in $\mathcal{G}_{t_1}$ are ordered by increasing deadlines, meaning that task G_i has earlier deadline than task G_{i+1}. We also know that each task in $\mathcal{G}_{t_1}$ has a deadline later than t_1.

Now assume a new aperiodic task A arrives at time t_2, with the execution time $c(A)$ and absolute deadline $dl(A)$. From time t_1 to t_2, some tasks in $\mathcal{G}_{t_1}$ could have executed up to t_2, which is reflected as follows:

- $\{G_1, ..., G_{k-1}\}$ – tasks completed by t_2:

  $$\{G_i \in \mathcal{G}_{t_1} \mid c_{t_2}(G_i) = 0,\ 1 \leq i \leq k-1\}$$

  where $c_{t_2}(G_i)$ denotes the remaining execution time of G_i at time t_2.

- G_k – task currently ready to run, according to EDF. It may have executed partially before, so we need only to consider its remaining execution time, $c_{t_2}(G_k) \leq c(G_k)$.

- $\{G_{k+1}, ..., G_n\}$ – not yet started tasks that need to execute fully:

  $$\{G_i \in \mathcal{G}_{t_1} \mid c_{t_2}(G_i) = c(G_i),\ k+1 \leq i \leq n\}$$

  where n is the number of tasks in $\mathcal{G}_{t_1}$.

So, when guaranteeing a new aperiodic task A at time t_2, we need not consider already completed tasks, but only the remaining portion of the current task and the tasks that have not started yet:

$$\mathcal{G}_{t_2} \subset \mathcal{G}_{t_1},\ \mathcal{G}_{t_2} = \{G_k, G_{k+1}, ...G_n\}$$

A new aperiodic task A is accepted if the set $\mathcal{G}' = \mathcal{G}_{t_2} \cup A$ is feasible, considering the offline scheduled and guaranteed tasks.

3.4.1 Acceptance Test for a Set of Aperiodic and Offline Scheduled Tasks

Spare capacities and intervals of slot shifting make sure that all offline scheduled tasks are guaranteed to complete before their deadlines. Those offline tasks are scheduled to execute as late as possible, but under run-time they can be executed earlier, i.e., we can shift their execution within their feasibility window.

Aperiodic tasks use unused resources in the offline schedule. The amount and location of available resources are represented as intervals and spare capacities. So, we want to insert aperiodic tasks without violating the feasibility of offline tasks.

Let $\mathcal{A} = \{T_1, T_2, ..., T_n\}$ be a set of aperiodic tasks that need to be scheduled together with the offline tasks. We accept the aperiodic set if each task in $\mathcal{A}$ is guaranteed to complete before its deadline, i.e., the following must hold:

$$\forall i, 1 \leq i \leq n:\ c(T_i) \leq \begin{cases} \text{sc}[t, dl(T_1)] & , i = 1 \\ \text{sc}[ft(T_{i-1}), dl(T_i)] & , i > 1 \end{cases}$$

where t is current time and notation $sc[t_1, t_2]$ means the spare capacity from time t_1 to time t_2. Otherwise, we need to reject some task(s).

Note that the spare capacities are not distributed in a uniform way throughout the schedule. Rather, as described in 3.2, the schedule is divided into intervals, each with an individual value of spare capacity. Consequently, the amount of spare capacity in a window depends on the position of that window in the schedule.

The finishing time of a firm aperiodic task T_i is calculated with respect to the finishing time of the previous task, T_{i-1}. Without any offline tasks, it is calculated the same as in EDF algorithm:

$$ft(T_i) = ft(T_{i-1}) + c(T_i)$$

Since we guarantee firm aperiodic tasks on the top of an offline schedule, we need to consider the feasibility of offline tasks. This extends the formula above with a new term that reflects the amount of resources reserved for offline tasks:

$$ft(T_i) = c_t(T_i) + \begin{cases} t + \text{R}[t, ft(T_1)] & , i = 1 \\ ft(T_{i-1}) + \text{R}[ft(T_{i-1}), ft(T_i)] & , i > 1 \end{cases}$$

where $R[t_1, t_2]$ stands for the amount of resources (in slots) reserved for the execution of offline tasks from time t_1 to time t_2. We can access $R[t_1, t_2]$ via spare capacities and intervals at runtime:

$$R[t_1, t_2] = (t_2 - t_1) - sc[t_1, t_2]$$

As $ft(T_i)$ appears on both sides of the equation, a simple solution is not possible. Rather, we present an algorithm for computation of finishing times of firm aperiodic tasks with complexity of $O(N)$, which is further discussed in next subsection.

3.4.2 Pseudo Code

We now present the acceptance test and algorithm for finishing time calculation in pseudo code. Read the comments below in parallel:

```
1:   for(i = 1; dl(G_i) < dl(A); i++);

2:   ft = getFT(max(ft(G_i), t), c(A));
3:   if(ft ≤ dl(A)){
4:       for(j = i + 1; j < n; j++){
5:           ft = getFT(ft, c_t(G_j));
6:           if(ft > dl(G_j)) ⇒ not feasible!
         }
7:       insert(A, G_t);
     }
8:   else reject A

9:   getFT(ft_p, remc){
10:      sc_r = start(I_c) + sc(I_c) − ft_p;
11:      while (remc > sc_r){
             if(sc_r (I_c) > 0)
```

```
                    remc = remc − sc_r;
                c++;
                ft_p = start(I_c)
                sc_r = sc(I_c);
            }
12:         return (ft_p + remc);
    }
```

Comments:

1. Find the position of the *last* task in set $\mathcal{G}_t$, (G_i), that has deadline *before* $dl(A)$.

2. Get the finishing time of A based on the finishing time of its predecessor G_i, and A's execution demand.

3. If A can be finished before its deadline, then...

4. ...go through all the tasks in $\mathcal{G}_t$ with deadlines *after* $dl(A)$.

5. Get the finishing time of current task, based on the finishing time of its predecessor.

6. If any of the investigated tasks fails to complete before its deadline, that means that adding A to $\mathcal{G}_t$ would result in a set that is not feasible, i.e., not all tasks in $\mathcal{G}_t$ will complete before their deadlines. We can either reject A or some other task(s) in $\mathcal{G}_t$.

7. Otherwise, if all tasks can be finished by their deadlines even if A added, then insert A into the set of previously guaranteed firm aperiodic tasks $\mathcal{G}_t$.

8. ...otherwise, if A cannot be completed before its deadline reject it.

9. **Function**: Calculates the finishing time of a task based on predecessor task's finishing time, and the execution demand of the investigated task.

10. Calculate the remaining spare capacity of the current interval, i.e., the interval that contains the finishing time of the previous task (the earliest possible start time for current task).

11. Without offline tasks, $ft(T_i) = ft(T_{i-1}) + c(T_i)$. In the presence of offline tasks, however, we do not have all CPU time available, but only the spare capacities in intervals. This step finds the number of intervals needed to accommodate the computation time of the task. We "simulate" the execution of the investigated task by going through intervals and "filling up free slots", until the remaining execution time is exhausted.

12. Return the calculated finishing time of the current task.

The complexity of algorithm above is $O(N)$, because we go through all tasks only once, and calculate their finishing times on the way, as depicted in figure 3.1. The *for-loop* picks a task and start the *while-loop*, which calculates its finishing time by going through the intervals. Then we pick another task, and continue traversing the intervals at the point where we got interrupted by *for-loop*, and so on. We do not have any nested loops, and we always continue forward.

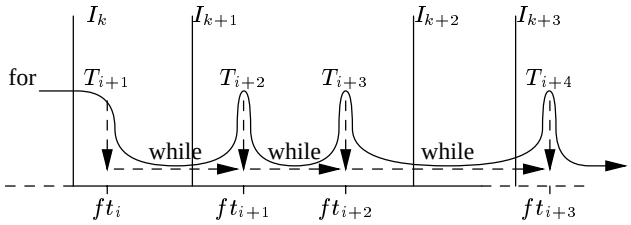


Figure 3.1: Online acceptance test for firm aperiodic tasks – algorithm complexity

3.4.3 Resource Reservation

The method presented here reserves resources implicitly, by only accepting a new task if it can be guaranteed *together* with all previously guaranteed ones. Consequently, removal of guaranteed tasks and changes in the set of tasks can be handled efficiently.

3.4.4 Rejection Strategies and Overload Handling

Our method allows for easy changes in the set of guaranteed tasks and thus supports rejection strategies and overload handling mechanisms. It allows a

new set of candidates to be submitted to the acceptance test and does not require modifications to the reserved resources for guaranteed tasks. We are currently investigating the application of overload handling schemes, such as presented in [4], [7].

3.4.5 Resource Reclaiming

Should aperiodic tasks use less resources than expressed in worst case parameters, our method directly reclaims these without recalculation of available resources. Rather, the next time the acceptance test is performed, the fact that a task has an earlier finishing time is considered in the calculations.

3.5 Example

Assume an offline schedule with intervals and spare capacities as depicted in figure 3.2 (the shaded boxes represent offline tasks). Let $\mathcal{G}_3$ be the set of previ-

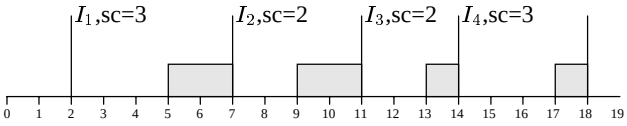


Figure 3.2: Example online firm aperiodic guarantee– static schedule

ously guaranteed but not completed firm aperiodic tasks at current time $t = 3$:

$$\mathcal{G}_3 = \{G_1(3, 10), G_2(2, 18), G_3(1, 19)\}$$

where the first parameter is the remaining execution time, the second absolute deadline. Tasks is $\mathcal{G}_3$ are ordered by increasing deadlines. At time $t = 3$ we have the execution scenario of both offline scheduled tasks and guaranteed aperiodic tasks from $\mathcal{G}_3$ as described in figure 3.3. Guaranteed firm aperiodic tasks will execute in the first available slots, in EDF order.

Now assume a firm aperiodic task $A(4, 16)$ arrives at run-time at $t = 3$. We perform the online guarantee algorithm to investigate if we can accept A:

1. Task G_1 has earlier deadline than A, so G_1's position in the set $\mathcal{G}_3$ remains unchanged, i.e., before A. We do not need to guarantee it again.

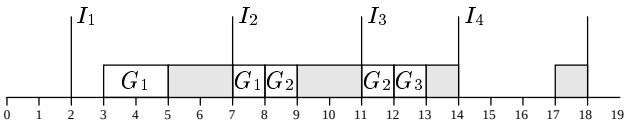
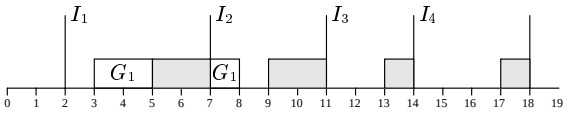


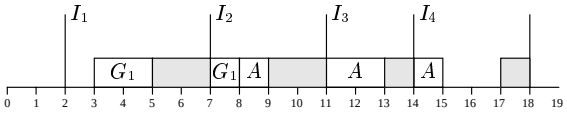
Figure 3.3: Example online firm aperiodic guarantee – execution without new task.



2. Task G_2 has a deadline after the deadline of A, which means that the A should execute before G_2. We must check if there is enough resources available for A to complete before its deadline. We calculate the finishing time of A which is slot 15:

$$ft_A = getFT(ft_{G_1}, c(A)) = 15 < 16$$

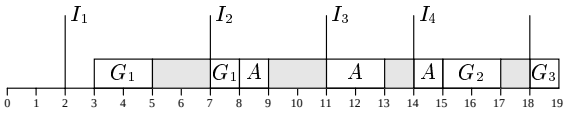
The finishing time is less than the deadline of A, which means that A could complete in time.



3. Now we must check if accepting task A will cause any of other guaranteed firm aperiodic tasks (G_2, G_3) to miss their deadlines. We calculate their finishing times:

$$\begin{aligned} ft_{G_2} = &\quad getFT(ft_A, c_3(G_2)) &= 17 < 18 \\ ft_{G_3} = &\quad getFT(ft_{G_2}, c_3(G_3)) &= 19 \leq 19 \end{aligned}$$

Both G_2 and G_3 can complete before their deadlines, which means that the new task A can be guaranteed and therefore inserted in the set of guaranteed firm aperiodic tasks $\mathcal{G}_3$.



3.6 Summary and Outlook

In this paper we presented an algorithm for the flexible handling of firm aperiodic tasks in offline scheduled systems. It is based on slot shifting, a method to combine offline and online scheduling methods.

First, a standard offline scheduler constructs a schedule, resolving complex task constraints such as precedence, distribution, and end-to-end deadlines. This is then analyzed for unused resources and leeway in task executions. The run-time scheduler uses this information to handle aperiodic tasks, shifting other task executions (”slots”) to reduce response times without affecting feasibility. We provided an $O(N)$ acceptance test for a set of aperiodic tasks on the offline schedule and guarantee tasks without explicit reservation of resources. Our method supports flexible, value based selections of tasks to reject or remove in overload situations, and simple resource reclaiming.

While the current algorithm enables the rejection and removal of tasks, it does not address the issue of selection. We are investigating into providing a number of overload handling strategies, e.g., [4], [7].

In a previous paper [10], we presented an offline test for sporadic tasks based on worst case arrival assumptions. It cannot utilize less frequent arrivals for firm aperiodic tasks since the runtime overheads to reflect the continuous changes in resource availability are prohibitively high. We are looking into applying the algorithm presented here to handle sporadic tasks at runtime.

Bibliography

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

[2] M. Spuri and G. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the Real-Time Systems Symposium*, Dec. 1994.

[3] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real- Time Systems*, 1, July 1989.

[4] G. Buttazzo and J. Stankovic. *Adding Robustness in Dynamic Preemptive Scheduling*. Kluwer Academic Publishers, 1995.

[5] G. Koren and D. Shasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In *Proceedings of the Real-Time Systems Symposium*, Dec. 1992.

[6] S. Baruah, G. Koren, D. Mao, and B. Mishra. On the competitiveness of on-line real-time task scheduling. *Real- Time Systems*, 2(4), June 1992.

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

[8] K. Ramamritham. Allocation and scheduling of complex periodic tasks. In *10th Int. Conf. on Distributed Computing Systems*, 1990.

[9] G. Fohler and C. Koza. Heuristic scheduling for distributed real-time systems. Technical Report 6/98, Institut für Technische Informatik, Technische Universität Wien, April 1989.

[10] Damir Isovic and Gerhard Fohler. Handling sporadic tasks in off-line scheduled distributed hard real-time systems. *Proc. of 11th EUROMICRO conf. on RT systems, York, UK*, June 1999.

Chapter 4

Paper C: Efficient Scheduling of Sporadic, Aperiodic, and Periodic Tasks with Complex Constraints

Damir Isovic and Gerhard Fohler
In Proceedings of the 21st IEEE Real-Time Systems Symposium, Orlando, Florida, USA, November 2000

Abstract

Many industrial applications with real-time demands are composed of mixed sets of tasks with a variety of requirements. These can be in the form of standard timing constraints, such as period and deadline, or complex, e.g., to express application specific or non temporal constraints, reliability, performance, etc. Arrival patterns determine whether tasks will be treated as periodic, sporadic, or aperiodic. As many algorithms focus on specific sets of task types and constraints only, system design has to focus on those supported by a particular algorithm, at the expense of the rest.

In this paper, we present an algorithm to deal with a combination of mixed sets of tasks and constraints: periodic tasks with complex and simple constraints, soft and firm aperiodic, and sporadic tasks. Instead of providing an algorithm tailored for a specific set of constraints, we propose an EDF based runtime algorithm, and the use of an offline scheduler for complexity reduction to transform complex constraints into the EDF model. At runtime, an extension to EDF, two level EDF, ensures feasible execution of tasks with complex constraints in the presence of additional tasks or overloads. We present an algorithm for handling offline guaranteed sporadic tasks, with minimum interarrival times, in this context which keeps track of arrivals of instances of sporadic tasks to reduce pessimism about future sporadic arrivals and improve response times and acceptance of firm aperiodic tasks.

A simulation study underlines the effectiveness of the proposed approach.

4.1 Introduction

Many industrial applications with real-time demands are composed of tasks of various types and constraints. Arrival patterns and importance, for example, determine whether tasks are periodic, aperiodic, sporadic, and soft, firm, or hard. The controlling real-time system has to provide for a *combined* set of such task types. The same holds for the various constraints of tasks. In addition to basic temporal constraints, such as periods, start-times, deadlines, and synchronization demands, e.g., precedence, jitter, or mutual exclusion, a system has to fulfill complex application demands which cannot be expressed as generally: Control applications may require constraints on individual instances [1], 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.

The choice of scheduling algorithm or paradigm determines the set of types and constraints on tasks during system design. Earliest deadline first or fixed priority scheduling, 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. Offline 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. 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 well as possible.

In this paper we present an algorithm to deal with mixed sets of tasks and constraints: periodic tasks with complex and simple constraints, soft and firm aperiodic, and sporadic tasks. Instead of providing an algorithm tailored for a specific set of constraints only, we propose an EDF based runtime algorithm, and the use of an offline scheduler for complexity reduction to transform complex constraints into the simple EDF model. So, at runtime, all tasks are constrained by start-time and deadline only, which serves as an "interface" between tasks of mixed constraints. An extension of EDF, *two level EDF* ensures the feasible execution of those tasks, whose complex constraints have been transformed offline, even in the presence of additional runtime activities and overloads. It serves as a basis for the sporadic task handling presented.

The offline transformation determines resource usage and distribution as well, which we use to handle sporadic tasks. An offline test determines and allocates resources for sporadic tasks such that worst case arrivals can be ac-

commodated at any time. At runtime, however, when a sporadic task arrives, we do not need to account for its arrival at least for its minimum inter-arrival time and reuse its allocated resources for aperiodic tasks. Our algorithm keeps track of sporadic task arrivals to update the "current worst case" and applies it for an online $O(N)$ acceptance test for aperiodic tasks.

Our methods also provides for the integration of offline and online scheduling: A complete offline schedule can be constructed, transformed into EDF tasks, and scheduled at runtime together with other EDF tasks. Thus, our method combines handling of complex constraints, efficient and flexible runtime scheduling, as well as offline and online scheduling.

A variety of algorithms have been presented to handle periodic and aperiodic tasks, e.g., [3], [4], [5]. Most concentrate on particular types of constraints. An on-line algorithm for scheduling sporadic tasks with shared resources in hard real-time systems has been presented in [6]. Scheduling of sporadic requests with periodic tasks on an *earliest-deadline-first (EDF)* basis [7] has been presented in [8]. The slot shifting algorithm to combine offline and online scheduling was presented in [9]. It focuses on inserting aperiodic tasks into offline schedules by modifying the runtime representation of available resources. While appropriate for including sequences of aperiodic tasks, the overhead for sporadic task handling becomes too high. An offline test for sporadic tasks was given in [10]. It does not, however, provide for firm aperiodic tasks, only soft ones. The method presented here is based on the offline transformation of slot shifting but provides a new runtime algorithm, in particular for efficient sporadic task handling and resource reclaiming at runtime. It handles firm and soft aperiodic tasks as well.

The use of information about amount and distribution of unused resources for non periodic activities is similar to the basic idea of slack stealing [5], [11] which applies to fixed priority scheduling. Our method applies this basic idea in the context of offline and EDF scheduling. It requires a small runtime data structure, simple runtime mechanisms, going through a list with increments and decrements, provides $O(N)$ acceptance tests and facilitates changes in the set of tasks, for example to handle overloads. Furthermore, our method provides for handling of slack of non periodic tasks, as well, e.g., instances of tasks can be separated by intervals other than periods.

The rest of this paper is organized as follows: The task model is described in section 4.2. The subsequent sections describe the different types of tasks: section 4.3 describes handling of periodic tasks, the offline complexity reduction of constraints, and a description of the extended EDF runtime scheduling method. Soft and firm aperiodic task handling is discussed in section 4.4. The

algorithm for handling sporadic task together with the other task types is presented in section 4.5. Simulation results in section 4.6 illustrate the effectiveness of the algorithm. Finally, section 4.7 concludes the paper.

4.2 Task and System Assumptions

In this paper, we distinguish between *simple* constraints, i.e., period, start-time, and deadline, for the earliest deadline first scheduling model, and *complex* constraints.

4.2.1 Complex constraints

We refer to such relations or attributes of tasks as *complex constraints*, which cannot be expressed directly in the earliest deadline first scheduling model using period, start-time, and deadline. Offline transformations are needed to schedule these at runtime with EDF. For some specific constraints such transformations have been presented, e.g., [12], [13]. Our method uses a general technique, capable of incorporating various constraints and their combinations. In the following, we list examples to illustrate and motivate the general approach.

Synchronization: Execution sequences, such as sampling - computing - actuating require a *precedence* order of task execution. An algorithm for the transformation of precedence constraints on single processors to suit the EDF model has been presented in [12]. Many industrial applications, however, demand the allocation of tasks, in particular for sensing and actuating to different processors, necessitating a distributed system with internode communication. The transformation of precedence constraints with an end-to-end deadline in this case requires subtask deadline assignment to create execution windows on the individual nodes so that precedence is fulfilled, e.g., [14]. The analysis presented in [15] focuses on schedulability analysis for pairs of tasks communicating via a network rather than the decomposition of the precedence graph.

Jitter: The execution start or end of certain tasks, e.g., sampling or actuating in control systems, is constrained by maximum variations. Strictly periodic execution can solve some instances of this problem, but over-constrains the specification. Algorithms are computationally expensive [16].

Non periodic execution: The commonly used model is periodic, i.e., instances of tasks are released at constant, periodic intervals of time. Non periodic constraints, such as some forms of jitter, e.g., for feedback loop delay in control systems [1], which require instances of tasks to be separated by *non constant* length intervals cannot be handled in this model, or have to be fitted into the periodic model at the cost of over constrained specification. Similar reasoning applies to constraints over more than one instance of a task, e.g., for iterations, data history or ages. A constraint can be of the type "separate the execution of instance i and $i + 4$ by no more than max and no less than min".

Non temporal constraints: Demands for reliability, performance, or other system parameters impose demands on tasks from a system perspective, e.g., to not allocate two tasks to the same node, or to have minimum separation times, etc.

Application specific constraints - engineering practice: Applications may have demands specific to their nature. Duplicated messages on a bus in an automotive environment, for example, may need to follow a certain pattern due to interferences such as EMI. Wiring can have length limitations, imposing allocation of certain tasks to nodes according to their geographical positions. An engineer may want to improve schedules, creating constraints reflecting his practical experience.

4.2.2 Task types

We assume all tasks in the system to be fully preemptive and to communicate with the rest of the system via data read at the beginning and written at the end of their executions.

Periodic tasks execute their invocations within regular time intervals. A periodic task T_P is characterized by its worst case execution time ($wcet$) [17], period (p) and relative deadline (dl).
The k^{th} invocation of T_P is denoted T_P^k and is characterized by its earliest start time (est) and relative deadline (dl).

We refer to a periodic task with complex constraints which have been transformed offline as an *offline task*.

Aperiodic tasks are invoked only once. Their arrival times are unknown at design time. A *firm* aperiodic task T_A has the following set of parameters: the arrival time (ar), worst case execution time and relative deadline. *Soft* aperiodic tasks have no deadline constraints.

Sporadic tasks can arrive at the system at arbitrary points in time, but with defined minimum inter-arrival times between two consecutive invocations. A sporadic task T_S is characterized by its relative deadline, minimum inter-arrival time (λ) and worst case execution time.
These attributes are known before the run-time of the system. Additional information available on-line, is its arrival time and its absolute deadline.

4.2.3 System assumptions

Distributed system: We consider a *distributed* system, i.e., one that consists of several processing and communication nodes [18]. While we allow for distributed system and distribution in the complex constraints, we handle those issues in the offline phase, i.e., at runtime, no task migration takes place.

Time model: We assume a discrete time model [19]. Time ticks are counted globally, by a synchronized clock with granularity of *slotlength*, and assigned numbers from 0 to ∞. The time between the start and the end of a slot *i* is defined by the interval $[slotlength * i, slotlength * (i + 1)]$. Slots have uniform length and start and end at the same time for all nodes in the system. Task periods and deadlines must be multiples of the slot length.

4.2.4 Task handling - overview

The following table gives an overview of when types of tasks are handled by our method.

		soft aper.	firm aper.	spo-radic	simp. per.	comp. per.
offline	sched.				x	x
	test			x		
online	sched.	x	x	x	x	x
	test	x	x			

4.3 Periodic Tasks - Offline Complexity Reduction

In this section we start describing the handling of various types of tasks, with periodic tasks with complex constraints. We will present a method for complexity reduction and online scheduling to ensure these transformed constraints.

4.3.1 Offline complexity reduction

Finding optimal solutions to most sets of complex constraints is an NP hard problem [20]. Consequently algorithms will be heuristic and produce suboptimal solutions only. Performing the complexity reduction offline, however, allows for elaborate methods, improvement of results and modifications in the non-successfull case. The transformation method should be flexible to include new types of constraints, to accommodate application specific demands and engineering requirements. A number of general methods for the specification and satisfaction of constraints can be applied for real-time tasks, e.g., [21] or [22]. Runtime scheduling has to ensure that tasks execute according to their constraints, even in the presence of additional tasks or overload situations.

We propose to use the offline transformation and online guarantee of complex constraints of the slot shifting method [9][1]. 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 [23], [9], [24]. It uses standard offline schedulers, e.g., [25], [23] to create schedules which are then analyzed to define start-times and deadlines of tasks.

First, the offline scheduler creates scheduling tables for the selected periodic tasks with complex constraints. It allocates tasks to nodes and resolves complex constraints by constructing sequences of task executions. The resulting offline schedule is a single feasible, likely suboptimal solution. These sequences consist of subsets of the original task set separated by allocation. Each task in a sequence is limited by either sending or receiving of internode messages, predecessor or successor within the sequence, or limits set by the offline scheduler. Start times and deadline are set directly to the times of internode messages or offline scheduler limits, or calculated recursively for tasks constrained only within sequences. A more detailed description can be found in [9]. The final result is a set of independent tasks on single nodes, with start-times and deadlines.

The offline scheduling algorithm we use [23] is based on heuristic search to handle complexity reduction, and provide for straightforward inclusion of

[1] We do not, however, use its runtime scheduling and handling of non periodic tasks.

additional constraints by providing an additional feasibility test. It works with precedence constraints as basic model, handles jitter constraints, and performs allocation and subtask deadline assignment. In addition to constraint transformation, the use of an offline scheduler provides for integration of offline and online scheduling as well.

4.3.2 Runtime guarantee of complex constraints

In the previous steps we created tasks with start-time and deadline constraints, which can be scheduled by EDF at runtime. The resulting feasible schedules represent the original complex constraints. Additional runtime tasks, however, can create overload situations, resulting in violations of the complex constraints. Thus, a mechanism is needed which ensures the feasible execution of these tasks, even in overload situations.

We propose an extension to EDF, *two level EDF* for this purpose. The basic idea is to schedule tasks according to EDF - "normal level", but give priority -"priority level" to an offline task when it needs to start at latest, similar to the basic idea of slack stealing [5] [11] for fixed priority scheduling. Thus, the CPU is not completely available for runtime tasks, but reduced by the amount allocated for offline tasks. So we need to know amount and location of resources available after offline tasks are guaranteed. Runtime efficiency demands simple runtime data structure and runtime maintenance.

Offline preparations After offline 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* 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. 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 spare capacities, as defined here, 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, or earlier (empty interval). Thus it is possible that a task executes outside its interval, i.e., earlier than the interval start, but not before its earliest starttime.

The spare capacities of an interval I_i are calculated as given in formula 4.1:

$$sc(I_i) \;=\; |I_i| - \sum_{T \in I_i} wcet(T) + min(sc(I_{i+1}), 0) \tag{4.1}$$

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 [23] for details.

Thus, we can represent the information about amount and distribution of free resources in the system, plus online constraints of the offline 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.

Online execution 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 - "normal level". $sc(I_c) = 0$ indicates that a guaranteed task has to be executed or else a deadline violation in the task set will occur. It will execute immediately - "priority level". Since the amount of time spent at priority level is known and represented in spare capacity, 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 offline task assigned to I_c executes spare capacity does not change. If an offline 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 ensures 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 [9] for more details.

4.4 Aperiodic Tasks

A first verion of slot shifting presented an algorithm to guarantee aperiodic tasks by inserting them into an offline schedule. Once guaranteed, the resources allocated for the aperiodic were removed by creating a new interval and adjusting spare capacity. While efficient for guaranteeing sequences of aperiodic tasks without removal, the runtime overhead for handling sporadic tasks efficiently is too high. Further, changes in the set of guaranteed tasks require costly deletion of intervals, recalculation of spare capacities, and new guarantees. Thus, flexible schemes for rejections, removal of guaranteed tasks, and overload handling induce prohibitively high overhead.

The new method presented here separates acceptance and guarantee. It eliminates the online modificiation of intervals and spare capacities and thus allows rejection strategies over the entire aperiodic task set.

4.4.1 Acceptance test

The basic idea behind our method is based on standard earliest deadline first guarantee, but sets it to work on top of the offline tasks. EDF is based on having full availability of the CPU, so we have to consider interference from offline scheduled tasks and pertain their feasibility.

Assume, at time t_1, we have a set of guaranteed aperiodic tasks $\mathcal{G}_{t_1}$ and an offline schedule represented by offline tasks, intervals, and spare capacities. At time $t_2, t_1 < t_2$, a new aperiodic A arrives. Meanwhile, a number of tasks of $\mathcal{G}_{t_1}$ may have executed; the remaining task set at t_2 is denoted $\mathcal{G}_{t_2}$. We test if $A \cup \mathcal{G}_{t_2}$ can be accepted, considering offline tasks. If so, we add A to the set of guaranteed aperiodics. No explicit reservation of resources is done, which would require changes in the intervals and spare capacities. Rather, resources are guaranteed by accepting the task only if it can be accepted *together* with the previous guaranteed and offline scheduled ones. This enables the efficient use of rejection strategies.

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 offline tasks, it is calculated the same way as in the EDF algorithm:

$$ft(A_i) = ft(A_{i-1}) + c(A_i)$$

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

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

where $R[t_1, t_2]$ stands for the amount of resources (in slots) reserved for the execution of offline tasks from time t_1 to time t_2. We can access $R[t_1, t_2]$ via spare capacities and intervals at runtime:

$$R[t_1, t_2] = (t_2 - t_1) - \sum_{I_c \in (t1, t2)} max(sc[I_c], 0)$$

As $ft(A_i)$ appears on both sides of the equation, a simple solution is not possible. Rather, we present an algorithm for computation of finishing times of hard aperiodic tasks with complexity of $O(N)$.

4.4.2 Algorithm

Let A_i be a firm aperiodic task we want to guarantee. Let $\mathcal{G}$ denote the set of previously guaranteed but not yet completed firm aperiodic tasks, such as each task in $\mathcal{G}$ has a deadline later than or equal to $dl(A)$:

$$G = \{A_j \mid i < j < n, dl(G_j) \geq dl(A_i)\}$$

Here is the pseudo code for the acceptance test and algorithm for finishing time calculation:

$ft = getFinishingTime(max(ft(A_{i-1}), t), c(A_i));$
/* check if accepting A_i causes any of the previously
guaranteed firm aperiodic tasks to miss its deadline */
$if(ft \leq dl(A_i))\{$
    $for(j = i + 1; j < n; j++)\{$
        $ft = getFinishingTime(ft, remc(G_j));$
        $if(ft > dl(G_j))$ $\Rightarrow$ not feasible!

```
        }
        insert(A, G);
}
else reject A

getFinishingTime(ft_p, remc){
    /* determine ft by "filling up"
    free slots until the c is exhausted. */
    sc_r = start(I_c) + sc(I_c) − ft_p;
    while (remc > sc_r){
            if(sc_r (I_c) > 0)
                    remc = remc − sc_r;
            c + +;
            ft_p = start(I_c)
            sc_r = sc(I_c);
    }
    return (ft_p + remc);
}
```

$remc$ = remaining execution time
ft_p = the finnishing time of predecessor task

The complexity of algorithm is $O(N)$, because we go through all tasks only once, and calculate their finishing times on the way. More detailed description of the algorithm can be found in [24].

4.5 Sporadic Tasks

In the previous section we described how firm aperiodic tasks are guaranteed online assuming no sporadic tasks in the system. Now we will see how sporadic tasks can be included in the aperiodic guarantee.

We will discuss ways to handle sporadic tasks with periodic tasks with complex constraints. We present a new algorithm which keeps track of sporadic task arrivals and reduces pessimism about possible future arrivals to improve guarantees and response times of aperiodic tasks.

4.5.1 Handling sporadic tasks

Pseudo-periodic tasks – Sporadic tasks can be transformed offline into pseudo-periodic tasks [26] which can be scheduled simply at runtime. The overhead induced by the method, however, can be very high: in extreme cases, a task handling an event which is rare, but has a tight deadline may require reservation of all resources.

Offline test – In an earlier paper [10], we have presented an offline test for sporadic tasks on offline tasks. It ensured that the spare capacity available was sufficient for the worst case arrival of sporadic tasks without reflecting it in the spare capacity. Consequently, firm aperiodic tasks cannot be handled at runtime.

Offline test and online aperiodic guarantees – A better algorithm will perform the offline test and decrease the needed resources from spare capacity. The resulting pessimism can be reduced by reclaiming a slot upon non arrival of a sporadic task. Aperiodic guarantee will be possible, but have to consider worst case arrival patterns of the sporadic tasks at any time.

Offline test and online aperiodic guarantees and reduced pessimism – The algorithm presented here performs the offline test, but does not change intervals and spare capacity for runtime efficiency. At runtime, its keeps track of sporadic arrivals to reduce pessimism, by removing sporadic tasks from the worst case arrival which are known to not arrive up to a certain point. An aperiodic task algorithm utilizes this knowledge for short response times.

4.5.2 Interference window

We do not know when a sporadic task $S_i \in \mathcal{S}$, will invoke its instances, but once an instance of S_i arrives, we do know the minimum time until the arrival of the next instance — the minimum inter-arrival time of S_i. We also know the worst case execution time of each sporadic task in $\mathcal{S}$. We use this information for the acceptance test of firm aperiodic tasks.

Assume a sporadic task S_i invokes an instance at time t (see figure 4.1). Let S_i^k denote current invocation of S_i, and S_i^{k+1} the successive one. At time t we know that S_i^{k+1} will arrive no sooner than $t + \lambda$, where λ is the minimum inter-arrival time of S_i. So, when S_i^k has finished its execution, S_i will not influence any of the firm aperiodic tasks at least until S_i^{k+1} arrives. This means that,

when calculating the amount of resources available for a firm aperiodic task with an execution that intersects with S_i's execution window, we do not need to take into account the interference from S_i at least between the finishing time of its current invocation, S_i^k, and the start time on the next invocation, S_i^{k+1}, as depicted in figure 4.1.

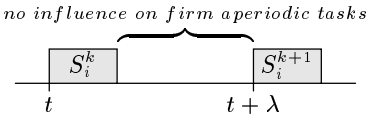


Figure 4.1: A sporadic task.

Let $EW(A_i)$ denote the execution window of A_i, i.e., the interval between A_i's arrival and its deadline:

$$EW(A_i) = [ar(A_i), dl(A_i)],\ |EW| = dl(A_i) - ar(A_i)$$

Now we will see how the execution of a previously guaranteed sporadic task $S_i \in \mathcal{S}$ can influence A_i's guarantee.

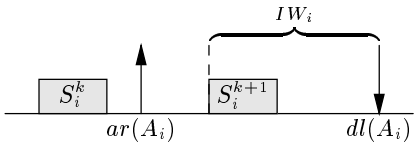
Let S_i^k denote the current invocation of S_i, i.e., the last invocation of S_i before A_i arrived. Let IW_i be the *interference window* of S_i, that is the time interval in which S_i may preempt and interfere with the execution of A_i. The following cases can be identified:

case 1: S_i^k is unknown, i.e., the sporadic task S_i has not started yet to invoke its instances. S_i can arrive any time and we must assume the worst case, that is S_i will start to invoke its instances at *t*, with maximum frequency. The interference window is the entire execution window of A_i, $IW_i = EW$.

case 2: S_i^k is known, i.e., S_i has invoked an instance before t. The following sub-cases can occur:

- **a)** $start(S_i^k) + \lambda \leq t$, i.e., the last invocation completed before A_i arrived, and the next invocation, S_i^{k+1}, could have arrived but it has not yet. This means S_i^{k+1} can enter A_i's execution window at any time, thus the same as in case 1, $IW_i = EW$.
- **b)** $end(S_i^k) \leq t < start(S_i^k) + \lambda$, i.e., the current invocation S_i^k has completed before *t*, and the next one has not arrived yet. But now

we know that the next one, S_i^{k+1} will not arrive until λ time slots, counted from the start time of S_i^k.



This means the interference window can be decreased with the amount of slots in EW for which we know that S_i^{k+1} cannot possibly arrive:

$$IW_i = [start(S_i^k) + \lambda, dl(A_i)]$$

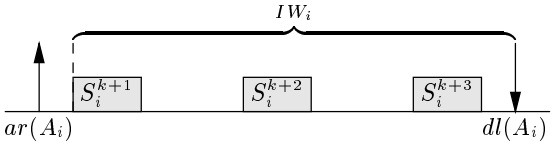
c) $t < end(S_i^k)$, i.e., the current invocation is still executing. In the worst case, the interference window is entire EW, $IW_i = EW$.

The processor demand approach, [27], can be used to determine the total processing time needed for all sporadic tasks, $c(\mathcal{S})$, which will invoke their instances within their interference windows:

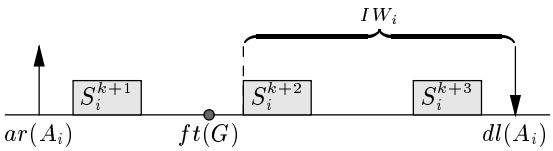
$$c(S) = \sum_{i=1}^{n} \left\lceil \frac{|IW_i|}{\lambda(S_i)} \right\rceil c(S_i) \tag{4.2}$$

Now we will see how the interference window can actually get "shrunk" when guaranteeing a firm aperiodic task A under runtime. It is usually not the case that A will start to execute as soon it arrives. This because of the offline tasks and previously guaranteed firm aperiodic tasks. In section 4.4, we presented a method for guaranteeing firm aperiodic tasks on top of offline tasks. The start time of the firm aperiodic task which is currently tested for acceptance is based on the finishing time of its predecessor, i.e., another firm aperiodic task with earlier deadline. Hence, in some cases the start of the interference window IW_i is set to the finishing time of A's predecessor.

Here is an example: assume a firm aperiodic task A to be accepted and a sporadic task S_i as in *case 2b* above. The interference window is defined as below:

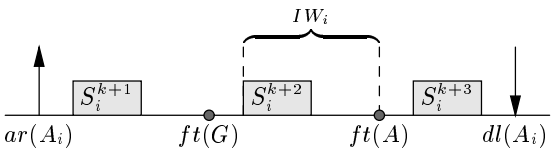


Assume another previously accepted firm aperiodic task G which will delay the execution of A:



We see that the earliest time A can start is set to the finishing time of its predecessor, $ft(G)$. So, all invocation of S_i that occured before $est(A)$ are taken care of when calculating $ft(G)$, and are not concidered when calculating $ft(A)$. The start of the interference window is now set to the start time of the first possible instance of S_i that can interfere with A, that is S_i^{k+2}.

Now we calculate the finishing time of A using the algorithm described in section 4.4. Without sporadic tasks, A would be guaranteed to finish at time $ft(A)$. Since A is guaranteed to finish before its deadline, we do not need to take into consideration the impact from S_i after the finishing time of A. Hence, the end of the interference window IW_i is set to $ft(A)$.



So, what actually happens is that only one instance of S_i is concidered when calculating $ft(A)$.

Now we can formalize what the impact of a sporadic task S_i on a firm aperiodic task A_i: Let A_{i-1} be the predecessor of A_i, i.e., the last firm aperiodic task that is guaranteed to execute before A_i (task G in example above). If S_i has not yet started to invoke its instances at the time we start with the acceptance test for A_i, we must assume the worst case, that is the first instance of S_i will start at the same time as the earliest start time of A_i:

$$est(S_i^1) = est(A_i) = max(t, ft(A_{i-1}))$$

We have *max* because A_{i-1} could have completed before the current time t, or A_i has no predecessor at all.

On the other hand, if S_i has started to invoke its instances, we can calculate when the next one after the earliest possible time of A_i can occur (S_i^{k+2} in example above):

$$est(S_i^{k+m}) = est(S_i^k) + \left\lceil \frac{ft(A_{i-1}) - est(S^k)}{\lambda(S_i)} \right\rceil \lambda(S_i)$$

To conclude, the time interval IW_i in which a sporadic task S_i may preempt and interfere with the execution of a firm aperiodic task A_i is obtained as:

$$IW_i = [\delta, ft(A_i)] \tag{4.3}$$

where δ is the earliest possible time S_i could preempt A_i and is calculated as:

$$\delta = \begin{cases} est(S_i^{k+m}) & \text{if } S_i \text{ known} \\ max(t, ft(A_{i-1})) & \text{otherwise} \end{cases} \tag{4.4}$$

The index $k+m$ points out the first possible invocation of S_i which has earliest start time after the finishing time of A_i's predecessor.

4.5.3 Algorithm description

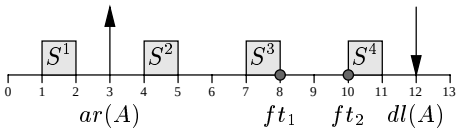
Assume a firm aperiodic task A_i that is tested for acceptance upon its arrival time, current time t. We want to make sure that A_i will complete before its deadline, without causing any of the guaranteed tasks to miss its deadline. A guaranteed task is either an offline task, a previously guaranteed firm aperiodic task or a sporadic task. Offline and sporadic tasks are guaranteed before the run-time of the system, [9], [10], while firm aperiodic tasks are guaranteed online, upon their arrival. The guarantee algorithm is performed as follows:

step 1: Assume no sporadic tasks and calculate the finishing time of A_i based only on offline tasks and previously guaranteed firm aperiodic tasks (as described in section 4.4).

step 2: Calculate the impact from all sporadic tasks that could preempt A_i before its finishing time calculated in the previous step (equation 4.2).

step 3: If the impact is greater than zero, the finishing time of A_i will be postponed (moved forward), because at run-time we need to execute all sporadic instances with deadlines[2] less than $dl(A_i)$. The impact reflects the amount of A_i that is to be executed after the finishing time calculated in step 1. Now we treat the remaining part of A_i as a firm aperiodic task and repeat the procedure (go to step 1). But this time we start calculating the sporadic impact at the finishing time of the first part of A_i. The procedure is repeated until there is no impact from sporadic tasks on A_i.

Example: assume a firm aperiodic task A which arrives at current time $t = 3$, with the execution demand $c(A) = 5$ and deadline $dl = 12$. Also assume a sporadic task S that has started to invoke its instances before t, in slot 1, with a minimum inter-arrival time $\lambda = 3$ and worst case computation time $c(S) = 1$. For simplicity reasons, assume no offline tasks and no previously guaranteed hard aperiodic tasks. First we calculate the finishing time of A, without considering the sporadic task S, i.e., $ft_1 = 8$.



The interference window of A is $IW_i = [4, 8]$. The impact of S in IW_i is equal to 2 (two instances). Now we take the impact (which tells us how much A is delayed by S) and calculate its finishing time, starting at time $t_1 = ft_1$, i.e., $ft_2 = 10$. We must check if we have any sporadic instances in the new interference interval $IW_i^{'} = [10, 10]$ (note that original $IW_i^{'}$ would be $[8, 10]$, but we always take the start time of the next instance after the previous finishing time, in this case $est(S^4) = 10$). The new impact is zero, which means that we can stop and the last calculated finishing time, $ft_2 = 10$, is the finishing time of A.

Implementation The first part of the algorithm is exactly the same as described in 4.4: first we locate the position of hard aperiodic task to be guaranteed, calculate its finishing time and check if any of previously guaranteed hard aperiodic tasks will miss its deadline. The second part, that calculates

[2]The deadline of a sporadic instance is set to the earliest start time of the next instance

the finishing time, is extended to handle the impact from the sporadic tasks as follows:

$getFinishingTime(ft_{pred}, c)\{$
/*determine ft without sporadics by "filling up"
free slots until the c is exhausted.*/
$\forall S_i \in \mathcal{S}$
if S_i started to invoke
$\delta = est(S_i^{k+m})$ /*eq (4.4)*/
else
$\delta = max(t, ft_{pred})$
$IW_i = [\delta, ft]$ /*eq (4.3)*/
$sum = sum + \left\lceil \frac{|IW_i|}{\lambda(S_i)} \right\rceil c(S_i)$ /*eq (4.2)*/
if $sum \neq 0$
$getFinishingTime(ft, sum)$
else
return ft
$\}$

The recursive formulation was chosen for simplicity of explanation: our implemtentation uses a loop. In the loop, time is increased from current to finish time, without going back. Thus the complexity remains linear, similar to the finishing time algorithm in 4.4.

4.6 Simulation Analysis

We have implemented the described mechanisms and have run simulations with the RTSim simulator [28] for various scenarios. We have tested the acceptance ratio for firm aperiodic tasks with the methods to handle sporadic tasks described in 4.5: no sporadic tasks for reference purposes ("no sporadics"), worst case arrivals without knowledge about invocations ("no info"), and updated worst case with arrival info ("updated"). We randomly generate offline tasks, sporadic and aperiodic task loads. The results were obtained for an offline scheduled task load of 0.5 and schedule length of 100 slots. We studied the acceptance ratio AR of the randomly arriving aperiodic tasks under randomly generated arrival patterns for the sporadic tasks. The worst case sporadic load, i.e., all sporadic tasks arriving with maximum frequency was set to 0.2. The arrival frequencies of sporadic tasks were set according to a factor, F_{fact}, such

that $interarrivaltime = minimum\ interarrivaltime \times F_{fact}$. Deadlines for firm aperiodic tasks were generated randomly within one schedule length. The maximum load demanded by the aperiodic tasks is 0.44.

Each point represents a sample size of some 1000 tests. 0.95 confidence intervals were smaller than 5%. We can see that our method improves the accep-

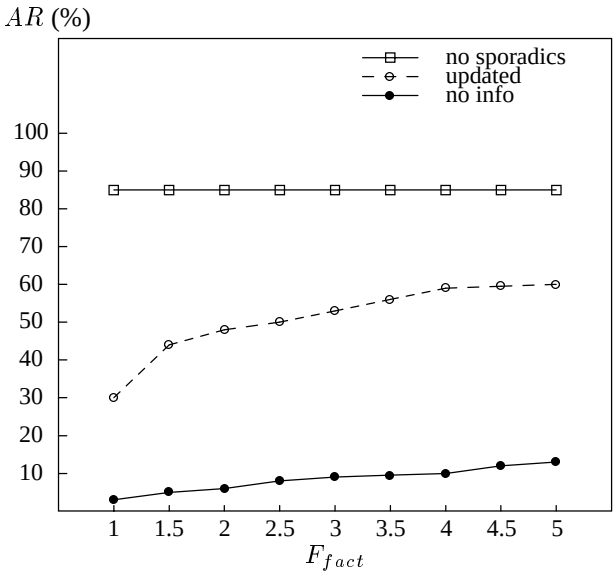


Figure 4.2: Guarantee Ratio for Firm Aperiodic Tasks

tance ratio of firm aperiodic tasks. This results from the fact that our methods reduces pessimism about sporadic arrivals by keeping track of arrivals.

4.7 Summary and Outlook

In this paper we have presented methods to schedule sets of mixed types of tasks with complex constraints, by using earliest deadline first scheduling and offline complexity reduction. In particular, we have proposed an algorithm to handle sporadic tasks to improve response times and acceptance of firm aperiodic tasks.

We have presented the use of an offline scheduler to transform complex constraints of tasks into starttimes and deadlines of tasks for simple EDF runtime scheduling. We provided an extension to EDF, two level EDF, to ensure feasible execution of these tasks in the presence of additional tasks or overloads. During offline analysis our algorithm determines the amount and location of unused resources, which we use to provide $O(N)$ online acceptance tests for firm aperiodic tasks. We presented an algorithm for handling offline guaranteed sporadic tasks, which keeps track of arrivals of instances of sporadic tasks at runtime. It uses this updated information to reduce pessimism about future sporadic arrivals and improve response times and acceptance of firm aperiodic tasks. Results of simulation study show the effectiveness of the algorithms.

Future research will deal with extending the algorithm to include interrupts, overload handling, and aperiodic and sporadic tasks with complex constraints as well. We are studying the inclusion of server algorithms, e.g., [29] into our scheduling model by including bandwidth as additional requirement in the offline transformation.

4.8 Acknowledgements

The authors wish to express their gratitude to Tomas Lennvall, Radu Dobrin, and Joachim Nilsson for useful discussions and to the reviewers for their helpful comments. The work presented in this paper was partly supported by the Swedish Science Foundation via ARTES.

Bibliography

[1] M. Törngren. Fundamentals of implementing real-time control applications in distributed computer systems. *Real-Time Systems*, 1997.

[2] V. Yodaiken. Rough notes on priority inheritance. Technical report, New Mexico Institut of Mining, 1998.

[3] M. Spuri, Giorgio C. Buttazzo, and F. Sensini. Robust aperiodic scheduling under dynamic priority systems. *In Proc. of the IEEE RTSS*, Dec. 1995.

[4] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems Journal*, 1(1):27–60, June 1989.

[5] S. R. Thuel and J.P. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *Proceddings of the Real-Time Symposium*, pages 22–33, San Juan, Puerto Rico, Dec. 1994.

[6] K. Jeffay. Scheduling sporadic tasks with shared resources in hard real-time systems. *Dept. of Comp. Sci., Univ. of North Carolina at Chapel Hill*, 1992.

[7] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in hard real-time environment. *Journ. of the ACM, 20, 1*, Jan. 1973.

[8] T. Tia, W.S. Liu, J. Sun, and R. Ha. A linear-time optimal acceptance test for scheduling of hard real-time tasks. *Dept. of Comp. Sc., Univ. of Illinois at Urbana-Champaign*, 1994.

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

[10] Damir Isovic and Gerhard Fohler. Handling sporadic tasks in off-line scheduled distributed hard real-time systems. *Proc. of 11th EUROMICRO conf. on RT systems, York, UK*, June 1999.

[11] R.I. Davis, K.W. Tindell, and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In *Proceddings of the Real-Time Symposium*, pages 222–231, Dec. 1993.

[12] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems Journal*, 2(3):181–194, Sept. 1990.

[13] R. Gerber, W. Pugh, and M. Saksena. Parametric dispatching of hard real-time tasks. *IEEE Transactions on Computers*, 44(3), March 1995.

[14] M. DiNatale and J.A. Stankovic. Applicability of simulated annealing methods to real-time scheduling and jitter control. In *Proceedings of Real-Time Systems Symposium*, Dec. 1995.

[15] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 50(2-3), 1994.

[16] S. Baruah, G. Buttazzo, S. Gorinsky, and G. Lipari. Scheduling periodic task systems to minimize output jitter. In *Sixth International Conference on Real-Time Computing Systems and Applications*, Dec. 1999.

[17] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *RT Systems Journal*, 1989.

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

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

[20] M.R. Garey., D.S. Johnson, B.B. Simons, and R.E. Tarjan. Scheduling unit-time tasks with arbitrary release times and deadlines. *IEEE Trans. on Soft. Eng.*, May 1981.

[21] F. Jahanian, R. Lee, and A. Mok. Semantics of modechart in real time logic. In *Proc. of the 21st Hawaii International Conference on Systems Sciences*, pages 479–489, Jan. 1988.

[22] J. Wurtz and K. Schild. Scheduling of time-triggered real-time systems. Technical report, German Research centre for Artificial Intelligence - DKFI GmbH, 1997.

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

[24] D. Isovic and G. Fohler. Online handling of hard aperiodic tasks in time triggered systems. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, June 1999.

[25] K. Ramamritham. Allocation and scheduling of complex periodic tasks. In *10th Int. Conf. on Distributed Computing Systems*, 1990.

[26] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, MIT, 1983. Report MIT/LCS/TR-297.

[27] K. Jeffay and D.L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of Real-Time Systems Symposium*, pages 212–221, Dec. 1993.

[28] G. Buttazzo, A. Casile, G. Lamastra, and G. Lipari. A scheduling simulator for real-time distributed systems. In *Proceedings of the IFAC Workshop on Distributed Computer Control Systems (DCCS '98)*, 1999.

[29] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of Real-Time Systems Symposium*, Dec. 1998.

Chapter 5

Paper D – Simulation Analysis of Sporadic and Aperiodic Task Handling

Damir Isovic and Gerhard Fohler
Technical Report , Mälardalen Real-Time Research Centre, Mälardalen University, May 2001.

Abstract

In this paper we present the simulation results for proposed algorithms to handle mixed sets of tasks and sonstraints: periodic tasks with complex and simple constraints, soft and firm aperiodics, and sporadic tasks.

5.1 Introduction

We have implemented the algorithms described in [1] and [2] and have run simulations for various scenarious.

In the first set of exepriments we simulated the online guarantee algorithm for firm aperiodic tasks, described in [1] (paper B in this thesis). We have studied the guarantee ratio for aperiodic tasks for different combinations of total system loads and aperiodic deadlines.

In the second set of exeperiments we have introduced sporadic tasks, as suggested in [2] (paper C), and have repeated the simulations for different combinations of periodic, sporadic and aperiodic tasks. We have measured the guarantee ratio of firm aperiodic tasks, depending on different scenarious for sporadic tasks. We also investigated how the variations in minimum inter-arrival times for sporadics influence aperiodic guarantee.

The simulation study underlines the effectiveness of the proposed approach.

5.2 Simulation environment

Simulations refered in previous papers were made with the RTSim simulator [3]. For the purpose of this thesis, we have developed a new simulator to provide for detailed analysis of slot shifting.

We also implemented a debuger, which provides for visual monitoring of the data structures during the simulations.

Simulations were performed in parallell on 5 different PC computers with the processor speed between 333 and 1500 MHz. Some 800 000 different interactions of sporadic, periodic and aperiodic tasks were simulated. The amount of data produced was more than 1 GB. The average size of an input file needed for one graph line was 30 MB. That would produce aproximately 50 MB of data to be analysed. The total lenght of simultions for both experiments was about 200 hours.

5.3 Experiment 1: Firm aperiodic guarantee

5.3.1 Experimental setup

For the first experiment series , we randomly generate generated offline and aperiodic task loads, so that the combined load of both periodic and aperiodic tasks was set to 10% - 100%. The deadline for the aperiodic tasks was

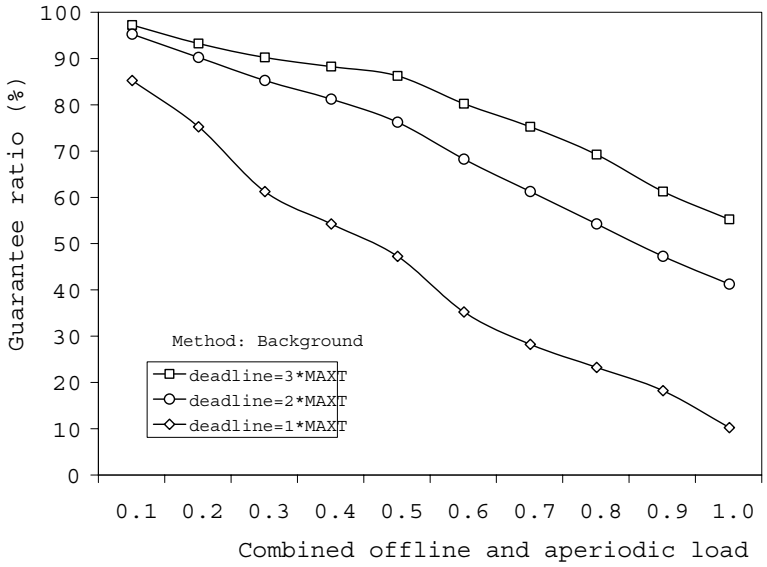


Figure 5.1: Guarantee ratio for aperiodic tasks – Background

set to their maximum execution time, MAXT, two times MAXT and three times MAXT. We studied the guarantee ratio of the randomly arriving aperiodic tasks.

The simplest method to handle aperiodic tasks in the presence of periodic tasks is to offline schedule them in background i.e., when there are no periodic instances ready to execute. The major problem with this technique is that, for high periodic loads, the response time of aperiodic requests can be too long. We compared our method to the background scheduling. We refer to our method as *Slot Shifting – Extended*, or SSE.

5.3.2 Results

In this subsection we present obtained results. Each point represents a sample size of 800-3000 simulation runs, with diffrent combinations of periodic and aperiodic tasks. 0.95 confidence intervals were smaller than 5%.

Figure 5.1 illustrates the performance of background scheduling for three different deadline settings of aperiodic tasks.

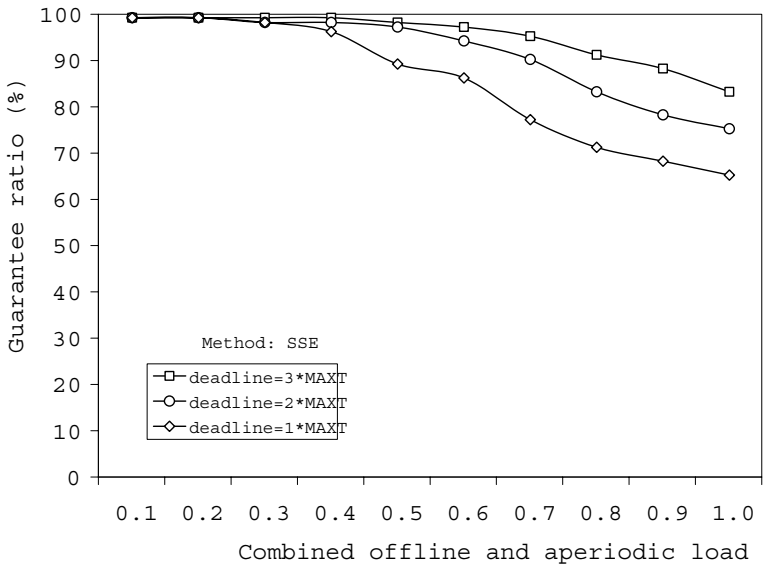


Figure 5.2: Guarantee ratio for aperiodic tasks – SSE

Figure 5.2 depicts the performance of SSE.

In figures 5.3, 5.4 and 5.5 we put both methods together, to see the difference in performance for different deadline settings.

As expected, backgound scheduling performed poorly in the high load situations, specially with tight aperiodic deadlines. For this reason, background scheduling can be adopted only when the aperiodic activities do not have stringent timing constraints and the periodic load is not high.

The graphs show the efficiency of the SSE mechanisms, as guarantee ratios are very high. As expected, the guarantee ratio for aperiodic tasks with larger deadlines is higher than for smaller deadlines. Even under very high load, guarantee ratios stay high.

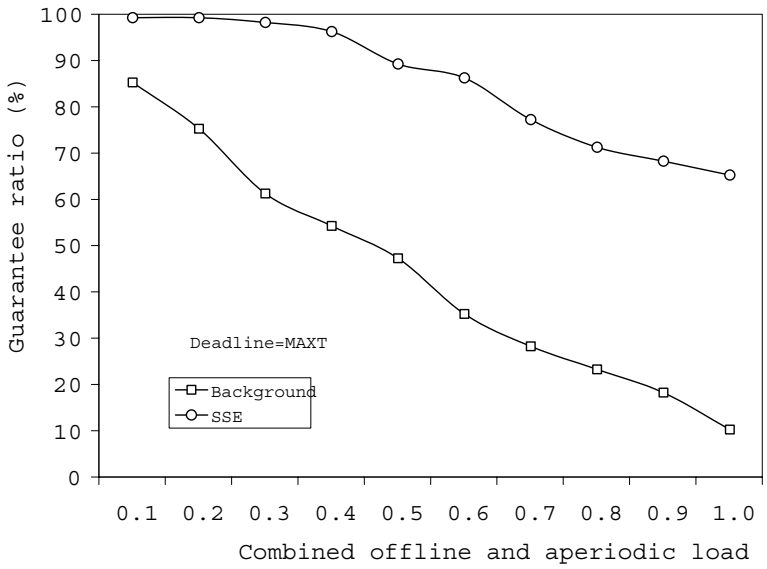


Figure 5.3: Guarantee ratio for aperiodic tasks, dl=MAXT – SSE vs Background

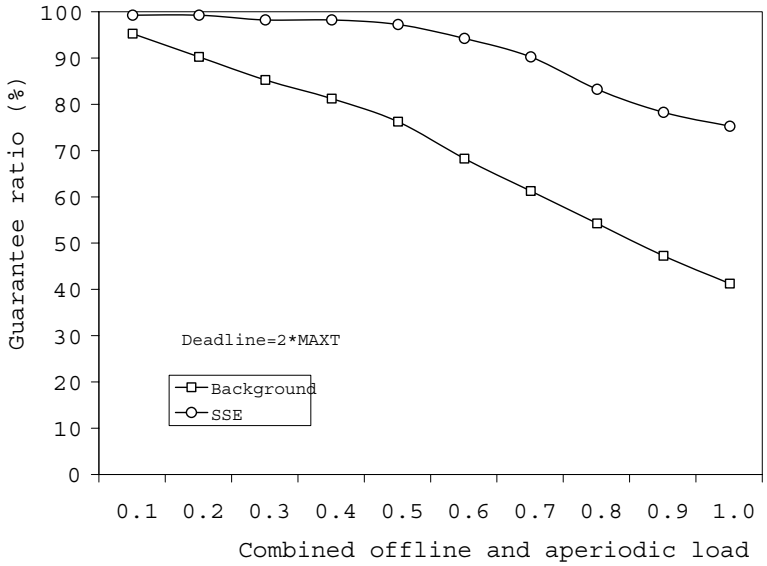


Figure 5.4: Guarantee ratio for aperiodic tasks, dl=2*MAXT – SSE vs Bgr

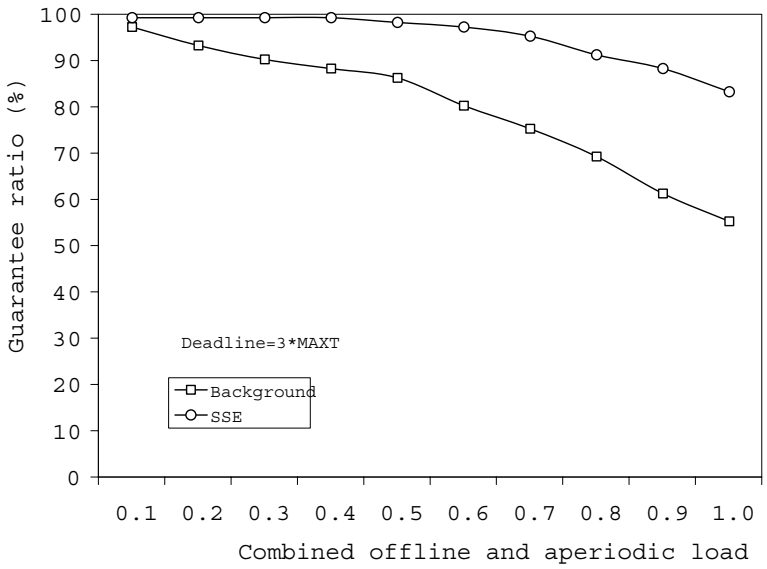


Figure 5.5: Guarantee ratio for aperiodic tasks, dl=3*MAXT – SSE vs Bgr

5.4 Experiment 2: Firm aperiodic guarantee with sporadics

In the second experiment, we have tested the acceptance ratio for firm aperiodic tasks with the methods to handle sporadic tasks described in in [2]: worst case arrivals without knowledge about invocations (refered as “no info”), and updated worst case with arrival info (“updated”).

5.4.1 Experimental setup

We studied the guarantee ratio of randomly arriving aperiodic tasks under randomly generated arrival patterns for the sporadic tasks. First we investigated the guarantee ratio for firm aperiodic tasks with combined loads 10% - 100%. The deadline for the aperiodic tasks was set to MAXT and 2*MAXT. The combined load was set to 100%.

In the second part of the experiment we varied the arrival frequencies of sporadic tasks according to a factor, f, such that the separation between in-

stances $averageMINT$ is equal to $averageMINT = f * MINT$. This means that if $f = 1$ then the instances are invoked with the maximum frequency, and if $f = 2$, the distance between two consecutive invocations is $2 * MINT$ on average.

5.4.2 Results

The results from the first part of the experimet experiment are summarized in figures 5.6 and 5.7, while the results from the second one are presented in figures 5.8 and 5.9.

We can see that our method improves the acceptance ratio of firm aperiodic tasks. This results from the fact that our methods reduce pesimism about sporadic arrivals by keeping track of them.

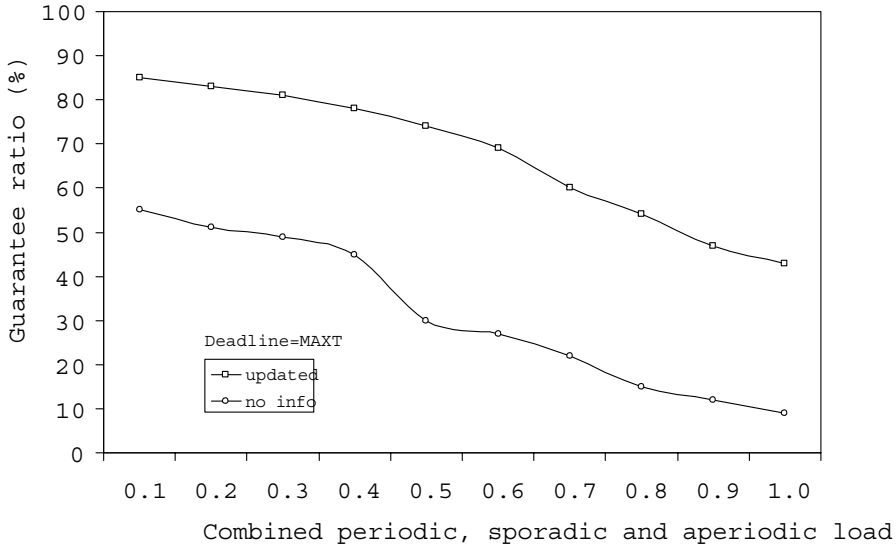


Figure 5.6: Guarantee ratio for aperiodic tasks with sporadics, dl=MAX: load variation

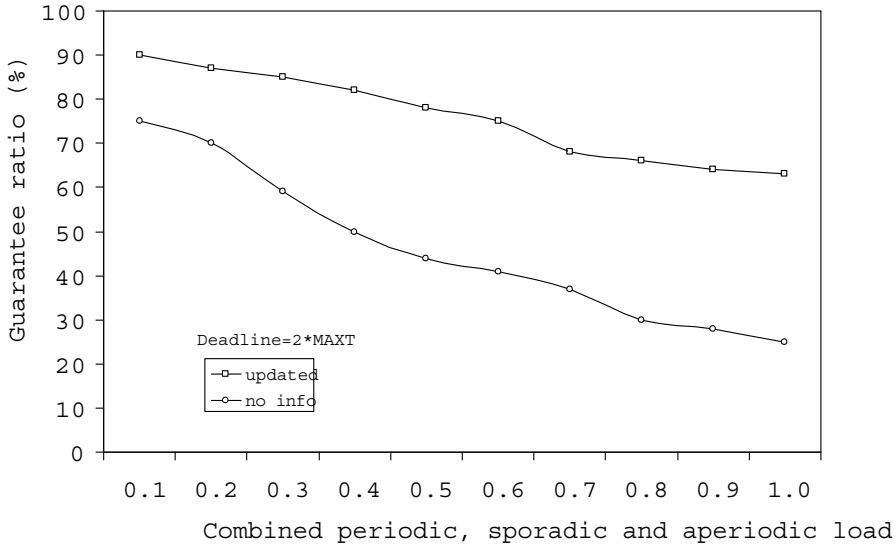


Figure 5.7: Guarantee ratio for aperiodic tasks with sporadics, dl=2*MAXT: load variation

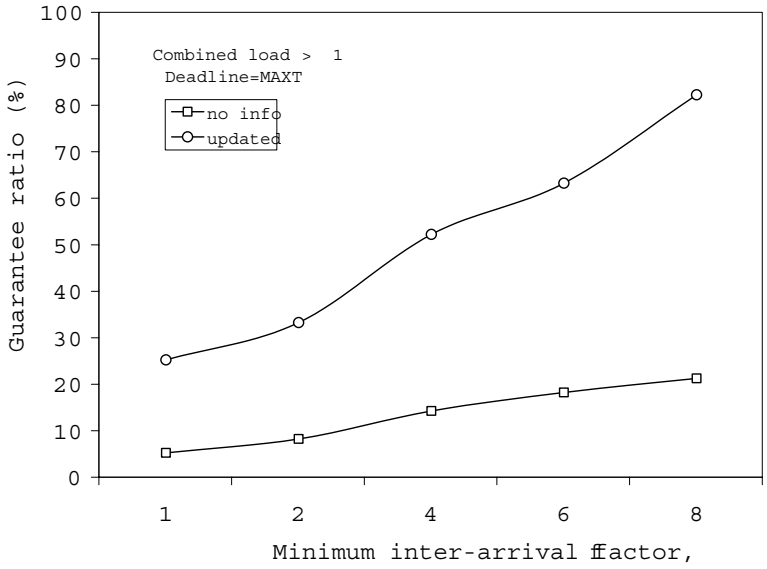


Figure 5.8: Guarantee ratio for aperiodic tasks with sporadics, dl=MAXT: variation of MINT

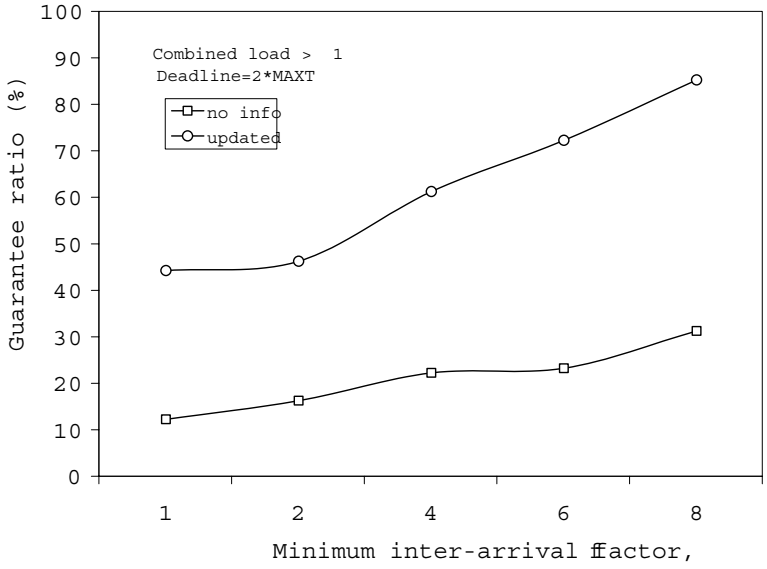


Figure 5.9: Guarantee ratio for aperiodic tasks with sporadics, dl=2*MAXT: variation of MINT

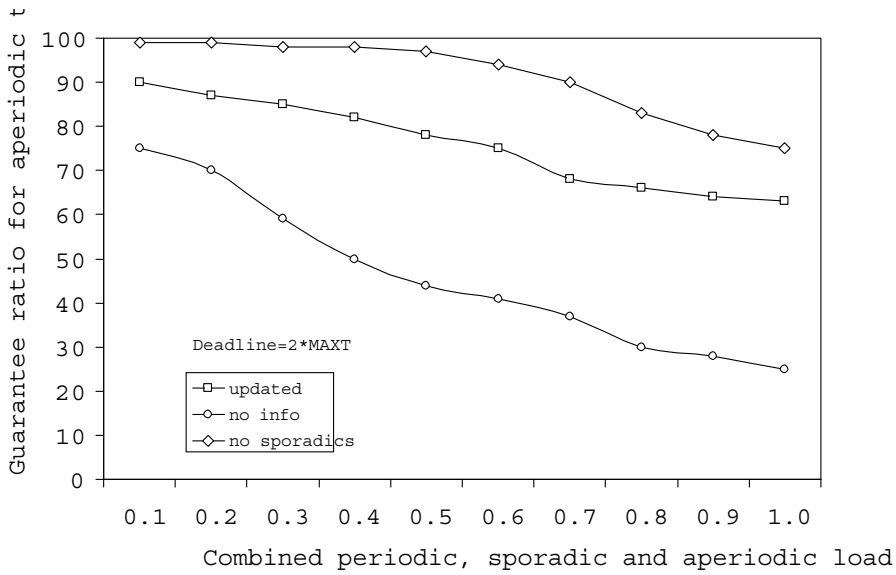


Figure 5.10: Guarantee ratio for aperiodic tasks with sporadics - Final results

5.5 Summary

In this report we have presented results of our algorithms to schedule sets of mixed types of tasks with complex constraits, by using earliest deadline first scheduling and offline complexity reduction. In particular, we presented an algorithms to efficently handle sporadic tasks in order to increase acceptance ratio for online arriving firm aperiodic tasks. We have simulated the proposed guarantee algorithms and the results underlines the effectiveness of the proposed approach. Figure 5.10 summarizes the simulation. We can see that guarantee ratio for firm aperiodic tasks is very high, even when we have sporadic tasks in the system. By keeping track off sporadic arrivals, we can accept firm tasks that otherwise would be rejected.

Future research will deal with the algorithm to include interrupts, overload handling, and complex constraints for both firm aperiodic and sporadic tasks.

Bibliography

[1] D. Isovic and G. Fohler. Online handling of hard aperiodic tasks in time triggered systems. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, June 1999.

[2] D. Isovic and Gerhard Fohler. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. *In Proceedings of the 21st IEEE Real-Time Systems Symposium, Walt Disney World, Orlando, Florida, USA*, November 2000.

[3] G. Buttazzo, A. Casile, G. Lamastra, and G. Lipari. A scheduling simulator for real-time distributed systems. In *Proceedings of the IFAC Workshop on Distributed Computer Control Systems (DCCS '98)*, 1999.

Index