Mälardalen University Doctoral Thesis
No.18

Combining Off-line Schedule Construction and Fixed Priority Scheduling in Real-Time Computer Systems

Radu Dobrin

September 2005

Department of Computer Science and Electronics
Mälardalen University
Västerås, Sweden

ISSN 1651-4238
ISBN 91-88834-87-5
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

Off-line scheduling and fixed priority scheduling (FPS) are often considered as complementing and incompatible paradigms. A number of industrial applications demand temporal properties (predictability, jitter constraints, end-to-end deadlines, etc.) that are typically achieved by using off-line scheduling. The rigid off-line scheduling schemes used, however, do not provide for flexibility. On the other hand, FPS has been widely studied and used in a number of industrial applications, mostly due to its simple run-time scheduling and small overhead. It provides more flexibility, but is limited with respect to predictability, as actual start and completion times of executions depend on run-time events.

In this thesis we first show how off-line scheduling and FPS can be combined to get the advantages of both – the capability to cope with complex timing constraints while providing run-time flexibility. The proposed approach assume that a schedule for a set of tasks with complex constraints has been constructed off-line. We present methods to analyze the off-line schedule and derive FPS attributes such that the run-time FPS execution matches the off-line schedule. In some cases, i.e., when the off-line schedule can not be expressed directly by FPS, we split tasks into instances (artifacts) to obtain a new task set with consistent task attributes. Our method keeps the number of newly generated artifact tasks minimal.

At the same time, we investigate the behavior of the existing FPS servers to handle non-periodic events, while the complex constraints imposed on the periodic tasks are still fulfilled. In particular, we pro-

vide a solution to server parameter assignment to provide non-periodic events a good response time, while still fulfilling the original complex constraints on the periodic tasks.

Secondly, we apply the proposed method to schedule messages with complex constraints on Controller Area Network (CAN). We analyze an off-line schedule constructed to solve complex constraints for messages, e.g., precedence, jitter or end-to-end deadlines, and we derive attributes, i.e., message identifiers, required by CAN's native protocol. At run time, the messages are transmitted and received within time intervals such that the original constraints are fulfilled.

Finally, we propose a method to reduce the number of preemptions in legacy FPS systems consisting of tasks with priorities, periods and offsets. Unlike other approaches, our algorithm does not require modification of the basic FPS mechanism. Our method analyzes off-line a set of periodic tasks scheduled by FPS, detects the maximum number of preemptions that can occur at run-time, and reassigns task attributes such that the tasks are schedulable by the same scheduling mechanism while achieving a lower number of preemptions. In some cases, there is a cost to pay for achieving a lower number of preemptions, e.g., an increased number of tasks and/or reduced task execution flexibility. Our method provides for the ability to trade-off between the number of preemptions and the cost to pay.

Dedicated to my family

Preface

This journey would not have been possible without the support from a number of people i meet during the last five years. I want to thank my supervisor Gerhard Fohler at the Department of Computer Science and Electronics for his guidance and constant constructive feedback through this long period of time.

Further more, I want to thank Ivica Crnkovic for his co-supervision, Salsart group members, Damir, Tomas, Larisa and Pau, for the time we spent together and for their help with reviewing the work presented in this thesis, as well as all my other colleagues at the Department of Computer Engineering for the time we have had during this period.

Many thanks go to Iain Bate and Guillem Bernat who have been a great help and support during my visit at University of York, as well as to Michael Gonzalez Harbour and the research group at the University of Cantabria, Mario, Julio, Patricia, Xavi, for their help and hospitality during my visit in Santader. A part of this work would have been extremely difficult to perform without the feedback provided by Peter Puschner who initiated me to the world of linear programming, Sasikumar Punnekat and Jukka Mäki-Turja for their availability whenever I had questions about FPS details, and Sanjoy Baruah for valuable discussions around optimization theory. Last, but definitely not least, I would like to thank my family for their support during this period of time.

This work has been supported by ARTES and the EU Information Society Technologies (IST) Project FIRST.

Radu

Publications

I have authored or co-authored the following publications:

Book chapters

- *A Component Based Real-time Scheduling Architecture*, Gerhard Fohler, Tomas Lennvall, Radu Dobrin, In Architectures for Dependable Systems, 2003. Springer Verlag, Editor(s):Rogerio de Lemos, Cristina Gacek, and Alexander Romanovsky

Conferences and workshops

- *Task Attribute Assignment of Fixed Priority Scheduled Tasks to Reenact Off-line Schedules*, Radu Dobrin and Gerhard Fohler, In Proceedings of Conference on Real-Time Computer Systems and Applications, Korea, 2000.
- *Implementing Off-line Message Scheduling on Controller Area Network (CAN)*, Radu Dobrin and Gerhard Fohler, In Proceedings of Emerging Technologies and Factory Automation, Antibes, France, 2001.
- *Translating Off-line Schedules into Task Attributes for Fixed Priority Scheduling*, Radu Dobrin, Gerhard Fohler and Peter Puschner, In Proceedings of Real-Time Systems Symposium, London, UK, 2001.

- *Attaining Flexible Real-Time Systems by Bringing Together Component Technologies and Real-Time Systems Theory* Johan Fredriksson, Mikael Åkerholm, Kristian Sandstrom and Radu Dobrin, In Proceedings of the 29th Euromicro Conference, Component Based Software Engineering Track Belek, Turkey , 2003.
- *Reducing the Number of Preemptions in Fixed Priority Scheduling*, Radu Dobrin and Gerhard Fohler, In Proceeding of Euromicro Conference on Real-Time Systems, Catania, Italy, 2004.

Licentiate Thesis

- Radu Dobrin: *Transformation Methods for Off-line Schedules to Attributes for Fixed Priority Scheduling*, Licentiate Thesis, Mälardalen University, May 2003.

Contents

1 Introduction **1**

1.1 Real-Time computer systems 1

1.1.1 Event- and time-triggered real-time systems . . . 2

1.2 Fundamental real-time scheduling paradigms 3

1.2.1 Off-line scheduling 3

1.2.2 On-line scheduling 5

1.3 System and task model 10

1.4 Real-time constraints - simple and complex 11

1.5 Scheduling complex constrained tasks in Fixed Priority Systems (FPS) 12

1.5.1 Related work 12

1.5.2 Motivation and application domains 13

1.5.3 Off-line vs. Fixed Priority Scheduling 14

1.6 Preemptions in FPS 17

1.6.1 Related work and motivation 17

1.7 Problem formulation 18

1.8 Results presented in the thesis 19

1.8.1 Transforming off-line schedules to FPS task attributes 19

1.8.2 Applying the results to schedule complex constrained messages on CAN 21

1.8.3 Handling non-periodic events together with complex constrained FPS tasks 22

1.8.4 Preemption reduction in FPS systems 24

1.9 Thesis outline 24

2 Transforming off-line schedules to FPS task attributes 25
2.1 Introduction 25
2.2 Off-line schedules 27
2.2.1 Target windows 28
2.2.2 Time triggered system operation and off-line schedule construction 28
2.2.3 FPS reenaction of off-line schedules 28
2.2.4 Increased runtime flexibility 29
2.2.5 Selectively reduced runtime flexibility 30
2.2.6 Transforming the off-line schedule to FPS tasks . 30
2.3 Problem description 31
2.3.1 Off-line schedule 31
2.3.2 Online scheduling 31
2.4 Attribute assignment algorithm 31
2.4.1 Problem formulation 32
2.4.2 Algorithm overview 32
2.4.3 Derivation of the inequalities 33
2.4.4 Attribute assignment - conflicts 35
2.4.5 ILP problem representation 39
2.4.6 Periods and offsets 41
2.5 Example 42
2.6 Discussion 47
2.7 Proofs 48
2.7.1 Proof 1 49
2.7.2 Proof 2 54
2.8 Chapter summary 56

3 Scheduling complex constrained messages on Controller Area Network (CAN) 59
3.1 Introduction 59
3.2 Controller Area Network (CAN) and message scheduling 61
3.3 Attribute assignment algorithm 62
3.3.1 Overview 62

3.3.2 CAN vs. processor scheduling 64
3.3.3 Priority inequalities 66
3.3.4 Attribute assignment - conflicts 67
3.3.5 Minimizing the final number of messages 69
3.4 Example 70
3.5 Chapter summary 74

4 Handling non-periodic events together with complex constrained fixed-priority tasks 77
4.1 Introduction 77
4.2 Existing FPS servers 78
4.3 Problem formulation 80
4.4 Motivating example 81
4.5 Proposed solution - overview 83
4.6 Server attribute assignment 84
4.6.1 Servers that do not preserve their capacity 85
4.6.2 Capacity preserving servers 87
4.7 Chapter summary 94

5 Controlling the number of preemptions in FPS 97
5.1 Introduction and problem description 97
5.2 Problem formulation and task model 100
5.3 Method overview 101
5.3.1 Preemption reduction cost 102
5.4 Solving a single preemption 103
5.4.1 Solving a preemption by eliminating the first condition 105
5.4.2 Solving a preemption eliminating the second condition 108
5.4.3 Solving a preemption eliminating the third condition 108
5.4.4 Artifact tasks 109
5.5 Reducing the number of preemptions - global approach . 111
5.5.1 Preemption dependency tree 111
5.6 A simple example 113

5.7 Performance evaluation 118
5.8 Chapter summary 120

6 Conclusions **123**

Bibliography **136**

Populärvetenskaplig svensk sammanfattning **137**

List of Figures

1.1 Table driven scheduling: run-time schedule 5
1.2 Run time RM schedule 6
1.3 Run-time EDF schedule 9
1.4 Preemption example 17

2.1 Sequence of tasks. 34
2.2 Example 1: Off-line Schedule and Target Windows . . . 37
2.3 Resulting FPS tasks. 42
2.4 Example 2: Off-line Tasks 43
2.5 Example 2: Off-line Schedule and Target Windows . . . 44
2.6 Example 2: FPS Tasks 47
2.7 Example 2: FP Schedule 47

3.1 Algorithm overview. 63
3.2 Off-line schedule 65
3.3 Sequence of messages. 67
3.4 Off-line Scheduled Messages and Target Windows . . . 71

4.1 Motivating example - original task set 82
4.2 Motivating example - problem 83
4.3 Server attribute assignment - **no** *constraint guarantees!* . 84
4.4 Server attribute assignment - *constraints guaranteed* . . 85
4.5 Non-capacity preserving server - off-line schedule 86
4.6 Polling server - FPS schedule 87

4.7 Old sequence of tasks. 89
4.8 New sequence of tasks to deal with possible server execution. 90
4.9 Deferrable serve - FPS schedule 94

5.1 A simple example 100
5.2 An off-line detected initial preemption 104
5.3 An off-line detected potential preemption 105
5.4 Pseudocode for the solving one preemption 110
5.5 Preemption dependency tree 113
5.6 Original FPS schedule: task C is preempted by A and B . 115
5.7 Example: preemption dependency tree 117
5.8 New FPS schedule, zero preemptions 118
5.9 Average preemption reduction 119
5.10 Preemption reduction cost 120
5.11 Number of FPS tasks 121

List of Tables

1.1 Real-time periodic tasks 4
1.2 Off-line scheduling table 4

2.1 Original tasks 36
2.2 Target windows derived from the off-line schedule 37
2.3 Example 1: Sequences of tasks 38
2.4 Example 2: Inequalities 45

3.1 Original set of messages 71
3.2 Target windows for message invocations 72
3.3 Inequalities 72
3.4 FP messages 75

4.1 Original tasks 81
4.2 FPS attributes 82
4.3 FPS attributes for constrained periodic tasks and a non-capacity preserving server 87
4.4 FPS attributes prepared for capacity preserving servers . 88
4.5 FPS attributes for constrained periodic tasks and deferrable server 93

5.1 Original FPS tasks 114
5.2 Target windows for the original task instances 115
5.3 The new FPS attributes that yield no preemptions 118

Chapter 1

Introduction

1.1 Real-Time computer systems

Real-time systems are computer systems in which the correctness of the system depends not only on the logical correctness of the computations performed, but also on which point in time the results are provided [52]. Delivering a result at a point in time beyond the latest possible, i.e., after its deadline, may result to catastrophic consequences in *hard real-time systems*. Example of such systems are medical control equipment or vehicle control systems. On the other hand, in *soft real-time systems*, e.g., multimedia applications, a number of deadlines can be missed without serious consequences. In this thesis we will primarily focus on hard real-time systems.

A real-time system typically consist of a number of *resources* (e.g., one or several processors), a number of *tasks*, designed to fulfill a number of *timing constraints*, and a *scheduler* that assigns each task a fraction of the processor(s) time, according to a *scheduling policy*. Tasks are usually *periodic* or *non-periodic*. Periodic tasks consist of an infinite sequence of invocations, called *instances* or *jobs*. Non-periodic tasks are invoked by the occurrence of an event. The choice of tasks and scheduling policy is made to satisfy some original constraints imposed on the system. Tasks can have various parameters, such as period, deadline,

priority, depending on the scheduling policy chosen to be used.

The scheduling policies are divided in *off-line*, [30, 28], and *on-line scheduling*, [38]. The main difference between the two is that, in off-line scheduling, the decision of which task to execute at which time point and on which processor is made at the design stage, and, at run-time, the dispatcher selects which task to execute from scheduling tables. On the other hand, in on-line scheduling, all decisions are made at run-time depending on the task priorities and their arrival times. At each point in time, the task which is ready to execute and has the highest priority, is dispatched to execute. On-line scheduling is furthermore divided in *fixed priority scheduling* (FPS), e.g., *rate monotonic* (RM), and *dynamic-priority scheduling*, e.g., *earliest deadline first* (EDF) [38].

A key issue in real-time systems is *predictability*, i.e., to be able to anticipate the behavior of the system *before* run-time and the guarantee that the system will behave as anticipated *at* run-time. At the same time, *run-time flexibility* is a desired feature, as not all run-time events can be completely accounted for in advance. Additionally, the choice of scheduling strategy in real-time systems is strongly related to the nature of the timing constraints which are to be fulfilled. As different scheduling schemes provide different levels of, e.g., predictability or flexibility, for the cost of a number of limitations, there is usually a trade-off between the ability to handle complex constraints and the level of flexibility provided by the selected scheduling strategy.

1.1.1 Event- and time-triggered real-time systems

In *event-triggered* systems, the activities happen in response to external events. The typical example of this is the sensor-actuator example: a sensor detects an external event and activates a task that reacts to this event (performs a computation), after which the task sends it's output to an actuator. This is an example of a system reacting and adjusting to an external event. One of the main issues about event-triggered systems is that external events can cause many tasks to be activated, thus, causing overload in the system, potentially leading to system failure.

Time-triggered systems, on the other hand, require a priori knowledge about all activities. In distributed time-triggered systems, each node must have the same notion of time, implying that clock synchronization is needed. The main advantage of time-triggered systems is the predictable behavior they provide at the cost of low run-time flexibility. An example of a time-triggered real-time operating system is MARS [29], and another example is TTP-OS an time triggered OS developed by TTTech [60].

In this thesis, we present mechanisms to handle real-time, complex constraints, while providing predictability and run-time flexibility for the task executions. In particular, we want to handle complex constraints while exploiting the run-time advantages provided by FPS.

1.2 Fundamental real-time scheduling paradigms

In real-time systems there are two major scheduling strategies: off-line and on-line scheduling. On-line scheduling is typically priority based and is additionally divided in fixed priority scheduling (FPS) and dynamic priority scheduling, e.g., earliest deadline first (EDF). In this section we present a brief introduction to each of them.

1.2.1 Off-line scheduling

This scheduling policy is also called *table-driven scheduling* or *cyclic scheduling*, [39], and it is a technique to allocate tasks to the processors and to resolve complex constraints by determining windows for tasks to execute in, and sequences, usually stored in scheduling tables. At run-time, a simple dispatcher selects which task to execute from the scheduling tables, ensuring tasks execute within the windows and thus meet their constraints. For off-line scheduled tasks, typically, the run-time dispatcher is invoked at regular time intervals, i.e., *slots*, and performs table lookup for task selection. The off-line scheduler assigns absolute points in time for the execution of all tasks.

Off-line scheduling for time-triggered systems provides determinism, as all times for task executions are determined and known in advance. In addition, complex constraints can be solved off-line, such as distribution, end-to-end deadlines, precedence, jitter, or instance separation. However, as all actions have to be planned before startup, run-time flexibility is lacking. While the times for temporal constraints have to be kept, e.g., a task cannot execute after its deadline, order constraints are relative, i.e., tasks can execute earlier provided the execution order of the schedule is maintained. The off-line schedule, however, prevents tasks from executing at a different time, even if resources become available earlier, e.g., by early termination of a task, i.e., the schedule is over constrained.

Example: let us assume we have 2 periodic tasks as illustrated in table 1.1, where *c* represents the execution requirement and *p* the period of the task.

$Task$	p	c
A	3	1
B	5	3

Table 1.1: Real-time periodic tasks

The processor utilization in this case is $\frac{1}{3} + \frac{3}{5} = 0.93$ and an off-line scheduler may came up with the a scheduling table schedule illustrated in table 1.2

time	2-3	3-6	6-7	7-10	10-11	11-12	12-13	13-14	14-15
task	A	B	A	B	B	A	B	A	B

Table 1.2: Off-line scheduling table

Consequently, the run-time dispatcher will perform table lookup and make sure that the run-time executions of the tasks matches the off-line scheduling table (figure 1.1)

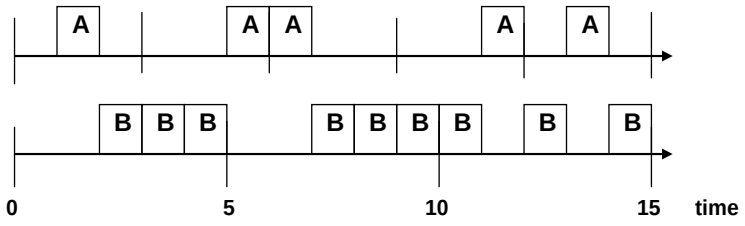


Figure 1.1: Table driven scheduling: run-time schedule

1.2.2 On-line scheduling

On-line scheduling is suitable for event triggered systems as it provides the ability to handle dynamic on-line events. SPRING [53] is an example of an event-triggered real-time operating system. This scheduling paradigm is usually priority driven, e.g., fixed or dynamic priority based.

Fixed Priority Scheduling (FPS)

FPS is one of the online scheduling strategies and it has been widely studied and used in a number of applications, mostly due its simple run-time scheduling, small overhead, and good flexibility for tasks with incompletely known attributes.

Temporal analysis of FPS algorithms focuses on providing guarantees that all instances of tasks will finish before their deadlines. If the priorities are determined by the size of the periods, i.e., the shorter the period, the higher the priority, the scheduling strategy is commonly referred to as *Rate Monotonic* (RM) scheduling. Liu and Layland [38] provided a minimum processor utilization bound (least upper bound) under which all task are guaranteed completion before their deadlines. The utilization bound is given by formula 1.1:

$$U_{lub} \leq n(2^{\frac{1}{n}} - 1) \tag{1.1}$$

When the number of tasks increase, the upper bound converges to

$$U_{lub} = ln2 \simeq 0.69$$

Some of the assumptions in the above analysis are that the tasks are independent, deadlines are equal to their periods and tasks are released at the beginning of their periods.

In our previous example where the tasks described in table 1.1 have an utilization of 0.93, the least upper bound would be

$$U_{lub} = 2(2^{\frac{1}{2}} - 1) \simeq 0.83$$

Note that, in this case, even if the utilization bound test cannot guarantee that the tasks can be scheduled by RM, the tasks still can be successfully scheduled when RM algorithm is used (figure 1.2).

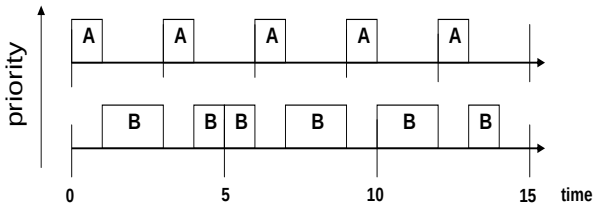


Figure 1.2: Run time RM schedule

In the case the task priorities are not scheduled by RM, i.e., priorities are not ordered according to the size of the periods, the designer has to perform a *response time analysis* (RTA) to verify that the tasks will complete before their deadlines under the worse case execution scenario.

An early RTA for the task model described by Liu and Layland was introduced by Joseph and Pandya [23], but with arbitrary assigned priorities. The formula used for deriving the worse case response time of a task T_i is presented in formula 1.2

$$R_i = C_i + \sum_{\forall j \in hp(i)} \lceil \frac{R_i}{P_j} \rceil C_j \tag{1.2}$$

where R_i is the response time of task T_i, C_i is the worse case execution time (WCET) of task T_i, $hp(i)$ denotes the set of tasks with priorities higher than T_i and P_j is the period of task T_j.

What we can see in the formula above is that R_i occurs in both sizes of the equation. The solution to calculate R_i is to solve the equation by using fix iterations on the following relation:

$$R_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \lceil \frac{R_i^n}{P_j} \rceil C_j$$

The RTA has been further extended by a number of approaches during the following years, to lift the original assumptions, such as task independency or deadlines less than or equal to periods, or to add new scheduling demands, e.g., offsets, distribution, or jitter.

The priority assignment in FPS systems, if RM is not suitable due to the application specific requirements, is, however, difficult, i.e., to find a feasible priority assignment for given set of tasks with arbitrary constraints.

The actual start and completion times of execution of tasks, however, are generally not known and depend largely on run-time events, compromising predictability.

The tasks to be scheduled by FPS policy can be either periodic or non-periodic. The periodic tasks are characterized by periods, offsets, deadlines, and same priorities for all invocations of the same task. These tasks became ready to execute at their release times, e.g., start of the

periods or offsets, and will be scheduled for execution once they have the highest priority among the tasks in the ready queue.

Non-periodic tasks can be either aperiodic or sporadic. While the aperiodic tasks attributes are unknown, i.e., arrival time, deadline or execution requirement, sporadic tasks have minimum inter-arrival times, i.e., the time interval in which at most one task instance can be released.

Furthermore, FPS is divided in *preemptive* and *non preemptive*. In preemptive FPS, tasks that are currently executing can be interrupted by any higher priority tasks at any time. On the contrary, in non preemptive FPS, tasks, once scheduled to execute, will do so until completion.

A main advantage of FPS is the flexibility it provides in terms of the ability to handle tasks with incompletely known attributes, e.g., aperiodic tasks. This is usually done by using servers that identify and use the slack in the system for non-periodic tasks.

A real-time system running according to the fixed priority scheduling strategy provides for flexible task executions since the scheduling decision, i.e., the choice of which task to execute at each time point is made at run-time. Additionally, FPS has the ability to handle on-line events with incompletely known attributes, such as aperiodic and sporadic events. However, FPS is limited with respect to the ability to handle multiple complex constraints, such as jitter, end-to-end deadlines or instance separation. For example, additional tasks or constraints added to an existing fixed priority based system require new schedulability tests which may not have been developed yet, or may find the system unschedulable in the new configuration.

Earliest Deadline First (EDF) scheduling

The main difference between FPS and EDF is that in dynamic priority scheduling, e,g., EDF, the priorities of tasks change dynamically during runtime depending on the task deadlines. As for FPS, EDF has a fairly simple run-time mechanism, the tasks must be executed in earliest absolute deadline order, and EDF also provides good flexibility to tasks, whose attributes are not completely known.

Under the assumptions used by Liu and Layland in calculation of the RM lowest upper bound, EDF performs better in the sense that a set of periodic tasks is schedulable by EDF ([38]) if and only if:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

The schedulability analysis for the EDF algorithm is very simple compared to the RM schedulability test. If the total task utilization is kept under 100% the task set is guaranteed to meet its deadlines under EDF scheduling, compared to RM where the situation becomes more complex if the task set has an utilization higher than 69%. In that case, an exact analysis is required, which is often to complex to perform on-line.

Tasks that are scheduled by EDF are similar to the FPS tasks, as they can be periodic or non-periodic (aperiodic or sporadic). The difference is that EDF tasks do not have fixed priority attributes, as the priorities are dynamically derived from the deadlines.

The tasks in our example (1.1) have an utilization of 0.93, so they are guaranteed to complete before their deadlines if scheduled by EDF. The run-time schedule under EDF scheduling is illustrated in figure 1.3.

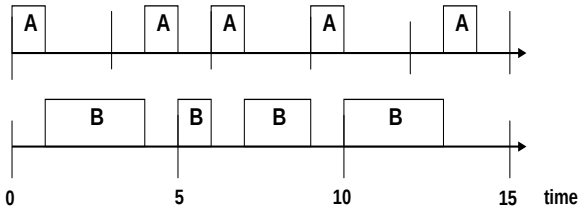


Figure 1.3: Run-time EDF schedule

Additions to the basic EDF algorithm to efficiently handle aperiodic tasks by using server mechanisms, e.g., total- or constant-bandwith

server, have been presented in [51], and in [1]. Baruah et. al., further extended constant bandwith server to multiprocessor systems [6, 7]. Normally, EDF does not behave well in the presence of overload in the systems, but Buttazzo et. al. [11] extended EDF to Robust EDF (RED), to handle overload in a predictable way.

1.3 System and task model

We consider a clock synchronized distributed system consisting of a number of processing and communicating nodes, assuming a discrete time model [27].

In general, real-time systems have to cope with a variety of events ranging from a periodic nature, e.g., sampling values provided by a temperature or pressure sensor, or non-periodic, e.g., interrupts or reacting to a button pressed by the user. In this work, we consider both types of events.

Periodic tasks They usually consist of an infinite sequence of invocations, called *task instances* or *jobs*, activated periodically. Hence, a periodic task is characterized by a period and an execution requirement, usually called *computation time*. Once a periodic task instance is scheduled to execute, and completes its execution, it will not be activated again until the beginning of its next period. An example is the electronic speed indicator in modern cars, which displays the instantaneous speed often enough, e.g., with a short activation period, such that the human eye percepts it as continuos with respect to the speed variations.

Non-periodic tasks Depending on the activation pattern, we identify two types of non-periodic tasks: *aperiodic* and *sporadic* tasks.

Aperiodic tasks are be activated by events, such as interrupts, a sensor providing a value above a predefined threshold, etc, at totally irregular time intervals. An example could be the cooling system in a vehicle, where an event handler turns on the cooling fans once the temperature has reached a certain temperature.

Sporadic tasks, on the other hand, provide the system designer slightly more information compared to aperiodics, as they have an *minimum inter-arrival time* between two consecutive invocations. Once a sporadic task has been activated, the designer can exclude the possibility of another activation within the time interval specified by the tasks minimum inter-arrival time. Hence, the chances to make accurate scheduling predictions and guarantees in the system are increased.

In this work, we primarily focus on periodic and, later on, aperiodic tasks, as this scenario implies the worse case assumptions on task activation patterns. However, the more information the system designer has, e.g., to deal with sporadic events in stead of aperiodics, the higher the chances to provide a better solution.

1.4 Real-time constraints - simple and complex

In this section we describe a number of constraints that are challenging to deal with in priority driven scheduling, e.g., FPS, while fairly easy to solve in off-line scheduling.

The most simple constraints in real-time systems are deadlines, i.e., a task, regardless from the scheduling strategy it is executed by, should execute and complete before a fixed point in time relative to it's activation. Often, however, the system designer has to map various performance requirements, e.g., such as quality of control in control systems, to specific demands on individual task or even task instances. This leads to a various number of complex constraints, the scheduler, at the end, must guarantee.

In the following, we briefly describe a number of such constraints:

- **Jitter –** The time interval between consecutive task executions is bounded by fixed values. In this case, the execution of consecutive instances of the same task has to be fixed between pre-determined points in time. Consequently, in some cases, different instances of the same tasks must have different attributes, e.g., priorities, leading to inconsistencies in FPS.

- **Precedence –** Tasks must execute in a pre-defined order, e.g., sampling and actuating tasks in real-time control systems. Work has been done to deal with precedence constraints in EDF [13] as well as precedence relations have been taken into account when performing the schedulability analysis in FPS [43]. However, the issue of attribute assignment for FPS is a challenging task as the task executions depend on the unpredictable run-time events.

- **Distribution –** Tasks are allocated to different nodes, e.g., to achieve inter-node communication. Well used in, e.g., automotive and avionics industry (e.g., [12]), together with FPS. Additionally, in some cases, tasks with precedence constraints have to be allocated to different nodes. Mapping of all these constraints to FPS attributes directly, may be a challenging task.

- **Instance separation –** Usually demanded in control systems [59], e.g., to achieve synchronized sampling, control computations and actuating [40]. That may require the execution of different instances of the same task separated by different time intervals, potentially leading to FPS attribute inconsistencies.

Depending on the underlaying scheduling mechanism in the system, such constraints are more or less challenging to guarantee. Deadlines, for example, can be guaranteed in all systems, i.e., table driven, FPS or EDF. However, instance separation, for example, may be a challenging issue to deal with in priority based systems, while easy to achieve in table driven ones.

1.5 Scheduling complex constrained tasks in Fixed Priority Systems (FPS)

1.5.1 Related work

Fixed priority scheduling (FPS) has been widely studied and used in a number of applications, mostly due by its simple run-time scheduling

and resulting small overhead. Modifications to the basic scheme to handle semaphores [48], aperiodic tasks [50], static [55] and dynamic [42] offsets, and precedence constraints [21], have been presented.

Priority assignment for FPS tasks has, for example, been studied in [5] and [20]. [47] studies the derivation of task attributes to meet overall constraints, e.g., demanded by control performance. Bate [8] provided solutions to priority assignment for deadline monotonic (DMPO). Modifications to the basic scheme to handle semaphores [48], aperiodic tasks [50], static [55] and dynamic [42] offsets, and precedence constraints [21], have been presented. Sandsröm [45] presented a method for pre-run-time scheduling of periodic control activities under the interference of sporadic interrupts.

Controller Area Network (CAN), has gained wider acceptance as a standard in a large number of industrial and automotive applications. The priority based message scheduling used in CAN has a number of advantages, some of the most important being the efficient bandwidth utilization, flexibility, simple implementation and small overhead. Early results on message scheduling on CAN have been presented in [57] and [56], in which the authors focused on fixed priority scheduling based on work presented in [38] and [37]. Later on, Zuberi [63] showed that static priority scheduling is not always the most suitable strategy. Earliest Deadline (EDF) can prove significantly better than fixed priority scheduling [41] with respect to schedulability.

1.5.2 Motivation and application domains

One of the purposes of the work presented in this thesis is to provide methods to combine the advantages of both scheduling strategies, off-line scheduling and FPS, i.e., predictability and ability to solve complex constraints while flexibility at run-time. Instead of direct mapping specific requirements into FPS attributes, our methods use the ability of off-line scheduling to solve complex constraints and, then, take advantage of the information provided in the off-line schedule to determine task attributes suitable for FPS.

1.5.3 Off-line vs. Fixed Priority Scheduling

Off-line scheduling, [28, 30], and fixed priority scheduling (FPS), [5, 20], are often considered as having incompatible paradigms, but complementing properties. FPS has been widely studied and used in a number of applications, mostly due to its simple run-time scheduling, small overhead, and good flexibility for tasks with incompletely known attributes. Temporal analysis of FPS algorithms focuses on providing guarantees that all task instances will finish before their deadlines. However, additional constraints to FPS schemes require new schedulability tests, which may not have been developed yet, or may find the system unschedulable in the new configuration. Hence, the run-time flexibility provided by FPS comes at the expense of ability to handle multiple constraints, such as, jitter, instance separation or end-to-end deadlines. The actual start and completion times of execution of tasks, however, are generally not known and depend largely on run-time events, compromising predictability.

Furthermore, FPS is widely used in a number of industrial applications involving network scheduling using Controller Area Network (CAN). An approach to time–triggered communication on controller area network has been presented in [33], while in [2], the authors presented an approach to enhance both event– and time–triggered communication in CAN. However, both approaches imply modifications to the native CAN protocol.

Off-line, table driven, scheduling for time-triggered systems, [28], on the other hand, provides predictability, as all times for task executions are determined and known in advance. The off-line scheduler allocates tasks to the processors and resolves complex constraints by determining windows for tasks to execute in, and sequences, usually stored in scheduling tables. At run-time, the dispatcher, invoked at regular time intervals, selects which task to execute from the scheduling tables, ensuring tasks execution within the off-line determined windows and, thus, meet their constraints.

However, as all actions have to be planned before startup, run-time flexibility is lacking. The advantage of solving complex constraints

comes at a price of limited run-time flexibility in terms of ability to handle tasks with incompletely known attributes, e.g., aperiodic or sporadic tasks.

Off-line scheduling for time triggered systems has, for example, been studied in [28, 30]. The choice of scheduling technique used in order to achieve different requirements has been analyzed and discussed [62], [39].

Instead of enhancing only either FPS or off-line scheduling alone, the solutions presented in this thesis provide for a combination of both, such that benefits of either scheme are accessible to the other. In particular, the presented methods provide solutions for the following scenarios:

Legacy Systems: Some safety critical applications, e.g., in the avionics domain [12], demand temporal partitioning of task executions or assertions not only about deadline being met, but restrictions on the actual times when task executions are performed. Typically, such applications are executed in time triggered architectures with off-line schedule construction. A move to fixed priority based systems has to ensure the specific demands will be met, which may be cumbersome applying standard FPS methods, as these concentrate on deadlines primarily.

The proposed methods transform these demands directly into attributes for tasks to be feasibly scheduled by FPS, pertaining the predictability provided by off-line schemes.

FPS systems with unresolved constraints: The feasibility test provided for FPS tasks defines the types of constraints which can be met. Additional constraints or combinations require modifications to existing tests or the development of new ones, which may not be available in limited time.

In addition, constraints demanded by complex applications, cannot be expressed generally. Control applications may require constraints on individual instances rather than fixed periods, reliability demands can enforce allocation and separation patterns, or engineering practice may require relations between system activities.

The proposed method resolves the need for developing of new specific tests for unusual constraints: first, a designer will apply known tests on the application under consideration. Should standard schemes prove to be incapable, the designer submits these tasks to an off-line scheduling scheme, which can use elaborate and general methods, such as search or constraint satisfaction, to provide a feasible off-line schedule. Then, by using the proposed methods, we derive attributes, such as period, priority, and offset for these tasks, such that they can be scheduled with FPS, while meeting the specific constraints.

Predictable flexibility: Off-line scheduling provides deterministic execution patterns for all tasks in the system, while FPS schemes provide flexibility for all tasks. Only few applications, however, will demand either determinism or flexibility uniformly for all activities in the system. Rather, only few selected tasks have tight restrictions on their executions, e.g., those sampling or actuating in a control system, with strict demands on jitter and variability, while a majority can execute flexibly.

Off-line scehduling in Controller Area Network (CAN): Increasing demands from industrial applications using CAN for communication, leads to increased complexity imposed on system. Consequently, messages transmitted by nodes have to fulfill a number of requirements in form of complex constraints.

On the other hand, off-line scheduling for time triggered systems provides determinism [28, 30], and, additionally, complex constraints can be solved off-line, but this scheduling strategy is not directly suitable for the native CAN protocol.

By using the proposed methods, we transform off-line scheduled transmission schemes into sets of messages that can be feasibly scheduled on CAN without modifying the basic CAN mechanism.

The methods allows the amount of flexibility at runtime to be set off-line in a predictable way by including restrictions on task execution as input to the transformation algorithm.

1.6 Preemptions in FPS

1.6.1 Related work and motivation

The impact of preemption related overhead in FPS in the context of real-time systems, is well recognized [9, 44]. In multimedia applications, for example, tasks may introduce a high context switch cost [17]. In fact, preemption related overhead in FPS may cause undesired high processor utilization, high energy consumption, or, in some cases, even infeasibility.

Example 1.4 illustrates a situation where task B, while preempted by A, misses its deadline due to the introduced context switch cost (a). However, B would meet its deadline if A would not preempt it (b).

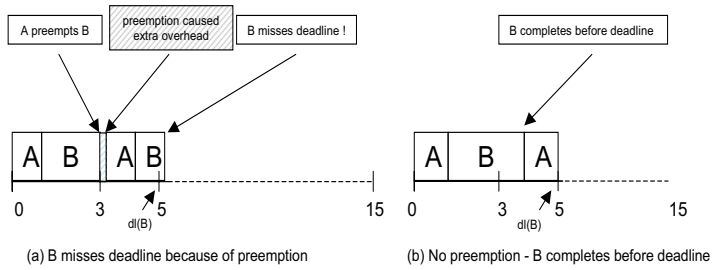


Figure 1.4: Preemption example

The direct preemption cost, i.e., costs to perform context switches [24], to handle interrupts [24, 22, 9], or to manipulate task queues [9, 24], has been analyzed. Cache-related preemption, i.e., indirect cost, [31, 46], has been analyzed to incorporate it into schedulability analysis, as well as approaches to bound the cache-related preemption delay have been presented [32]. Approaches to reduce the number of preemptions in FPS have been presented [61, 25, 26], where tasks, besides their priorities, are assigned a threshold value such that they can be preempted only by other tasks with priorities higher than the threshold. This approach results, in essence, in a dual priority system which is not directly

suitable for legacy systems, where exchanging the scheduler or modifying it by, e.g., adding mutexes to simulate preemption threshold, is not desirable, or not possible.

In [10], the author showed that RM introduces a higher number of preemptions than EDF. At the same time, reducing the number of preemptions can also be beneficial from an energy point of view in systems with demands on low power consumption. When a task is preempted there is a great probability that its contents in the cache will be lost. When the execution of the task is again resumed it will cause a lot of energy consuming accesses to off-chip memory. An access to off-chip memory is typically 10-100 times more expensive than an on-chip cache access in terms of energy consumption. Reducing the number of preemptions will reduce these additional expensive memory accesses due to reduced cache pollution.

Moreover, schedulability of a task set in preemptive FPS does not imply the schedulability if the same task set in non-preemptive FPS.

1.7 Problem formulation

In this work we aim to combine the advantages provided by two fundamental scheduling paradigms: off-line and fixed priority scheduling.

First, we introduce a method to transform off-line scheduled tasks with complex constraints to attributes for fixed priority scheduling. First, an off-line schedule is constructed for a set of tasks to fulfill their complex constraints. Then, by analyzing the off-line schedule together with the original constraints, we derive FPS attributes, i.e., priorities, offsets, deadlines, such that the tasks, when scheduled by FPS, will execute flexibly, while fulfilling the same complex constraints of the original off-line scheduled tasks.

Instead of direct mapping specific requirements into FPS attributes, our methods use the ability of off-line scheduling to solve complex constraints and, then, take advantage of the information provided in the off-line schedule to determine task attributes suitable for FPS.

Thus, we combine FPS flexibility at runtime with the capability of off-line scheduling to resolve complex constraints.

Secondly, we apply the proposed approach to implement off-line message scheduling in CAN. We use off-line scheduling to reduce the complexity of mapping complex constraints to unique identifiers required for message scheduling on CAN. We take advantage of the CAN particularities to optimize our solution, as well as we modify our previous approach to meet the scheduling mechanism requirements on CAN.

As one of the main advantages provided by FPS is flexibility in terms of the ability to handle events with incompletely known attributes, we investigate the ability of our method to handle non-periodic events by using existing fixed priority servers. In particular, we provide mechanisms to allow for the inclusion of non-periodic events via servers, as well as a method to assign attributes to servers such that the non-periodic event receive the best service while the constraints on the periodic tasks are fulfilled.

Finally, we address the issue of preemption cost in FPS systems. We introduce a method to control the number of preemptions in existing FPS systems, with no modifications to the underlaying scheduler. As preemption reduction does not come for free, we provide users of FPS systems for the ability to trade of the level of preemptions for the preemption reduction cost, with no modifications to the original scheduler.

1.8 Results presented in the thesis

1.8.1 Transforming off-line schedules to FPS task attributes

First, we present a method to combine off-line schedule construction with fixed priority scheduling by determining task attributes for the off-line scheduled tasks, such that the original schedule is reconstructed if

the tasks are scheduled by FPS at run-time. The method analyzes an off-line schedule together with original task constraints to create sequences and windows of tasks. Priorities and offsets are set to ensure task orders in sequences and relations between windows.

FPS cannot reconstruct all schedules with periodic tasks. The constraints expressed via the off-line schedule may require that instances of a given set of tasks need to be executed in different order on different occasions. Hence, there not always exist a valid FPS priority assignment that can achieve these different orders. Our methods detects such situations, and circumvents the problem by splitting a task into its instances. Then, the algorithm assigns different priorities to the newly generated "artifact" tasks, the former instances. Lower priority tasks can be added for run-time use.

Key issues in resolving the priority conflicts are the number of artifact tasks created, and the number of priority levels. Depending on how the priority conflict is resolved, the number of resulting tasks may vary, e.g., splitting a task with a large number of instances over LCM would result in a large number of artifacts. We minimize the number of artifacts by generating optimal solutions with an ILP-based algorithm.

We assume that a schedule for a set of tasks with complex constraints has been constructed off-line and we present a method to analyze the off-line schedule and derive an FPS task set with FPS attributes priority, offset, and period, such that the runtime FPS execution matches the off-line schedule. The proposed method analyzes the schedule and sets up inequality relations for the priorities of the tasks under FPS. Integer linear programming (ILP) is then used to find a FPS priority assignment that fulfills the relations. In case the priority relations for the tasks of the off-line schedule are not solvable we split tasks into the number of instances, to obtain a new task set with consistent task attributes. By using ILP, we can ensure that our schedule translation algorithm keeps the number of newly generated artifact tasks minimal, while additional demands can be added by inclusion in the goal function.

1.8.2 Applying the results to schedule complex constrained messages on CAN

Furthermore, we extended the previously introduced approaches ([16]), to implement off-line scheduling in Controller Area Network (CAN) ([15]). The ILP-formulation is modified to ensure unique priorities for the messages, as required by the CAN protocol.

In this approach, we apply the previously developed methods to take advantage of the benefits of off-line scheduling in (CAN). Assuming that a schedule, for a set of tasks transmitting messages on CAN, has been constructed off-line, we present a method that analyzes the off-line schedule and derives a set of periodic messages with fixed priorities, which can be scheduled on CAN. Based on the information provided by the off-line schedule, the method derives inequality relations between the priorities of the messages under FPS. In case the priority relations of the messages are not solvable, we split some messages into a number of artifacts, to obtain a new set of messages with consistent identifiers. We use integer linear programming to minimize the final number of messages.

The off-line analysis we perform in the proposed method is simplified when applied to CAN, due to the CAN properties which we can take advantage of:

- The message length is constant in CAN – This fact ease the off-line analysis we perform in our methods, as it avoids situations where the order of transmission may change as a consequence of variable transmission time. In task scheduling we assume task executing for worst case execution time (WCET) while the actual execution time at run-time is most likely much less.

- CAN scheduling is non-preemptive – In preemptive task scheduling, the execution order may change, due to variations in the execution times. In CAN this problem no longer exist since the message length is constant and no preemptions may occur.

An additional issue is that CAN scheduling require unique priorities.

We solve the issue by simply adding an extra constraint to the ILP.

By using our method, we solve the issue of attribute assignment for a set of messages with complex constraints and schedule them on CAN while preserving the native CAN mechanism.

1.8.3 Handling non-periodic events together with complex constrained FPS tasks

Motivation and related work

One of the main advantages provided by FPS is the capability to handle non-periodic events, e.g., aperiodic tasks. In FPS, non periodic events are commonly handled by servers, e.g., background scheduling, polling or deferrable servers, or slack stealing. FPS servers are scheduled as tasks with periods, capacities and priorities, to use the slack in the system for non-periodic tasks.

Polling, Deferrable and Sporadic Server, as well as Priority Exchange have been introduced by Lehoczky and Sprunt et. al. in '87 and '89, [36, 49], while Slack Stealing was introduced by Lehoczky et. al. in '92, [35].

All of them have advantages and disadvantages in terms on responsiveness of the non-periodic tasks vs. computational overhead and utilization bounds for the periodic tasks. In 1995, Tia et. al. [54] proved the non optimality of the existing fixed-priority servers, i.e., no existing approach can minimize the response time of non-periodic tasks while still guaranteeing the feasibility of the periodic ones. The main reason is that, in some cases, aperiodics may have to execute later than the earliest possible start of execution in order to achieve the best response time.

However, the utilization bound analysis was provided for the servers together with periodic tasks scheduled by RM. In our methods, on the other hand, the tasks are not scheduled by RM as we aim to deal with periodic tasks with complex constraints. As a consequence, our task set consists of tasks with arbitrary priorities, offsets and deadlines shorter than periods.

In this work we investigated the possibility to use the existing FPS servers to handle non-periodic events together with the tasks obtained by using the our transformation approach. We aimed to be able to handle non-periodic events while not jeopardizing the complex timing constraints of the periodic tasks.

Proposed solution

In this work we have analyzed the servers based on their run-time behavior with respect to their activation times.

For analysis purposes, we can divided the FPS servers in two major categories depending on the behavior of the server upon the presence (or absence) of a non-periodic event: servers that do not preserve their capacity during their period if no aperiodic requests are pending, and servers that do. In the first case, if no aperiodic tasks are waiting to be served at the beginning of the server period, the server capacity is waisted and replenished at the beginning of the next period.

We show that servers that do not preserve their capacity during their periods, if no aperiodic requests are pending at the beginning of the period, can be easily incorporated in our method while guaranteeing the constraints of the periodic tasks, by treating them as periodic tasks that are ready to execute at the beginning of their periods, and do not suspend themselves. In this case, the only difference between the server and the periodic tasks is that the server may not execute at all if no aperiodic requests are pending at the beginning of its period.

Furthermore, we provide a mechanism to be able to handle non-periodic events by existing server that preserve their capacity during their periods until an aperiodic requirement occurs. The difficulty of scheduling this type of servers together with periodic, complex constrained tasks, is that the server execution, i.e., at which time the server starts its execution within its period, can not be predicted such that it can be included in the off-line schedule construction.

While our method can feasibly schedule the existing servers together with the periodic FPS tasks created by the method described in chapter 2, any feasible attribute assignment can be used on the servers as long as

the designer can guarantee the completion of the periodic tasks before their deadlines. That is because, in our off-line to FPS transformation, we specified the release times and deadline of the tasks, i.e., the target windows, in which if the tasks execute and complete, the original constraints are guaranteed.

1.8.4 Preemption reduction in FPS systems

Proposed solution

Once we have migrated off-line scheduling based systems to FPS, we aimed to further improve FPS by controlling the number of preemptions. In particular, we propose a method to reduce the number of preemptions in legacy FPS systems consisting of tasks with priorities, periods and offsets. Approaches to reduce the number of preemptions exist, but they modify the basic FPS mechanism.

Our method analyzes off-line a set of periodic tasks scheduled by FPS, detects the maximum number of preemptions that can occur at runtime, and reassigns task attributes such that the tasks are schedulable by the same scheduling mechanism, while achieving a significantly lower number of preemptions.

In some cases, there is a cost to pay for a lower number of preemptions in terms of increased amount of tasks and/or reduced task execution flexibility. Our method provides for the ability to choose a user-defined number of preemptions with respect to the cost to pay.

1.9 Thesis outline

In chapter 2 we present an a method to combine off-line and fixed priority scheduling. We apply the results to CAN message scheduling in chapter 3 and we extend it to fully take advantage of FPS mechanism in chapter 4 where we handle non-periodic events. In section 5 we introduce a method to control the number of preemptions in FPS and we conclude the thesis in chapter 6.

Chapter 2

Transforming off-line schedules to FPS task attributes

2.1 Introduction

Fixed priority scheduling (FPS) has been widely studied and used in a number of applications, mostly due by its simple run-time scheduling and resulting small overhead. Modifications to the basic scheme to handle semaphores [48], aperiodic tasks [50], static [55] and dynamic [42] offsets, and precedence constraints [21], have been presented. Consequently, FPS enables good flexibility for tasks with incompletely known attributes. Temporal analysis of FPS algorithms focuses on meeting deadlines, i.e., guarantees that all instances of tasks will finish before their deadlines. The actual times of executions of tasks, however, are generally not known and depend largely on run-time events, compromising predictability.

Off-line scheduling for time-triggered systems, on the other hand, provides strong predictability, as all times for task executions are determined and known in advance. In addition, complex constraints can be solved off-line, such as distribution, end-to-end deadlines, precedence,

jitter, or instance separation. All this is enabled at the expense of loosing run-time flexibility, as all actions have to be planned before.

In this chapter, we present an algorithm to combine off-line schedule construction with fixed priority run-time scheduling. The resulting systems have a time-triggered base that is complemented with event-triggered on-line scheduling. This allows us to combine benefits of off-line scheduling, in particular a distributed system, complex, constrained tasks, and end-to-end deadlines, with online scheduling, which allows flexible task execution. A number of tasks are specified to execute predictable, while allowing flexibility for all others.

Our method works by transforming off-line scheduled tasks with their original constraints into tasks with attributes suited for fixed priority scheduling, i.e., periods, deadlines, and offsets, which will reenact the original offline schedule at runtime. It divides the off-line schedule and its tasks into windows and sequences, sets priorities to ensure execution orders within windows, and determines priorities and offsets to ensure orders and relations between windows. As FPS cannot reconstruct all schedules with periodic tasks, our algorithm can split tasks into several instances to achieve consistent task attributes. Tasks with lower priorities can be added for run-time scheduling.

Priority assignment for FPS tasks has been studied in, e.g., [5], [20], and [47] study the derivation of task attributes to meet a overall constraints, e.g., demanded by control performance. Instead of specific requirements, our algorithm takes an entire off-line schedule and all task requirements to determine task attributes. A method to transform off-line schedules into earliest deadline first tasks has been presented in [19].

The proposed method allows the amount of flexibility at runtime to be set off-line in a predictable way by including restrictions on task execution as input to the transformation algorithm.

FPS cannot reconstruct all schedules with periodic tasks with the same priorities for all instances directly. The constraints expressed via the off-line schedule may require that instances of a given set of tasks

need to be executed in different order on different occasions. Obviously, there exist no valid FPS priority assignment that can achieve these different orders. Our algorithm detects such situations, and circumvents the problem by splitting a task into its instances. Then, the algorithm assigns different priorities to the newly generated "artifact" tasks, the former instances.

Key issues in resolving the priority conflicts are the number of artifact tasks created, and the number of priority levels. Depending on how the priority conflict is resolved, the number of resulting tasks may vary, depending on the periods of the split tasks. Our algorithm minimizes the number of artifact tasks. By using an ILP solver for the derivation of priorities, additional demands such as reducing number of preemptions levels can be added by inclusion in the goal function.

The chapter is organized as follows: in section 2.2 we discuss the rationale of our method. We give a brief description of the problem in section 2.3 followed by the problem formulation and the basic idea of the proposed algorithm in section 2.4. We illustrated our method by an example in Section 2.5. In section 2.7 we give the format proofs to the proposed approach and we summarize the chapter in section 2.8.

2.2 Off-line schedules

In this section, we discuss the rationale of our method and position its functionality with respect to application timing constraints, off-line scheduler, and FPS online execution of tasks. We discuss the complexity reduction of the NP hard scheduling problem with general constraints achieved by the off-line scheduler and how the new method provides for a selective choice of degree of online flexibility of the resulting FPS tasks.

Before starting the discussion, we introduce the term *target windows*.

2.2.1 Target windows

We define the target window of a task as the interval of time in which the instance will execute and complete at run-time. For example, the target window of a task scheduled by the RM algorithm will be the period of the task. The target window of a task scheduled off-line will consist of the time slots that the off-line scheduler assigned to the task.

2.2.2 Time triggered system operation and off-line schedule construction

We assume a distributed system of stand-alone computers connected via a shared network. Tasks are allocated to these nodes, communicate across the system, and are demanded to fulfill complex constraints, such as precedence, end-to-end deadlines, and jitter. The off-line scheduler allocates tasks to nodes and resolves complex constraints by determining windows for tasks to execute in, and sequences, usually stored in scheduling tables. The resulting off-line schedule is one feasible, likely suboptimal solution. At run-time, a simple dispatcher selects which task to execute from the scheduling tables, ensuring tasks execute within the windows and thus meet their constraints.

This way, the complexity of the original scheduling problem is reduced off-line, which allows for elaborate methods, improvement of results and modifications in the failure case. The transformation method can be flexible to include new types of constraints, to accommodate application specific demands and engineering requirements. Runtime scheduling is simplified to executing taskset within the windows constructed by the off-line scheduler, called *target windows* further on; then all original constraints will be fulfilled.

2.2.3 FPS reenaction of off-line schedules

For off-line tasks, typically, the runtime dispatcher is invoked in each node at regular time intervals, *slots*, and performs table lookup for task selection. While being simple, this is not the only way to ensure tasks

execute in the windows, from now on called *target windows*, and in the order computed by the off-line scheduler.

In this work, we propose to use standard fixed priority scheduling instead: by deriving priorities and offsets for tasks in such a way that tasks execute within their target windows and fulfill the precedence requirements when scheduled by FPS, the system will reenact the off-line schedule. Thus, the advantages of deterministic off-line scheduling can be combined with FPS scheduling at runtime.

Strict reenaction on a slot basis will over constrain the schedule for the sake of determinism by eliminating all flexibility of the resulting tasks. While this is demanded in some safety critical applications which apply FPS scheduling, e.g., for legacy reasons, the potential for flexibility in FPS scheduling remains unused.

2.2.4 Increased runtime flexibility

By modifying target windows, flexibility can be increased at runtime, while keeping the original constraints. Target windows created by the off-line scheduler resolve two types of constraints: *temporal*, e.g., start of periods, end-to-end deadlines, sending or receipt of messages over the network, or jitter, instance separation, etc, and *order of task execution*, as determined by the off-line scheduler, e.g., for data flow processing, precedence, mutual exclusion. The off-line scheduler resolves both types of constraints by assigning absolute points in time for the execution of all tasks. While the times for temporal constraints have to be kept, e.g., a task cannot execute after its deadline, order constraints are relative, i.e., tasks can execute earlier provided the execution order of the schedule is maintained.

The off-line schedule, however, prevents tasks from executing at a different time, even if resources become available earlier, e.g., by early termination of a task, i.e., the schedule is over constrained.

We provide an execution pattern which is more flexible than interpreting an offline schedule but nevertheless guarantees to meet the given constraints. We propose to join target windows of order constrained tasks which have the same temporal constraints. These tasks

form chains according to their order inside these new target windows. Thus, we can exploit more flexibility while maintaining the constraints resolved by the offline schedule.

2.2.5 Selectively reduced runtime flexibility

While desirable in general, additional flexibility may be harmful for some tasks, e.g., those sampling and actuating in a control system. For such tasks, the deterministic execution provided by the offline schedule has to be pertained. Eliminating flexibility for *all* tasks in the system to preserve only few, over constrains the system.

We propose to prevent the joining of target windows or to reduce the length of some windows selectively to keep the strict execution behavior of selected tasks, while providing flexibility for the rest. Thus, our methods allows the amount of run-time flexibility of a task to be set offline in a predictable way.

2.2.6 Transforming the off-line schedule to FPS tasks

In this cahpter, we present a method which determines attributes for tasks assigned to target window and associated chains such that, if executed according fixed priority scheduling, they will execute inside their target window and according to the order inside the chains.

Target windows can be derived from off-line schedules directly, without further knowledge about the original timing constraints. In that case, the off-line schedule will be re-enacted exactly by the FPS tasks, providing the same determinism. The resulting assignment, however, will lead to inflexible schedules and inefficient attributes.

We envision our method to be used with transformations as described above, providing for predictable flexibility while deriving simple attributes.

2.3 Problem description

2.3.1 Off-line schedule

First, an off-line schedule is created for a set of tasks and constraints. While our method does not rely on a particular off-line scheduling algorithm, we have used the one described in [3] for our implementation and analysis. The schedule is usually created up to the least common multiple, LCM, of all task periods. LCM/p(T_i) instances of each task T_i with period p(T_i) will execute in the schedule.

The off-line scheduler resolves constraints such as distribution, end-to-end deadlines, precedence, etc, and creates scheduling tables for each node in the system, listing start- and finishing-times of all task executions. These scheduling tables are more fixed than required by the original constraints, so we can replace the exact start- and finishing-times of tasks with feasibility windows, taking the original constraints into account. A task receiving (sending) a message over the network, for example, has to start (finish) after (before) the scheduled transmission time, giving more leeway than the rigid scheduling table, defining release times (deadlines). Slot shifting [4] uses this method to transform off-line schedules into task for earliest deadline first scheduling.

2.3.2 Online scheduling

At run-time, we want tasks to be scheduled according to fixed priority assignment. Our method assigns priorities, offsets, deadlines, periods. We refer to an *execution window*, $Wexec(T_i^j)$, of an instance T_i^j of a task T_i, as the time interval in which T_i^j will execute and complete if scheduled by FPS together with the other instances of the other tasks.

2.4 Attribute assignment algorithm

We present a method which determines attributes for tasks assigned to target windows and associated chains such that, if executed according

to fixed priority scheduling, they will execute inside their target window and obey the order constraint of the task chains.

2.4.1 Problem formulation

Given a set of off-line scheduled tasks, $Original\_Tasks$ where

$$Original\_Tasks = \{T_1, T_2, \ldots, T_n\}$$

with constraints represented by target windows, $TW_i,\ i = 1, 2, \ldots$, we want to transform them into a set of tasks, *FPS_Tasks* where

$$FPS\_Tasks = \{\overline{T_1}, \overline{T_2}, \ldots, \overline{T_m}\},$$

with attributes suitable for FPS, i.e., $prio(\overline{T_i})$, $o(\overline{T_i})$, $dl(\overline{T_i})$, $p(\overline{T_i})$, such that:

1. Each instance of each task $\overline{T_i}$ will execute at run time inside its target window
2. The order of execution enforced by the original task constraints is preserved

if the tasks $\overline{T_i} \in \{FPS\_Tasks\}$ are scheduled by FPS.

2.4.2 Algorithm overview

As input to our method we have:

Taskset $T_i \in \{Original\_Tasks\},\ i = 1, 2, \ldots$, with constraints expressed in:

- Off-line schedule, up to LCM, expressing the original task constraints, that gives off-line scheduled start- and finishing times for each instance j of each task T_i

$$\begin{aligned} st(T_i^j) &= off-line\ scheduled\ start\ time\ of\ T_i^j \\ ft(T_i^j) &= off-line\ scheduled\ finishing\ time\ of\ T_i^j \end{aligned}$$

- Target windows, $TW_n,\ n = 1, 2, \ldots$, that gives earliest start times and deadlines for each instance T_i^j of each task T_i:

$$est(T_i^j) = begin(TW_n) = begin(TW(T_i^j))$$

and

$$dl(T_i^j) = end(TW_n) = end(TW(T_i^j))$$

We start with target windows and sequences of instances. We translate order constraints into priority constraints between the new FP tasks.

We may not be able to find a FPS schedule with the same number of tasks as the original one, but we may have to create new tasks by splitting some of the original off-line tasks. The resulting number of FPS tasks is to be minimized.

Output: we are looking for a set of tasks, $\overline{T_i} \in \{FPS\ Tasks\}$, with:

- Priorities, $prio(\overline{T_i})$
- Offsets, $o(\overline{T_i})$
- Periods, $p(\overline{T_i}),\ where\ \overline{T_i} \in \{FPS\_Tasks\}$

2.4.3 Derivation of the inequalities

Given the target windows derived from the original constraints and the off-line schedule and defined as $TW_n = \{T_i^j \mid est(T_i^j) = t_k = begin(TW_n)$, and $dl(T_i^j) = end(TW_n)\}$, where $begin(TW_n)$ is the starting time and $end(TW_n)$ the end time of the n^{th} target window, we derive the *sequences of tasks* corresponding to the start of each target window. The derivation of the sequences is illustrated in figure 2.1.

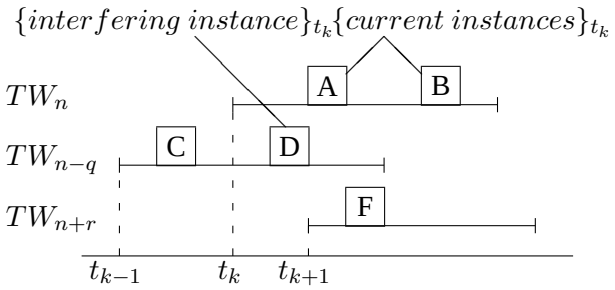


Figure 2.1: Sequence of tasks.

A **sequence of tasks** SEQ_k consists of task instances, ordered by increasing scheduled start times according to the off-line schedule. A sequence may contain instances T_i^j of tasks T_i such that $est(T_i^j) = t_k$, referred to as $\{current\ instances\}_{t_k}$, or instances T_s^q of tasks T_s from overlapping target windows such that $est(T_s^q) < t_k$ and $ft(T_s^q) > t_k$, which we refer to as $\{interfering\ instances\}_{t_k}$, where t_k is the starting time of TW_n, i.e., $begin(TW_n)$.

$$\begin{aligned} SEQ_k &= \{\{current\ instances\}_{t_k} \cup \\ &\quad \cup\{interfering\ instances\}_{t_k}\}_{ordered} = \\ &= < \overset{k}{S}_1, \overset{k}{S}_2, \ldots, \overset{k}{S}_N > \end{aligned}$$

Where:

- $\{current\ instances\}_{t_k} = \{T_i^j \mid est(T_i^j) = t_k\}$
- $\{interfering\ instances\}_{t_k} = \{T_s^q \mid est(T_s^q) < t_k \wedge ft(T_s^q) > t_k\}$

Additionally, we define:

- $first(SEQ_k) = \overset{k}{S}_1 = first\ task\ instance\ in\ SEQ_k$
- $last(SEQ_k) = \overset{k}{S}_N = last\ task\ instance\ in\ SEQ_k$

The priority assignment has to preserve the execution order expressed in the off-line schedule. Therefore, from each sequence of tasks SEQ_k, $k = 1, 2, \ldots$, we derive priority relations between the task instances within SEQ_k.

$$prio(\overset{k}{S}_1) > prio(\overset{k}{S}_2) > \ldots > prio(\overset{k}{S}_N)$$

The priority inequality system derived from the sequences of tasks, includes all task instances in the off-line schedule.

2.4.4 Attribute assignment - conflicts

Based on the order of execution expressed by the inequalities derived in Section 2.4.3, we derive attributes - priorities and offsets - for each task.

Our goal is to provide tasks with fixed offsets and fixed priorities. It may happen, however, that we have to assign different offsets/priorities to different instances of the same task, in order to reenact the off-line schedule at run time. These cases cannot be expressed directly with fixed priorities and fixed offsets and are the sources for *offset assignment conflicts* or *priority assignment conflicts*. In both cases, we split the conflicting task into instances such that, further on, each instance will be considered as an independent task with one instance during LCM.

By **offset assignment conflict** we mean that different instances of the same task have to be assigned different offsets in order to ensure the run-time execution of each one of them in the derived target window.

for $1 \leq i \leq nr\_of\_off - line\_scheduled\_tasks$
  for $1 \leq j \leq n,\ where\ n = nr\_of\_instances(T_i)$
    if: $begin(TW(T_i^j)) - (j-1)p(T_i) \neq$
      $\neq begin(TW(T_i^{j+1})) - j * p(T_i)$,

T_i	p	c
A	5	1
B	10	3
C	20	8

Table 2.1: Original tasks

Then: $We\ split\ T_i\ into\ T_{i,1},\ T_{i,2}, \ldots, T_{i,n}$

The offset and period assignment will be described in section 2.4.6.

Priority assignment conflicts are detected after the derivation of the sequences, and occurs in the cases when two different instances of the same task have to be assigned different priorities in order to ensure the run-time execution of each one of them in the derived target window, and in the right position in the sequence the tasks belongs to. In this case, since a priority assignment involves more than one task, there is typically a choice of which task to split.

In our method, we split tasks that causes offset assignment conflicts into instances *before* deriving the sequences of instances. By that, we reduce the probability of priority assignment conflict eventually caused by the same tasks since the new created tasks will have only one instance during LCM.

We illustrate the issues with an example. Assume that we have the off-line schedule in Figure 2.2 expressing the original constraints of the taskset in table 2.1.

Let's assume that we have a precedence constraint between the $(4m+1)^{th}$ instance of A and the $(2m+1)^{th}$ instance of B, and a precedence constraint between the $(2n+2)^{th}$ instance of B and the $(4n+3)^{th}$ instance of A (see 2.1).

T_i^j	$TW(T_i^j)$
A^1	[0,5]
A^2	[5,10]
A^3	[10,15]
A^4	[15,20]
B^1	[0,10]
B^2	[10,20]
C^1	[0-20]

Table 2.2: Target windows derived from the off-line schedule

$$\begin{aligned} A^{4m+1} &\rightarrow B^{2m+1},\ m = 0, 1, 2, \ldots \\ B^{2(n+1)} &\rightarrow A^{4n+3},\ n = 0, 1, 2, \ldots \end{aligned} \tag{2.1}$$

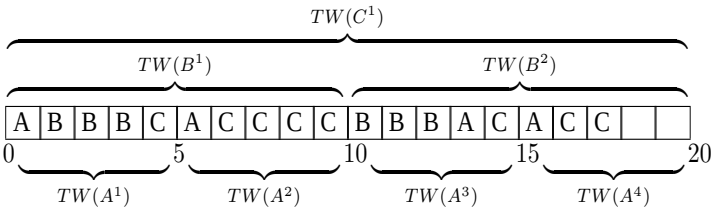


Figure 2.2: Example 1: Off-line Schedule and Target Windows

The time points $t_1 = 0,\ t_2 = 5,\ t_3 = 10,\ t_4 = 15$ mark the beginning of the target windows (Figure 2.2), i.e., $TW_1 = TW(A^1) = [0, 5]$, $TW_2 = TW(B^1) = [0, 10]$, $TW_3 = TW(C^1) = [0, 20]$, $TW_4 = TW(A^2) = [5, 10]$, etc. The full description of the derived target windows is presented in table 2.2.

Now, by analyzing the overlapping between the target windows, we derive sequences of instances for each start of each target window and the priority relations (inequalities) between the tasks of each target window (see table 2.3).

k	t_k	$\left\{ \begin{array}{c} current \\ inst. \end{array} \right\}_{t_k}$	$\left\{ \begin{array}{c} intf. \\ inst. \end{array} \right\}_{t_k}$	SEQ_k	Priority inequalities
1	0	A^1, B^1, C^1	None	A^1, B^1, C^1	$prio(A^1) > prio(B^1)$ $prio(B^1) > prio(C^1)$
2	5	A^2	C^1	A^2, C^1	$prio(A^2) > prio(C^1)$
3	10	A^3, B^2	C^1	B^2, A^3, C^1	$prio(B^2) > prio(A^3)$ $prio(A^3) > prio(C^1)$
4	15	A^4	C^1	A^4, C^1	$prio(A^4) > prio(C^1)$

Table 2.3: Example 1: Sequences of tasks

In this example, we can easily see that we do not have any offset assignment conflicts, since the target windows of the instances of the same task begins at the same point in time relative to the task period. According to the sequence $SEQ_1 =< A^1, B^1, C^1 >$, task A must be assigned a higher priority then task B. On the other hand, according to the sequence $SEQ_3 =< B^2, A^3, C^1 >$, task B must be assigned higher priority then A. In this case we have a cycle of priority inequalities that has to be solved:

$$prio(A^1) > prio(B^1) > \ldots > prio(B^2) > prio(A^3)$$

We solve this issue by splitting the task with the inconsistent priority assignment into a number of new periodic tasks with different priorities. The instances of the new tasks comprise all instances of the original tasks.

Since a priority assignment conflict involves more than one task, like in our example, there is typically the choice of which task to split. Our goal is to find the splits which yield the smallest number of FPS tasks. In our example, we have to break the chain by splitting either A or B into instances and considering each one of these instances as individual tasks. Depending on the number of instances of A and B during LCM, the choice of the task to be split influences the number of artifact tasks

created.

In order to minimize the number of artifact tasks, we create an integer linear programming problem from the derived system of priority inequalities to first identify which instances to split, if any, and to derive priorities for the resulting FPS tasks. The flexibility of the ILP solver allows for simple inclusion of other criteria via goal functions.

In section 2.4.6 we present a complete solution for our example.

2.4.5 ILP problem representation

The ILP problem formulation presented in this section is a contribution from Peter Puschner.

A linear programming (LP) problem consists of a linear goal function in a number of variables and a set of linear inequality relations of the variables. LP solving searches a value assignment for all variables (solution) that optimizes (minimizes or maximizes) the given goal function under the given constraints. If the values of a solution have to be integral the problem is called an integer linear programming (ILP) problem.

The aim of the given attribute assignment problem is to find a task set, i.e., a minimum number of tasks together with their priorities, that fulfills the priority relations of the sequences of the schedule. As mentioned above, each task of the task set is either one of the original tasks or an artifact task created from one of the instances of an original task selected for splitting.

The problem is translated into an ILP problem, because we are only interested in integral priority assignments and solutions. In the ILP problem the goal function G to be minimized computes the number of tasks to be used in the FPS scheduler

$$G = N + \sum_{i=1}^{N} (k_i - 1) * b_i,$$

where N is the number of tasks in the off-line schedule, k_i is the number of instances of task T_i, and b_i is a binary integral variable that

indicates if T_i needs to be split into its instances.

The constraints of the ILP problem reflect the restrictions on the task priorities as imposed by scheduling problem. The priority relations of the original tasks of the off-line schedule form the basis for the ILP constraints. To account for the case of priority conflicts, i.e., when tasks have to be split, the constraints between the original tasks are extended to include the constraints of the artifact tasks. Thus each priority relation

$$prio(\overset{k}{S}_l) > prio(\overset{k}{S}_{l+1})$$

with $\overset{k}{S}_l = T_i^j$ and $\overset{k}{S}_{l+1} = T_p^q$, is translated into an ILP constraint

$$p_i + p_i^j > p_p + p_p^q,$$

where the variables p_i and p_p stand for the priorities of the FPS tasks representing the original tasks T_i and T_p, respectively, and p_i^j, p_p^q stand for the priorities of the artifact tasks T_i^j and T_p^q (in case it is necessary to split the off-line tasks). Although this may look like a constraint between four tasks (T_i, T_i^j, T_p, T_p^q) it is in fact a constraint between two tasks – for each task only its original (T_i resp. T_p) or its artefact tasks (T_i^j resp. T_p^q) can exist in the FPS schedule. A further set of constraints for each off-line task T_i ensure that only either the original tasks or its artefact tasks are assigned valid priorities (greater than 0) by the ILP solver. All other priorities are set to zero.

$$\begin{aligned} p_i &\leq (1 - b_i) * M \\ \forall j : p_i^j &\leq b_i * M \end{aligned}$$

In these constraints M is a large number, larger than the total number of instances in the off-line schedule. The variable b_i for task T_i, which also occurs in the goal function, is the binary variable that indicates if T_i has to be split, i.e., b_i allows only a task or its artifact tasks to assume valid priorities. Since the goal function associates a penalty equal to the number of instances of T_i for each b_i that has to be set to

1, the ILP problem indeed searches for a solution that produces a minimum number of task splits. The constraints on the variables b_i complete the ILP constraints: $b_i \leq 1$.

The solution of the ILP problem yields the total number of tasks as the result of the goal function. The values of the variables p_i and p_i^j for each task represent a priority assignment for tasks and artifact tasks that satisfies the priority relations of the scheduling problem. If $p_i > 0$ or $p_i^j > 0$ then the respective task T_i or T_i^j exists in the FPS schedule and its priority is p_i or p_i^j, respectively. If a variable (p_i or p_i^j) has been assigned the value zero the task/artifact task is not included into the FPS schedule, i.e., $FPS\_tasks = \{T_i : p_i > 0\} \cup \{T_i^j : p_i^j > 0\}$ and for each task in FPS_tasks the priority $prio(T_i)$ is the value of the priority variable of the corresponding task/artifact task, i.e., p_i or p_i^j.

In our example (Example 1), the solution provided by the solver is:

$$
\begin{aligned}
&b_A = b_C = 0\\
&b_B = 1,\ meaning\ that\ task\ B\ is\ to\ be\ split\\
&p_A = 3\\
&p_B^1 = 2\\
&p_B^2 = 4\\
&p_C = 1
\end{aligned}
$$

2.4.6 Periods and offsets

Since the priorities of the FP tasks have been assigned by the LP-solver, we can now focus on the assignment of periods and offsets. Now we have a set of tasks with priorities, *FPS_tasks*, produced by the LP-solver, consisting of a subset of the original taskset, $\{orig\_tasks\} \subseteq \{Original\ Tasks\}$, and a set of artifact tasks, $\{art\_tasks\}$:

$$FPS\_tasks = \{orig\_tasks \cup art\_tasks\}$$

Based on the information provided by the LP-solver, we assign periods and offsets to each task in $\overline{T_i} \in \{FPS\_Tasks\}$, in order to ensure

$\overline{T_i}$	p	c	o	prio
A	5	1	0	3
B1	20	3	0	2
B2	20	3	10	4
C	20	8	0	1

Figure 2.3: Resulting FPS tasks.

the run time execution within their respective target windows, as following:

$$
\begin{aligned}
for \quad & 1 \leq i \leq nr\_of\_tasks\_in(FPS\_tasks) \\
& p(\overline{T_i}) = \frac{LCM}{nr\_of\_instances(\overline{T_i})} \\
& o(\overline{T_i}) = begin(TW(\overline{T_i^1}))
\end{aligned}
$$

The final set of tasks, derived from the original off-line scheduled tasks in example 1, by performing the steps described in Sections 2.4.4, 2.4.5 and 2.4.6, are illustrated in Figure 2.3. The highest value represents the highest priority.

2.5 Example

We illustrate the ability of the method to transform off-line scheduled tasks with complex constraints to FPS tasks, with an example.

We assume we have the taskset described in Figure 2.4 and the earliest start times and the deadlines of the off-line tasks, are equal to the start and end of the periods. In this case, lets assume the precedence constraints are:

$Task$	p	c	Node
A	15	2	0
B	15	1	0
C	15	5	0
D	10	3	0
E	10	2	0
F	15	3	1
G	15	4	1

Figure 2.4: Example 2: Off-line Tasks

$$A \rightarrow B \rightarrow C; D \rightarrow E; F \rightarrow G; F \rightarrow C$$

and it takes one time slot to send a message between 2 nodes. Additionally, we want the FPS execution of task A to be fixed between $(est(A) + 2)$ and $(est(A) + 4)$.

First, an off-line schedule is constructed, by an arbitrary off-line scheduler, to meet the constraints, and we derive the target windows for each task instance (figure 2.5).

The priority inequalities between the instances are derived in the same way as in the example presented in Section 2.4 and shown in table 2.4.

From the inequalities, we can see that we have a number of priority assignment conflicts, i.e.,

$$prio(E^1) > prio(B^1) > prio(C^1) > prio(D^2)$$

resulting from SEQ_3 corresponding to $t_3 = 5$, and

$$prio(D^2) > prio(E^2)$$

from SEQ_4, meaning that we have a cycle of inequalities consisting of

$$prio(E^1) > prio(B^1) > \ldots > prio(E^2)$$

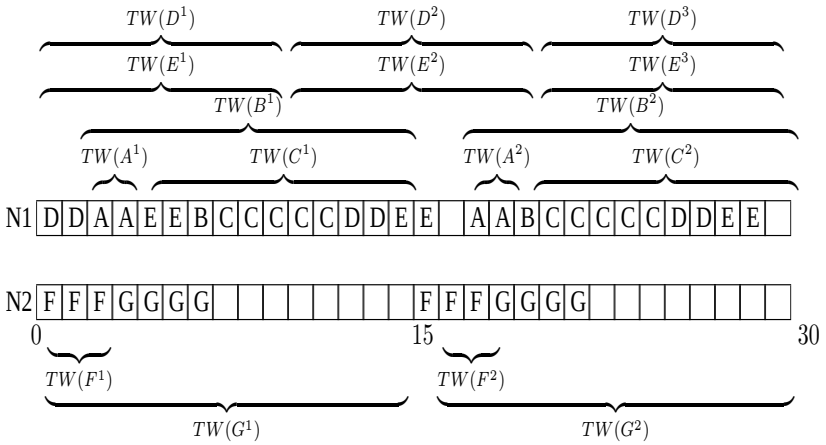


Figure 2.5: Example 2: Off-line Schedule and Target Windows

Another cycle of inequalities is given by SEQ_1, SEQ_2, SEQ_3 and SEQ_4:

$$prio(D^1) > prio(E^1) > \ldots > prio(C^1) > prio(D^2)$$

ILP formulation At this point, we formulate the ILP problem. The goal function is to minimize the number of artifact tasks, and, thus, to get information about which task(s) to split. In our case, the goal function is:

$$minG = 7 + bA + bB + bC + 2bD + 2bE + bF + bG;$$

t_k	Node	$\left\{ \begin{array}{c} current \\ inst. \end{array} \right\}_{t_k}$	$\left\{ \begin{array}{c} intf. \\ inst. \end{array} \right\}_{t_k}$	SEQ_k	inequalities
0	0	D^1, E^1	None	D^1, E^1	$prio(D^1) > prio(E^1)$
	1	F^1, G^1	None	F^1, G^1	$prio(F^1) > prio(G^1)$
2	0	A^1, B^1	E^1	A^1, E^1, B^1	$prio(A^1) > prio(E^1)$ $prio(E^1) > prio(B^1)$
4	0	C^1	E^1, B^1	E^1, B^1, C^1	$prio(E^1) > prio(B^1)$ $prio(B^1) > prio(C^1)$
10	0	D^2, E^2	C^1	C^1, D^2, E^2	$prio(C^1) > prio(D^2)$ $prio(D^2) > prio(E^2)$
15	0	None	E^2	E^2	-
	1	F^2, G^2	None	F^2, G^2	$prio(F^2) > prio(G^2)$
17	0	A^2, B^2	None	A^2, B^2	$prio(A^2) > prio(B^2)$
20	0	C^2, D^3, E^3	None	C^2, D^3, E^3	$prio(C^2) > prio(D^3)$ $prio(D^3) > prio(E^3)$

Table 2.4: Example 2: Inequalities

The priority constraints derived from the task sequences are:

$$
\begin{array}{ll}
constraint\ 1: & \\
p_D + p_D^1 > p_E + p_E^1 + 1 & ; \quad p_F + p_F^1 > p_G + p_G^1 + 1; \\
p_A + p_A^1 > p_E + p_E^1 + 1 & ; \quad p_E + p_E^1 > p_B + p_B^1 + 1; \\
p_E + p_E^1 > p_B + p_B^1 + 1 & ; \quad p_B + p_B^1 > p_C + p_C^1 + 1; \\
p_C + p_C^1 > p_D + p_D^2 + 1 & ; \quad p_D + p_D^2 > p_E + p_E^2 + 1; \\
p_F + p_F^2 > p_G + p_G^2 + 1 & ; \\
p_A + p_A^2 > p_B + p_B^2 + 1 & ; \\
p_C + p_C^2 > p_D + p_D^3 + 1 & ; \\
p_D + p_D^3 > p_E + p_E^3 + 1 & ;
\end{array}
$$

Additionally, we need ensure for each task T_i that only either the original task or its artifact are assigned valid priorities (greater than 0) by the

solver. All other priorities are set to zero.

$$
\begin{aligned}
&constraint\ 2: \\
&p_A < 17 - 17bA; p_A^1 < 17bA; p_A^2 < 17bA; \\
&p_B < 17 - 17bB; p_B^1 < 17bB; p_B^2 < 17bB; \\
&p_C < 17 - 17bC; p_C^1 < 17bC; p_C^2 < 17bC; \\
&p_D < 17 - 17bD; p_D^1 < 17bD; p_D^2 < 17bD; p_D^3 < 17bD; \\
&p_E < 17 - 17bE; p_E^1 < 17bE; p_E^2 < 17bE; p_E^3 < 17bE; \\
&p_F < 17 - 17bF; p_F^1 < 17bF; p_F^2 < 17bF; \\
&p_G < 17 - 17bG; p_G^1 < 17bG; p_G^2 < 17bG;
\end{aligned}
$$

where bA, bB, bC, bD, bE, bF, bG are boolean variables.

The output provided by the ILP is:

$$
\begin{aligned}
&Value\ of\ objective\ function: 11 \\
&p_A = 5, p_B = 3, p_C = 2, p_D = 0, p_E = 0, p_F = 1, p_G = 0 \\
&p_A^1 = 0, p_A^2 = 0, p_B^1 = 0, p_B^2 = 0, p_C^1 = 0, p_C^2 = 0 \\
&p_D^1 = 5, p_D^2 = 1, p_D^3 = 1, p_E^1 = 4, p_E^2 = 0, p_E^3 = 0 \\
&p_F^1 = 0, p_F^2 = 0, p_G^1 = 0, p_G^2 = 0 \\
&bA = 0, bB = 0, bC = 0, bD = 1, bE = 1, bF = 0, bG = 0
\end{aligned}
$$

In the output provided by the solver, we see that the boolean variables associated to the tasks C and D are equal to 1, meaning that instances of these two tasks have to be transformed into artifacts, to solve the priority inconsistencies and to ensure the minimum number of final tasks.

Finally we assign offsets and periods to the FPS tasks in order to ensure the execution within their target windows (Figure 2.6). The schedule obtained by scheduling the final taskset by FPS is illustrated in Figure 2.7.

$\overline{T_i}$	p	c	o	prio
A	15	2	2	5 (highest)
B	15	1	2	3
C	15	5	4	2
D1	30	2	0	5
D2	30	2	10	1
D3	30	2	20	1
E1	30	2	0	4
E2	30	2	10	0
E3	30	2	20	0
F	15	3	0	1
G	15	4	0	0 (lowest)

Figure 2.6: Example 2: FPS Tasks

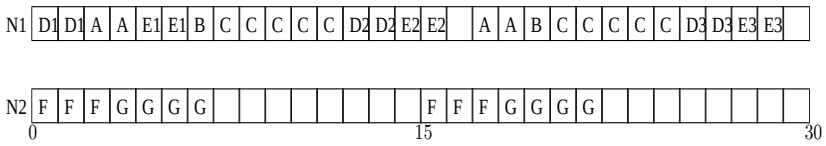


Figure 2.7: Example 2: FP Schedule

2.6 Discussion

Our method does not introduce artifacts or reduce flexibility unless required by constraints: a set of FPS tasks, scheduled off-line according to FPS, and transformed by our method will execute in the same way as the original FPS tasks. The FPS tasks resulting from our method execute flexibly, unless prevented by reducing target windows for strict predictability for some tasks. Tasks can execute earlier if preceding tasks finish earlier than the assumed worst case execution time, or may even change order of execution, if tasks are not ready to run, provided the

priority order is kept.

In some cases, we have to perform additional splits, due to violation of the periodicity in the off-line schedule, which gives different offsets for different instances of the same task.

By minimizing the number of artifact tasks, our method minimizes the number of offsets in the system as well, since we don't change offsets unless we have to split tasks. By using ILP, we minimize the number of artifact tasks and, implicitly, offsets.

While our method is capable of deriving FPS tasks for general off-line schedules, the resulting task set and attributes may be awkward in extreme cases, e.g., the off-line schedule includes non periodic patterns or changes execution orders of tasks. Still, our method allows these to be re-enacted with standard FPS scheduling.

Note that our method allows, e.g., tasks to execute according to earliest deadline first order, although using FPS, by creating artifact tasks with different priorities. Thus, the properties of, in this case, EDF can be exploited in an FPS system. The resulting increase in utilization comes from the periods of the artifact tasks being set to LCM.

Target windows can be derived from off-line schedules directly, without further knowledge about the original timing constraints. In that case, the off-line schedule will be re-enacted exactly by the FPS tasks, providing the same determinism. The resulting assignment, however, will lead to inflexible schedules and inefficient attributes.

We envision the proposed method to be complemented by a run-time enforcement mechanism, such as a watch-dog, to ensure tasks do not overrun their budgets.

On-line tasks with lower priority can easily be added to the fixed priority schedule, while an on-line acceptance test can be performed on the higher priority sporadic or aperiodic tasks.

2.7 Proofs

We prove the correctness of the method in two steps. First, we prove that FPS tasks will meet their deadlines, and second, that we preserve

the order of execution enforced by the task constraints expressed in the off-line schedule.

2.7.1 Proof 1

Theorem: Any instance T_i^j of any task T_i, produced by the method described in Section 2.4, execute at run-time within its derived target window, if scheduled by FPS together with all other FP tasks. In fact it complete its execution before it's off-line scheduled finishing time, $R(T_i^j) \leq ft(T_i^j)$.

Proof: We want to prove that

$$\forall t_k,\ SEQ_k =< \overset{k}{S}_1, \overset{k}{S}_2, \ldots, last(SEQ_k) >$$

$$\begin{aligned} R(\overset{k}{S}_i) \quad \leq \quad & ft(\overset{k}{S}_i) \leq dl(\overset{k}{S}_i) = end(TW(T_i^j)), \\ & \forall i \in [1, nr\_of\_tasks\_in\_SEQ_k] \end{aligned}$$

where $R(\overset{k}{S}_i) = est(\overset{k}{S}_i) + c(\overset{k}{S}_i) + \sum_{\forall j \in hp(i)} c(T_j)$.

T_j is a task instance belonging to either SEQ_k or $SEQ_{k+n},\ n \geq 1$. All task instances with earliest start times less or equal to t_k are included in SEQ_k as either $current\_instances_{t_k}$ or $interfering\_instances_{tk}$. However, there might be an interference from task instances belonging to 'later' sequences SEQ_{k+n} that are not taken into account when deriving the priority inequalities corresponding to SEQ_k.

What do we know?

1. The start and end of the target windows of a task represent the task's earliest start time and deadline respectively
   $\forall\, TW_n,\ \forall\, T_i^j \in TW_n,$
   $est(T_i^j) = begin(TW_n)\ and\ dl(T_i^j) = end(TW_n)$

2. The sum of the wcet of the tasks in a sequence are less than or equal to the finishing time of the last task in the sequence (from the off-line schedule).
   $\forall SEQ_k,\ \forall\, i \in [1, nr\_of\_tasks\_in\_SEQ_k]$,
   $t_k + c(\overset{k}{S}_i) + c(\overset{k}{S}_{i-1}) + \ldots + c(\overset{k}{S}_1) \leq ft(\overset{end}{S}_i)$,

3. The priorities of the tasks in a sequence are assigned in descending order absed on the order of execution specified in the off-line schedule.
   $\forall SEQ_k$,
   $prio(\overset{k}{S}_1) > prio(\overset{k}{S}_2) > \ldots > prio(last(SEQ_k))$,
   (given by ILP)

4. $\forall i \in [1, nr\_of\_tasks\_in\_SEQ_k]$,
   $ft(\overset{k}{S}_i) \leq end(TW(\overset{k}{S}_i))$,
   (given by the off-line schedule)

We use induction.

(a) As the first step, we prove that the theorem is true for the "last" sequence in the off-line schedule, SEQ_{end} corresponding to the time t_{end} since in this case we don't have interference from task instances belonging to "later" sequences:

$$\forall\, i,\ \overset{end}{S}_i \in SEQ_{end},\ R(\overset{end}{S}_i) \leq ft(\overset{end}{S}_i)$$

Proof:

The last sequence derived from the off-line schedule is:

$$SEQ_{end} =< \overset{end}{S}_1, \overset{end}{S}_2, \ldots, last(SEQ_{end}) >$$

Then,

$$\forall i \in [1, nr\_of\_instances\_in\_SEQ_{end}].$$

From (2)

$$t_{end} + c(\overset{end}{S}_i) + c(\overset{end}{S}_{i-1}) + \ldots + c(\overset{end}{S}_1) \leq ft(\overset{end}{S}_i)$$

From (3):

$$c(\overset{end}{S}_1) + c(\overset{end}{S}_2) + \ldots + c(\overset{end}{S}_{i-1}) = \sum_{\forall j \in hp(i)} c(T_j)$$

since there is no interference from task instances belonging to 'later' sequences.

That results in:

$$t_{end} + c(\overset{end}{S}_i) + \sum_{\forall j \in hp(i)} c(T_j) \leq ft(\overset{end}{S}_i)$$

Additionally we know that $est(\overset{end}{S}_i) \leq t_{end}$ (from the definition of the sequences):

$$est(\overset{end}{S}_i) + c(\overset{end}{S}_i) + \sum_{\forall j \in hp(i)} c(T_j) \leq ft(\overset{end}{S}_i)$$

so:

$$\underline{R(\overset{end}{S}_i) \leq ft(\overset{end}{S}_i)}$$

meaning that $\overset{end}{S}_i$ complete its execution before its off-line scheduled finishing time, and implicitly, before the end of its target window (from (1) and (4)).

(b) We assume

$$\begin{aligned} \forall i &\in [1, nr\_of\_inst.\_in\_SEQ_k], \\ if\ SEQ_k &= <\overset{k}{S}_1, \ldots, last(SEQ_k) > \ then\ R(\overset{k}{S}_i) \leq ft(\overset{k}{S}_i) \end{aligned}$$

We prove SEQ_{k-1}:

$$\forall\, i,\ \overset{k-1}{S}_i \in SEQ_{k-1},\ R(\overset{k-1}{S}_i) \leq ft(\overset{k-1}{S}_i)$$

Here, we have two cases:

Case1:

The sequences do not overlap:

$$SEQ_k \cap SEQ_{k-1} = \varnothing$$

In this case we have no interference; the proof given in (a) still holds.

Case2:

The sequences do overlap:

$$SEQ_k \cap SEQ_{k-1} = common\_tasks$$

where:

$$\begin{aligned} common\_tasks &= \{T^j \mid T^j \in SEQ_k \wedge T^j \in SEQ_{k-1} \wedge \\ &\quad \wedge st(T^1) < st(T^2) < \ldots < st(T^m)\} \end{aligned}$$

so *common tasks* is a squence of tasks ordered increasingly according to their off-line scheduled start times.

$$common\_tasks =< T^1, T^2, \ldots, T^m >$$

Then:

$$\begin{aligned} SEQ_{k-1} &= < \overset{k-1}{S}_1, \ldots, \overset{k-1}{S}_n > \cup \\ &\quad \cup \underbrace{< \overset{k-1}{S}_{n+1}, \ldots, last(SEQ_{k-1}) >}_{\text{common\_tasks}} \\ &= < \overset{k-1}{S}_1, \ldots, \overset{k-1}{S}_n, \underbrace{\overset{k-1}{S}_{n+1}, \ldots, last(SEQ_{k-1})}_{\text{common\_tasks}} > \end{aligned}$$

We know (from the assumption) that:

$$\forall T_i \in common\_tasks \subseteq SEQ_k \Rightarrow R(T^i) \leq ft(T^i)$$

and, from the priority assignment mechanism and (3):

$$prio(\overset{k-1}{S}_i) > prio(T^j),\ \forall i \in [1,n],\ \forall T^j \in \{common\_tasks\},$$

Then, $\forall i \in [1,n] : (from(2))$

$$\begin{aligned} R(\overset{k-1}{S}_i) &= \underbrace{t_{k-1}}_{\geq \text{est}(\overset{k-1}{S}_i))} + c(\overset{k-1}{S}_i) + \underbrace{c(\overset{k-1}{S}_{i-1}) + \ldots + c(\overset{k-1}{S}_1)}_{\sum_{\forall j \in hp(i)} c(T_j)} \\ &\leq ft(\overset{k-1}{S}_i) \end{aligned}$$

Hence:

$$R(\overset{k-1}{S}_i) = est(\overset{k-1}{S}_i) + c(\overset{k-1}{S}_i) + \sum_{\forall j \in hp(i)} c(T_j) \leq ft(\overset{k-1}{S}_i). \square$$

2.7.2 Proof 2

Theorem: If there is a precedence relation expressed in the original task constraints and/or the off-line schedule between any two instances T_m^i, T_n^j of any two tasks $T_m,\ T_n,\ T_m^i \rightarrow T_n^j$, then T_m^i execute before T_n^j when scheduling the FPS-task produced by the method described in 2.4 by FPS, assuming that the precedence requirement is fulfilled in the off-line schedule.

Proof: We prove that if there is an overlapping, in terms of time, between the target windows of the two instances, $TW(T_m^i)\ and\ TW(T_n^j)$, then T_m^i is assigned a higher priority then T_n^j, $prio(T_m^i) > prio(T_n^j)$. Additionally, we know that any instance of any FP task execute inside its target window if scheduled by FPS (2.7.1) and, if there is an overlapping between the two target windows, then the target window of T_m^i must begin before the beginning of the target window of T_n^j:

$$begin(TW(T_m^i)) \leq begin(TW(T_n^j))$$

We have two cases:

1. The target windows start at the same point in time.

   $$begin(TW(T_m^i)) = begin(TW(T_n^j)) = t_k$$

   then: $T_m^i,\ T_n^j \in current\_instances_{t_k} \in SEQ_k$, and the off-line execution of T_m^i is before the off-line execution of T_n^j (from the off-line schedule). Then:

   $$prio(T_m^i) > prio(T_n^j)$$

   since

   $$prio(\overset{k}{S}_1) > prio(\overset{k}{S}_2) \ldots > prio(last(SEQ_k))$$

   where the off-line execution of $\overset{k}{S}_i$ is before the off-line execution of $\overset{k}{S}_{i+1},\ \forall i \in [1, nr\_of\_instances\_inSEQ_k]$.

2. The target window of T_m^i starts before the target window of T_n^j

$$begin(TW(T_m^i)) = t_k < begin(TW(T_n^j)) = t_p$$

Here, again, we have two cases:

(a) The tasks do not interfere with each other, i.e., T_m^i finish its execution before the start of the target window of T_n^j. :

$$ft(T_m^i) \leq begin(TW(T_n^j)) = t_p$$

(b) T_m^i finish its execution, according to the off-line schedule, inside the target widow of T_n^j

$$ft(T_m^i) > begin(TW(T_n^j)) = t_p$$

then:

$$T_m^i \in \{interfering\_instances\}_{t_p} \in SEQ_p$$

and

$$T_n^j \in \{current\_instances\}_{t_p} \in SEQ_p$$

and T_m^i is off-line scheduled to execute before T_n^j (from the off-line schedule). Then:

$$prio(T_m^i) > prio(T_n^j)$$

since

$$prio(\overset{p}{S}_1) > prio(\overset{p}{S}_2) > \ldots > prio(last(SEQ_p))$$

where $\overset{p}{S}_i$ is off-line scheduled to execute before $\overset{p}{S}_{i+1}$, $\forall i \in [1, nr\_of\_instances\_in(SEQ_p)]$.□

2.8 Chapter summary

In this chapter we have presented a method that combines off-line schedule construction with fixed priority run-time scheduling. We use off-line schedules and target windows to express complex constraints and predictability for selected tasks, and derive attributes for tasks, such that if applying FPS at run-time, the tasks will execute within the specified target windows and fulfill the original constraints.

Thus, the method solves issues arising from legacy systems, e.g., partition scheduling for avionics applications, allows to handle constraints not covered by FPS feasibility tests, while using standard FPS at runtime. At the same time, it provides for predictable flexibility, i.e., the restricted execution of selected tasks, e.g., for sampling and actuating in control systems, while enabling runtime flexibility for others.

Our method analyzes the off-line schedule and the target windows and derives priority relations between task instances, expressed in a set of inequalities. In certain cases, the method splits tasks into instances, creating artifact tasks, as not all off-line schedules can be expressed directly with FPS. We use standard integer linear programing to solve the priority inequalities and minimize the number of artifact tasks created. Finally, we assign offsets and periods to the task set provided by ILP in order to ensure the correct run-time execution within the derived target windows.

In some cases, we have to perform additional splits, due to a violation of the periodicity in the off-line schedule, which gives different offsets for different instances of the same task. By minimizing the number of artifact tasks, our method minimizes the number of offsets in the system as well. The number of artifact tasks and offsets can be decreased by reducing target windows, if the resulting loss in flexibility is acceptable.

Our method does not introduce artifacts or reduce flexibility unless required by constraints: a set of FPS tasks, scheduled off-line according to FPS, and transformed by our method executes in the same way as the original tasks.

To this point, we have concentrated on reconstructing the off-line schedule. Using the flexibility of the ILP solver, we can add objectives by inclusion in the goal function. We have implemented the described method and tested the results both via response time analysis and the original off-line schedule and original task constraints. Our algorithm determines task attributes trying to keep priorities for all instances of periodic tasks the same. This will lead to inconsistent priority assignments for some schedules, e.g., those created with a earliest deadline first strategy. Our algorithm attempts to resolve the arising priority conflicts by splitting the task into several instances with different priorities. Here, we assume that task dependencies have been resolved off-line.

Chapter 3

Scheduling complex constrained messages on Controller Area Network (CAN)

3.1 Introduction

Controller Area Network (CAN) has gained wider acceptance as a standard in a large number of industrial applications. The priority based message scheduling used in CAN has a number of advantages, some of the most important being the efficient bandwidth utilization, flexibility, simple implementation and small overhead. Early results on message scheduling on CAN have been presented in [57] and [56], in which the authors focused on fixed priority scheduling based on work presented in [38] and [37]. Later on, Zuberi [63] showed that static priority scheduling is not always the most suitable strategy. Earliest Deadline (EDF) can prove significantly better then fixed priority scheduling [41].

Off-line scheduling for time triggered systems, on the other hand, provides determinism [28], [30], and, additionally, complex constraints

can be solved off-line, but this scheduling strategy is not suitable for CAN.

In this chapter we apply the previous results presented in chapter 2 to message scheduling on CAN. We present a method that transforms off-line scheduled transmission schemes into sets of messages that can be scheduled on CAN. It assumes a schedule has been constructed for a set of off-line scheduled messages to meet their complex constraints. Our method takes the off-line schedule, with derived time intervals in which messages must be transmitted, i.e., *Target Windows*, and assigns FPS attributes, (i.e., priorities) to the messages. It then, provides information about periods and offsets the messages have to be sent with by the sending nodes, such that the message transmission at runtime matches the off-line schedule. It does so by deriving priority inequalities, which are then resolved by integer linear programming (ILP).

FPS cannot reconstruct all schedules with periodic messages with the same priorities for all instances (invocations) directly. The constraints expressed via the off-line schedule may require different message sequences for invocations of the same message, as, e.g., by earliest deadline first, leading to inconsistent priority assignment. This phenomenon can be expressed as a cycle of inequalities. Our algorithm detects such situations, and circumvents the problem by splitting a message into its invocations. Then, the algorithm assigns different priorities to the newly generated "artifact" messages, the former invocations.

Key issues in resolving the priority conflicts are the number of artifact messages created. Depending on where a priority conflict circle is "broken", the number may vary, depending on the periods of the split messages. Our algorithm minimizes the number of artifact messages by solving the priority ILP.

Priority assignment for FPS tasks has, for example, been studied in [5], [4] and [20]. [47] study the derivation of task attributes to meet a overall constraints, e.g., demanded by control performance. Instead of specific requirements, our algorithm takes an entire off-line schedule and all message requirements to determine message attributes. A method to transform off-line schedules into earliest deadline first tasks

has been presented in [18]. In the previous chapters we presented methods to deal with priority assignment for off-line CPU scheduled tasks.

The rest of the chapter is organized as follows. In section 3.2 we give a brief overview of message scheduling on CAN. The method we are proposing is presented in section 3.3 and, then, illustrated with an example in section 3.4. We summarize the chapter in section 3.5.

3.2 Controller Area Network (CAN) and message scheduling

CAN consists of the physical and data link layers. Each CAN frame consist of seven fields. In this paper we focus on the identifier field (ID). The identifier field may have two lengths: 11 bits, which is the standard format, and 29 bits, the extended format, and it controls message addressing and bus arbitration . In this approach we focus only on the former since the nodes can set message filters in order to receive only the identifiers they are interested in.

The nodes are connected via a wired OR (or wired AND) CAN bus. The time axis is divided in slots which must be larger or equal to the time it takes the signal to propagate back and forth the bus, $t = \frac{2L}{V}$, where L is the bus length and V is the propagation speed of the signal. When a node has to send a message, it calculates the message ID which may be based on the priority of the message. The message ID must be unique in order to prevent eventually ties. The message is then sent to the bus interface chips, which, further on, write the message ID on the bus, bit by bit, whenever the bus is idle at the beginning of a time slot. After writing a bit on the bus, the chip waits for the signal to propagate along the bus and, then, reads the bus. If the bit read is different from the bit sent, then there is another message on the bus with a higher priority, and the sending node aborts the transmission. Otherwise the node gets the right to send the message without being preempted.

In our approach, we assume a schedule has been constructed for a set of off-line messages. The proposed method transforms the off-line scheduled messages into as set of messages suitable for priority-based

CAN message scheduling. Since we do not want to assign any other attributes than priorities to the message ID's (e.g., periods and offsets), due to the restrictions enforced by the ID format, we rather provide information about attributes that have to be assigned by the programmer to the sending nodes and messages (periods, offsets and priorities) in order to ensure the run-time transmission of the messages according to the specifications expressed in the off-line schedule. Furthermore, we assume that the nodes are clock synchronized and the off-line schedule has been constructed by taking into account the increased bandwidth consumption due to the exchange of messages required by the time synchronization method.

3.3 Attribute assignment algorithm

3.3.1 Overview

Figure 3.1 gives an overview of the algorithm.

1,2 Initially, the off-line schedule table for a set of messages with constraints, is given.

3 Target windows for each invocation of each message are derived from the original message constraints and the off-line schedule.

4 Sequences are now straightforward to derive from the target windows and the transmission order expressed in the off-line schedule.

5 The analysis of each sequence provides a set of inequalities between priorities of invocations of different messages.

6 We use integer linear programming to solve the system of inequalities and the result is the final set of messages with fixed priorities.

Off-line schedule: The input to our method is the off-line schedule expressing the constraints specified for the messages to be sent on CAN.

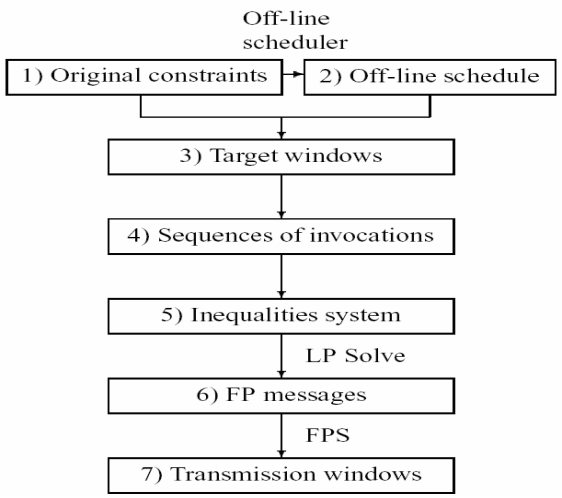


Figure 3.1: Algorithm overview.

The schedule is usually created up to the least common multiple, LCM, of all message periods. We have $LCM/T(M_i)$ invocations of each message M_i with period $T(M_i)$ in the off-line schedule.

The off-line scheduler resolves constraints such as distribution, end-to-end deadlines, precedence, etc, and creates scheduling tables for each node in the system, listing start- and finishing-times of all message invocations. These scheduling tables are more fixed than required by the original constraints, so we can replace the exact sending- and receiving-times of messages with target windows, taking the original constraints into account.

Target windows $(TW(M_i^j))$ of each invocation M_i^j of each message M_i, are derived from the off-line schedule and the original constraints transformed into earliest start times and deadlines.

$$TW(M_i^j) = [t_m, t_n]$$

where

$$t_m = begin(TW(M_i^j)) \; and \; t_n = end(TW(M_i^j))$$

The *earliest transmission start time*, $est(M_i^j)$, of an invocation M_i^j of a message M_i, is provided by the message constraints expressed in the off-line schedule. The *scheduled receiving time*, $srt(M_i^j)$, of an invocation M_i^j of a message M_i, is the time when M_i^j is received by the receiving node according to the off-line schedule. The *scheduled transmission start time*, $start(T_i^j)$, of an invocation M_i^j of a message M_i, is the time when M_i^j is sent on the bus, according to the off-line schedule.

3.3.2 CAN vs. processor scheduling

There are a number of particularities in CAN compared to processors scheduling, which we can take advantage of.

1. Message length is constant. That eliminates the issues that may arise in processor scheduling when tasks, at run-time, execute less than WCET on which the analysis has been performed. In our case, the messages will not start to be transmitted earlier at run-time, as in processor scheduling, due to tasks executions for less than WCET.

2. CAN scheduling is non-preemptive. This ease the calculation of the sequences of messages as we can eliminate interference from earlier transmitted messages. For example, lets assume we have 2 messages off-line scheduled to be ready to be transmitted at time 0, and a message C with an earliest transmission time $est(C) = 5$. The off-line schedule is illustrated in figure 3.2.

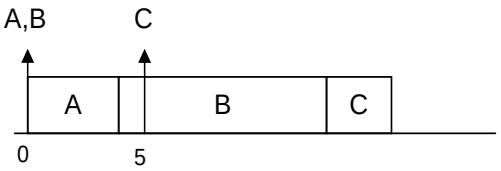


Figure 3.2: Off-line schedule

In this case, we have 2 sequences, one starting at time 0 and the other one starting at time 5. Now, in processor scheduling, the sequences would be:

$$\begin{aligned} SEQ_0 &= < A, B > \\ SEQ_5 &= < B, C > \end{aligned}$$

However, in CAN scheduling message B would not interfere with the message C as C will not preempt it. Hence, the sequences in CAN are:

$$\begin{aligned} SEQ_0 &= < A, B > \\ SEQ_5 &= < C > \end{aligned}$$

The main advantage of having less messages in the sequences is that it will lead to fewer priority inequalities to be solved by ILP. The fewer constraints we send to ILP, the bigger chance to find an attribute assignment that yields least artifacts.

3. CAN scheduling requires unique identifiers (priorities). This we can easily deal with by adding an extra constraint to th ILP solver requiring unique priorities.

$$\forall i, p(M_i) \neq p(M_{i+1})$$

Consequently, we redefine our sequences introduced in chapter 2.

A **sequence** $S(t_k)$ consists of invocations of messages M_i^j ordered by increasing scheduled transmission start times according to the off-line schedule. A sequence may contain invocations M_i^j such that

$$begin(TW(M_i^j)) = est(M_i^j) = t_k$$

that we denote as *current invocations* of $TW(M_i^j)$, and invocations M_p^q from overlapping target windows such that

$$est(M_p^q) < t_k \; and \; start(M_p^q) > t_k$$

denoted as *interfering invocations* of $TW(M_i^j)$.
Additionally:

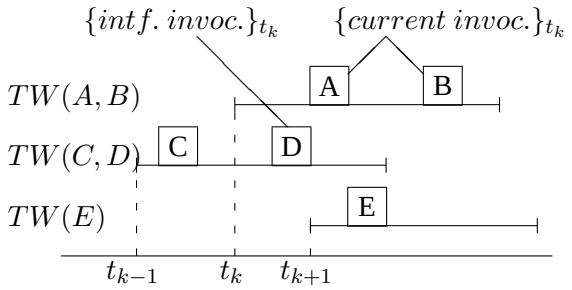
- $first(S(t_k)) = S(t_k)^1$ = first message invocation in the sequence $S(t_k)$
- $last(S(t_k)) = S(t_k)^N$ = last message invocation in $S(t_k)$.

The derivation of a sequence corresponding to a time t_k is illustrated in figure 3.3. Note that, in figure 3.3, message 'E' is not included in the sequence corresponding to the time t_k, since it's earliest transmission start time is greater then t_k. 'E' will be instead a *current invocation* in the sequence corresponding to the time t_{k+1}.

We refer to an *transmission window*, $W_{trans}(M_i^j)$, of an invocation M_i^j of a message M_i, as the time interval in which M_i^j will be sent and received *at runtime*. We want to find fixed priorities, offsets, and deadlines such that the transmission window of each message invocation M_i^j, $W_{trans}(M_i^j)$, will be contained within the respective target window $TW(M_i^j)$, and transmission order specified off-line, kept.

3.3.3 Priority inequalities

Our algorithm derives relations (inequalities of priorities) among the invocations of the messages by traversing the off-line schedule represented by the series of target windows in increasing order of time. It



$$S(t_k) =< S(t_k)^1, S(t_k)^2, S(t_k)^3 >=< D, A, B >$$

Figure 3.3: Sequence of messages.

determines priority inequalities between invocations according to the sequences $S(t_k)$ associated with target windows, such that:

$$P(S(t_k)^1) > P(S(t_k)^2) > \ldots > P(S(t_k)^N)$$

where

$$S(t_k)^1 = first(S(t_k))\ and\ S(t_k)^N = last(S(t_k))$$

Note that the inequalities have to take into account relations between priorities of invocations of the current target window and possibly interfering target windows.

3.3.4 Attribute assignment - conflicts

Our goal is to provide messages with fixed priorities periodically sent on the bus. It may happen, however, that we have to assign different priorities or/and offsets to different invocations of the same message in order to reenact the off-line schedule at run time. These cases cannot be expressed directly with fixed priorities and fixed offsets and are the sources for *offset assignment conflicts* or *priority assignment conflicts*.

In both cases, we split the conflicting message into artifacts, such that, further on, each artifact will be considered an independent message, invoked only once during LCM. Thus we create a number of artifact messages equal to the number of invocations during LCM of the message to be split minus one (since the original message will be replaced by a number of messages equal to the number of its invocations).

By *offset assignment conflict* we mean that different invocations of the same message may have to be invoked at different points in time, relative to the sending task period, in order to ensure the run-time transmission of each one of them in the derived target window.

for $1 \leq i \leq nr\_of\_off - line\_sched\_messages$
  for $1 \leq j \leq n,\ where\ n = LCM/T(M_i)$
    if $begin(TW(M_i^j)) - (j-1) * T(M_i) \neq$
      $\neq begin(TW(M_i^{j+1})) - j * T(M_i)$,
      (where $T(M_i)$ is the period of the message M_i)
    then split M_i into $M_{i,1}, M_{i,2}, \ldots, M_{i,n}$

By splitting M_i, we remove it from the original set of messages, *orig_messages*, and we insert $M_{i,1}$, $M_{i,2}$, ..., $M_{i,n}$ into *orig_messages*.

Priority assignment conflicts are detected after the derivation of the sequences, and occurs in the cases when two different invocations of the same task may have to be sent with different priorities in order to ensure the run-time transmission of each one of them in the derived target window, and in the right position in the sequence the message belongs to. In this case, since a priority assignment involves more than one message, there is typically a choice of which message to split.

In our method, we split messages that causes offset assignment conflicts into artifacts *before* deriving the sequences of invocations. By

that, we reduce the probability of priority assignment conflict eventually caused by the same messages since the new created messages will be invoked only once during LCM.

3.3.5 Minimizing the final number of messages

In order to minimize the number of artifact messages, we create an integer linear programming problem from the derived system of priority inequalities to first identify which messages to split, if any, and to derive priorities for the resulting fixed priority messages. We aim for the minimum amount of artifact messages, and implicitly priorities, due to the limited amount of priorities available when scheduling messages on CAN.

The inequalities obtained from the execution order within the sequences, may form a circular chain of priority relations between messages invocations, e.g.,

$$P(M_i^j) > P(M_m^n) > \ldots > P(M_i^{j+k}) > \ldots > P(M_m^{n+q})$$

Here, we use a higher value to represent a higher priority. In this case we cannot assign the same priority to both invocations j and $(j+k)$ of M_i, nor to invocations n and $(n+q)$ of M_m. We have to break the chain by splitting either M_i or M_m into artifacts and considering each one of them as individual messages, which will result in a larger number of messages compared to the number of original off-line scheduled messages. We formulate a goal function for an integer linear programming solver to identify the minimum amount of messages with fixed priorities.

$$G = \#final\_msgs = \#orig\_msgs + \sum_{i=1}^{N}(|M_i| - 1) * b_i$$

where:

- $\#final\_msgs$ is the number of final messages
- $\#orig\_msgs$ is the number of original messages
- $|M_i|$ = number of invocations of M_i in LCM
- b_i is a boolean variable associated to each message M_i, $b_i \in \{0, 1\}$. b_i = 1 means that M_i needs to be split into $|M_i|$ messages.

Additionally, the solver provides priority values for the messages (split or non-split).

At this point we have a set of messages with fixed priorities, *final_msgs*, produced by the ILP-solver. Finally, we assign periods and offsets to each message (i.e., provide information about when the messages are to be sent by the controller interface) provided by the ILP-solver in order to ensure the run time transmission of the messages within their respective target windows, as following:

$$\begin{aligned} for \quad & 1 \leq i \leq \#(final\_msgs) \\ & T(M_i) = \frac{LCM}{nr\_of\_invocations(M_i)} \\ & offset(M_i) = begin(TW(M_i^1)) \end{aligned}$$

3.4 Example

We illustrate the method with an example. Assume that we have the set of messages, sent from two nodes, shown in table 3.1.

Additionally we assume that we have a precedence constraint between the $(4m + 1)^{th}$ invocation of A and the $(2m + 1)^{th}$ invocation of B,

$$A^{4m+1} \rightarrow B^{2m+1}$$

Message	Node	message size	period (T)
A	1	1	5
B	2	3	10
C	1	4	20

Table 3.1: Original set of messages

where $m = 0, 1, 2, \ldots,$ and a precedence constraint between the $(2n+2)^{th}$ invocation of B and the $(4n+3)^{th}$ invocation of A,

$$B^{2(n+1)} \rightarrow A^{4n+3}$$

where $n = 0, 1, 2, \ldots$.

First, an off-line schedule is constructed by an arbitrary off-line scheduler to meet the constraints. Then, by analyzing the off-line schedule, the target windows are derived for each message invocation.

An off-line schedule that meet the original constraints is illustrated in figure 3.4.

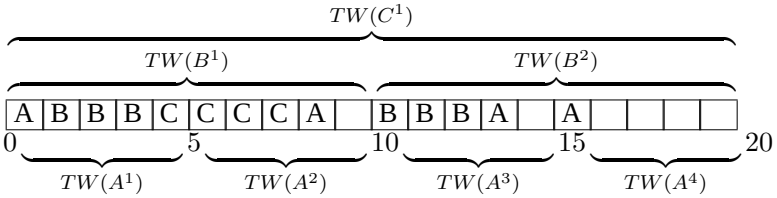


Figure 3.4: Off-line Scheduled Messages and Target Windows

The target windows corresponding to each message invocation are presented in table 3.2.

Next, by analyzing the overlapping between the target windows, we derive sequences for message transmission. The derivation of the inequalities, performed as described in section 3.3.3, is illustrated in the table 3.3.

At time t_k=10, we have the inequality $P(B^2)>P(A^3)$ added to the relations obtained at t_1=0 and t_2=5: $P(A^1)>P(B^1)$, and $P(A^2)>P(C^2)$.

M_i^j	$TW(M_i^j)$
A^1	[0,5]
A^2	[5,10]
A^3	[10,15]
A^4	[15,20]
B^1	[0,10]
B^2	[10,20]
C^1	[0-20]

Table 3.2: Target windows for message invocations

t_k	Message invocations	$S(t_k)$	inequalities
0	A^1, B^1, C^1	A^1, B^1, C^1	$P(A^1) > P(B^1)$ $P(B^1) > P(C^1)$
5	A^2	A^2	
10	A^3, B^2	B^2, A^3	$P(B^2) > P(A^3)$
15	A^4	A^4	

Table 3.3: Inequalities

That gives a circular chain of priorities that must be solved:

$$P(A^1) > P(B^1) > \ldots > P(B^2) > P(A^3)$$

In this case, there are 2 options to solve the problem: we can either choose to split message A, or message B. Splitting B will create two artifact messages, while splitting A will result in four.

ILP formulation At this point we are ready to formulate the ILP problem. The goal function is to minimize the number of artifacts

$$minG = \#final\_msgs = \#orig\_msgs + \sum_{i=1}^{n}(|M_i| - 1) * b_i$$

with the constraints formulated from the priority inequalities. Consequently, the ILP formulation is:

$$\begin{aligned} minG &= 3 + 3bA + bB \\ \\ subject\ to: & \\ & PA + PA1 > PB + PB1 + 1; \\ & PB + PB1 > PC + PC1 + 1; \\ & PB + PB2 > PA + PA3 + 1; \end{aligned}$$

An additional constraint is to ensure for each message M_i that only either the original message or its artifact are assigned valid priorities (greater than 0) by the ILP solver. All other priorities are set to zero.

$$\begin{aligned} PA &< 8 - 8bA; \\ PA1 &< 8bA; \\ PA2 &< 8bA; \\ PA3 &< 8bA; \\ PA4 &< 8bA; \\ \\ PB &< 8 - 8bB; \\ PB1 &< 8bB; \\ PB2 &< 8bB; \end{aligned}$$

where bA, bB are boolean variables

$$bA \leq 1$$
$$bB \leq 1$$
$$bC \leq 1$$

The solution provided by the ILP solver is:

- $b_A = b_C = 0$
- $b_B = 1$, meaning message B is to be split,
- $p_A = 2$
- $p_{B^1} = 3$
- $p_{B^2} = 1$
- $p_C = 4$
- $\#final\_msg = \#orig\_msg + \sum_{i=1}^{N}(|T_i| - 1) * b_i = 4$

Periods and offstes We see that the solution to our problems implies the inclusion of 2 artifacts, B1 and B1 for the former instances of B. As the ILP solver has already privided valid priorities to the new messages, we still have to assign periods and offsets to ensure their transmission within their original target windows.

The final set of messages is shown in table 3.4. The lowest value represents the highest priority.

3.5 Chapter summary

In this chapter applied the results presented in chapter 2 to CAN message scheduling, by scheduling complex constrained off-line scheduled messages on CAN. We use off-line schedules and target windows to express complex constraints and predictability for selected messages. We,

msg	node	msg size	T	offset	dl	prio
A	1	1	5	0	5	2
B1	2	3	20	0	10	3
B2	2	3	20	10	20	1
C	1	4	20	0	20	4

Table 3.4: FP messages

then, derive attributes for the off-line scheduled messages, such that the messages will be transmitted within the specified target windows while fulfilling the original constraints, when scheduled on CAN.

In this approach we take advantage of the CAN particularities compared to processor scheduling, e.g., constant message length and non-preemptive message scheduling, while satisfying the CAN requirement on unique message identifiers.

Our method analyzes the off-line transmission scheme and the target windows and derives priority relations between the invocations of the messages, expressed in a set of inequalities. In certain cases, the method splits messages into instances, creating artifact messages with fixed priorities, as not all off-line schedules can be expressed directly with FPS. We use standard integer linear programing to solve the priority inequalities and minimize the number of artifact messages created. Finally, offsets and periods can be assigned in the implementation, to the set of sending tasks provided by ILP in order to ensure the run-time transmission of the messages within the derived target windows.

In some cases, we may perform additional splits, due to violation of the periodicity in the off-line schedule, which gives different offsets at which different instances of the same message have to be sent. The number of artifact messages caused by offset assignment conflicts, could be decreased by reducing target windows, if the resulting loss in flexibility is acceptable. The priority inversion phenomenon, due to the non-preemption of message transmission, can be solved by modifying the start of the target windows of the messages with precedence relations

considering the precision achieved in the global time synchronization.

Our method does not introduce artifacts or reduce flexibility unless required by constraints: the fixed priority messages provided by our method, with input consisting of a set of messages with fixed priorities, scheduled off-line according to FPS, will be transmitted within the derived target windows and in the off-line specified transmission order.

Chapter 4

Handling non-periodic events together with complex constrained fixed-priority tasks

4.1 Introduction

In the previous chapters we have shown how off-line scheduling and FPS can be combined to take advantage of the benefits provided by both, i.e., predictability and ability to handle complex timing constraints provided by off-line scheduling and flexibility provided by FPS. However, one of the main advantages obtained by using FPS is the capability to handle non-periodic events, i.e., aperiodic tasks.

In FPS, non-periodic events are commonly handled by servers, e.g., background scheduling, polling or deferrable servers, or slack stealing. In this chapter we investigate the possibility to use the existing FPS servers to handle non-periodic events together with the tasks obtained by using the approach presented in chapter 2. What we want is to be able to handle non-periodic events in the best way, i.e., to provide them a good

response time, while not jeopardizing the complex timing constraints of the periodic tasks.

This chapter is organized as follows: section 4.2 gives a description of the existing FPS serves. In section 4.3 we give the problem formulation followed by the proposed solution in section 4.5 and we conclude the chapter in section 4.7.

4.2 Existing FPS servers

A FPS server is scheduling entity that has a capacity to serve events that not arrive periodically. Thus, a server has period, priority and a capacity that represents how much processor time can be used during a given period of time (servers period) for non-periodic events.

For analysis purposes, we can classify the FPS servers in two major categories depending on the behavior of the server upon the presence (or absence) of a non-periodic event: servers that do not preserve their capacity during their period if no aperiodic requests are pending, and servers that do. In the first case, if no aperiodic tasks are waiting to be served at the beginning of the server period, the server capacity is waisted and replenished at the beginning of the next period.

Background scheduling This is the simplest algorithm to handle non-periodic events in FPS. These are served in the background, i.e., whenever the processor is not executing periodic tasks. In this case, the non-periodic events are scheduled at the lowest priority in the system. While the scheduling overhead introduced by this method is minimal, the response time of aperiodic tasks is poor (long).

Polling server In this case, the polling server [36, 49] is a task with FPS parameters, i.e., period, priority and worse case computation time, usually called *capacity*. At run-time, the server gives its capacity to any aperiodic tasks that are pending. The server parameters are assigned in the same way as the parameters of the period tasks, usually by RM.

The server is active at regular periods of time, specified by the server period, and serves any pending aperiodic tasks within the limit of its capacity. If no aperiodics are pending, the server suspends itself until the next activation. In that case, the server capacity is wasted.

Both background scheduling and polling server belong to the category of servers that do not preserve their capacity during their periods, if no aperiodic requests are pending.

Deferrable server This method is the first one in the category of servers that preserve their capacity during their periods if no aperiodic requests are pending. It was introduced by Lechoczky et. al. [36] to overcame the limitations of the polling server. The main difference between deferrable server and polling server is that deferrable server preserves its capacity during its period if no aperiodics are pending. Thus, the response time of the aperiodic tasks is improved compared to the usage of the polling server. However, the schedulability bound for the periodic tasks under RM suffers, as deferrable server violates the basic assumption that task execute as soon they are ready to run and have the highest priority.

Priority exchange Lechoczky et. al. [36] further introduced this server to improve the RM schedulability bound for the periodic task set, compared to the deferrable server. However, the response time of the aperiodic tasks is slightly increased.

The main difference between deferrable server and priority exchange is that PE exchanges its capacity for the execution time of lower priority tasks, if no aperiodic requests are pending. By doing so, the schedulability bound under RM is improved, compared to deferrable server. On the other hand, priority exchange introduces more computational run-time overhead than deferrable server

Sporadic server This technique was introduced by Sprunt et. al. [49]. The main difference between this technique and the ones introduced in deferrable server and priority exchange is that the server replenishes its

capacity whenever it has been entirely consumed by aperiodic tasks, rather then at the beginning of the period.

Slack stealing Was introduced by Lechoczky and Ramos-Thuel [35] to improve the response time of non-periodic events compared to the previous approaches. In this method, the basic idea is to 'steal' all available processor time unused by periodic tasks. While analysis of previous approaches is based on WCET of periodic tasks, slack stealing is using even available processor time due to periodic task executions less than WCET. While the original approach used a static computation of the slack in the system, Davis et. al. [14] presented a dynamic method to calculate and use the slack in the system, i.e., to calculate the slack upon every aperiodic request. However, the computational overhead introduced by this approach is significantly higher than the original one.

Slack stealing was originally considered optimal in the sense that it can use all available slack in the system to serve non-periodic events and, thus, minimize the response time of the aperiodic tasks. However, in 1995, Tia et. al. [54] proved the non optimality of the existing fixed-priority servers, i.e., no existing approach can minimize the response time of non-periodic tasks while still guaranteeing the feasibility of the periodic ones. The main reason is that, in some cases, aperiodics may have to execute later than the earliest possible start of execution in order to achieve the best response time.

4.3 Problem formulation

For a given set of tasks with complex constraints, we want to find FPS parameters for both tasks and existing FPS servers, e.g., polling, deferrable or slack stealing, such that the periodic tasks will execute feasibly (i.e., meeting the original complex constraints) while non-periodic events are provided a short response time by the existing FPs servers.

In other words, we want to find FPS parameters for the tasks, and as high server priority as possible, such that the non-periodic requests are

served as fast as possible while the periodic tasks execute fulfilling the original complex constraints.

4.4 Motivating example

Ideally, we would like to assign the servers the highest priority to make sure that the aperiodic requests are provided a fast service. However, we have to take into consideration the schedulability of the periodic tasks as well, i.e., we can not minimize the reaction time to serve an aperiodic request on the expense of any of the periodic tasks. Moreover, in our task model, we have to deal not only with periodic tasks scheduled according to RM, but with tasks with complex constraints as well for which the priority assignment has been performed according to the method described in chapter 2

Motivating example Let's assume we have a set of 3 tasks as illustrated in table 4.1 where we want to fix the execution times of, e.g., task B at fixed points in time, e.g., an instance separation of 11 time units between B's instances.

$Task$	p	c
A	5	1
B	10	3
C	20	6

Table 4.1: Original tasks

However, since B doesn't have the shortest period, priority assignment according to RM would not guarantee the constraint satisfaction. Instead we use our method described in chapter 2 and assign the second instance of B an offset equal to 1 (table 4.2). As an attribute inconsistency occurs between B's instances, we create artifacts for these as shown in table 4.2.

$Task$	p	c	prio	offset	dl
A	5	1	2	0	5
B1	20	3	3	0	10
B2	20	3	3	11	20
C	20	6	1	0	20

Table 4.2: FPS attributes

In this system, if we don't need to deal with non-periodic events, the task will execute in a FPS system fulfilling the original constraint on task B (figure 4.1).

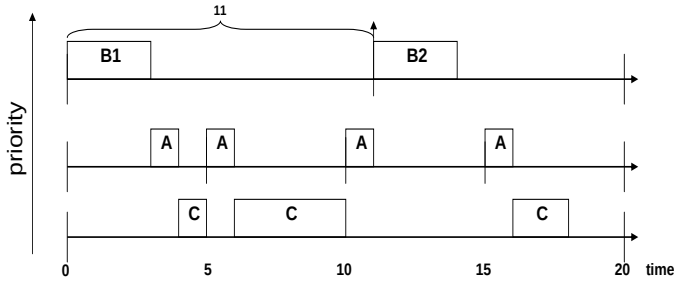


Figure 4.1: Motivating example - original task set

Let us now assume we want to be able to handle non-periodic events as well, together with our tasks. The processor utilization in the system is 75%. Thus, there are 5 time units of slack up to LCM (20), if the tasks execute for WCET. We can use, for example, a polling server with a period of 5 and a capacity 1 to distribute the capacity evenly over LCM. Now, if the server is not aware of the constraint on the instances of B, it could have the highest priority and, if there are any aperiodic jobs pending at the start of the server period, they will be served immediately while the periodic task will meet their deadlines. However, if an aperiodic task J arrives at time 0, it will be scheduled by the server and the constraint on B's instances will no longer hold even if all tasks meet

their deadlines (figure 4.2)

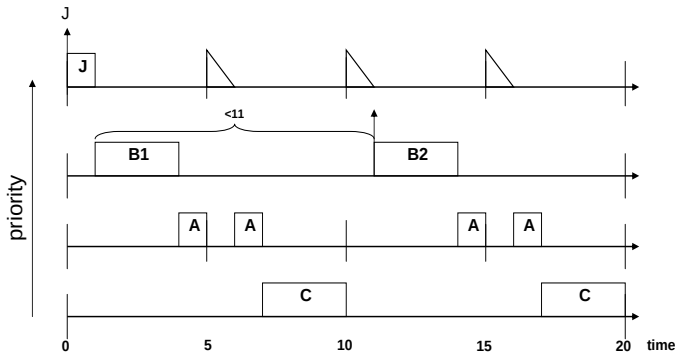


Figure 4.2: Motivating example - problem

In our example, the server was one that does not preserve its capacity. In this case, the server execution is interfering with the periodic tasks (upon an aperiodic request) for a period of time no longer than its capacity, starting at the beginning of its period, i.e., if no aperiodic requests are pending then, the server suspends itself until the beginning of its next period.

In the case that the chosen server is a capacity-preserving one, e.g., deferrable server, things could get even worse. That is because the server not only executes either at the beginning of its period or not at all, but anywhere within its period. In that case its potential interference intervall is drastically increased. The server execution will interfere with the periodic tasks for a period of time not longer than its capacity, but starting anywhere within its period

4.5 Proposed solution - overview

What we can conclude from the example above is that if the server is not aware of the constraints on the periodic tasks, i.e., the server parameter assignment is performed after the attribute assignment to the periodic

tasks, the constraints cannot be guaranteed (figure 4.3). However, if the server is aware of the original constraints, i.e., the system designer assigns server parameters by taking into consideration the constraint on the periodic tasks, the server execution can be controlled such that it would not interfere with the constrained periodic tasks.

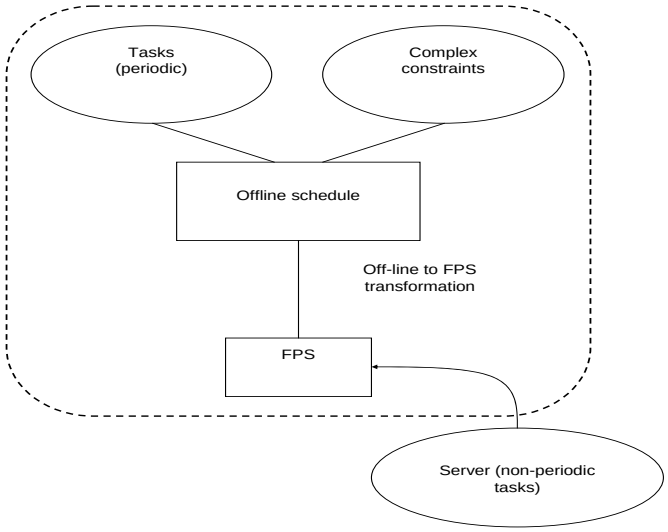


Figure 4.3: Server attribute assignment - **no** *constraint guarantees!*

To do so, we have to perform the attribute assignment on the periodic tasks and server(s) at the same time in the off-line phase, i.e., both tasks with constraints and the server(s) should be used as input to the method presented in chapter 2 (figure 4.4).

4.6 Server attribute assignment

To perform the server attribute assignment, by taking into account the available resources in the system, as well as the original constraints on the periodic tasks, we divided the servers in two groups: servers that

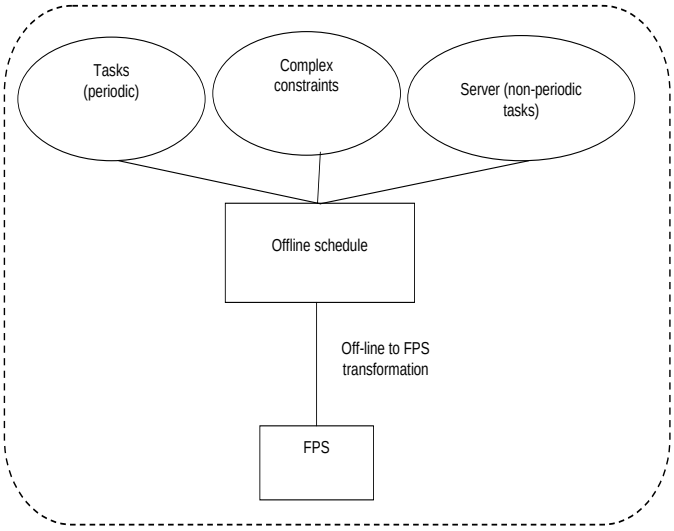


Figure 4.4: Server attribute assignment - *constraints guaranteed*

preserve their capacity during their period, and servers that don't.

4.6.1 Servers that do not preserve their capacity

In this case the server will become active at the beginning of its period, if any aperiodic request is pending, and will execute when it will have the highest priority among the tasks in the ready queue. In that sense, the server will behave exactly as a periodic task with a *worst case execution time* equal to its capacity and a *best case execution time equal* to 0, making the procedure of off-line schedule construction for the periodic tasks and server quite trivial. The important thing is to make sure that the server execution will not influence the execution of the constrained tasks.

Example - polling server

In the example described in section 4.4, an off-line schedule constructed to guarantee both server and periodic tasks with the separation constraint on the instances of task B could be the one presented in figure 4.5. In this example we use polling server, and we choose to assign it a period of 5 time units and a capacity of 1 to equally distribute the slack in the system over the LCM. However, the system designer could choose a hod will assign .

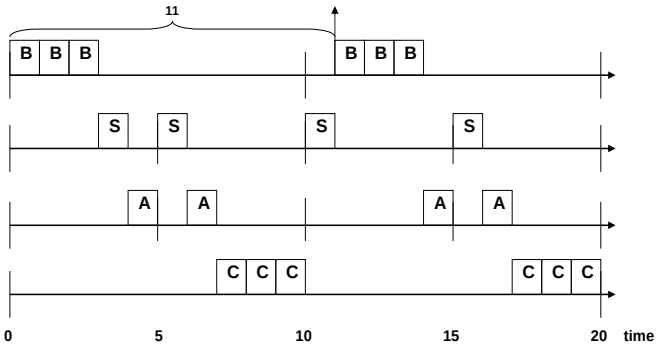


Figure 4.5: Non-capacity preserving server - off-line schedule

Then, by using the off-line schedule that includes the server as input to the method presented in chapter 2, we end up with FPS attributes for both periodic tasks and server as presented in table 4.3

Let's now assume that an aperiodic request *J* occurs at time 4 with an execution requirement of 3 time units. J be served starting at time 5, since it was not pending at the beginning of the server period, and will complete at time 16 while all periodic tasks meet their deadlines and the original constraint on B's instances holds. The run-time execution of the periodic tasks and the service provided by the server to the aperiodics is illustrated in figure 4.6.

The task attributes shown in table 4.3 illustrate the highest priority

$Task$	p	c	prio	offset	dl
A	5	1	2	0	5
B1	20	3	4	0	10
B2	20	3	4	11	20
C	20	6	1	0	20
S	5	1	3	0	5

Table 4.3: FPS attributes for constrained periodic tasks and a non-capacity preserving server

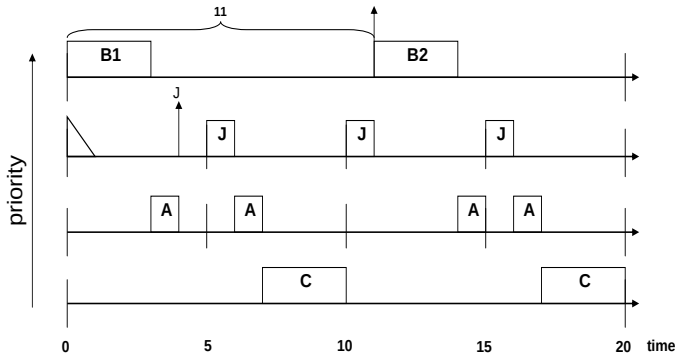


Figure 4.6: Polling server - FPS schedule

that the server can be assigned without jeopardizing the original complex constraints. Hence, we can feasibly schedule the periodic tasks and guarantee their constraints while providing the best possible service to non periodic events.

4.6.2 Capacity preserving servers

In this case, the server can start to serve an aperiodic request anywhere within its period. That makes it difficult to schedule it off-line, i.e., to construct an off-line schedule for both constrained periodic tasks and server while taking full advantage of the server flexibility to handle ape-

riodics. Instead, we can try to make sure that the execution scenarios for the server are limited, in order to guarantee the constraints on the periodic tasks, while still provide for flexibility.

Ideally, we would always like to have as high priority as possible on the server, but not on the expense of the periodic tasks and their constraints. Hence, we have to protect the constrained periodic tasks from possible server interference.

At the same time, the system designer must be aware of the behavior of the server as well, i.e., the mapping of complex constraints to FPS attributes must be performed by taking into account the possibility that the server may start executing at any point in time during its period. Hence, the time intervals between the release time and deadline of the tasks, i.e., the *target windows* must represent time windows in which each task, if it executes and completes, will fulfill its original constraints.

In out previous example, we have to assign the instances of B deadlines of 3 and 14 respectively (table 4.4).

$Task$	p	c	prio	offset	dl
A	5	1	2	0	5
B1	20	3	3	0	3
B2	20	3	3	11	14
C	20	6	1	0	20

Table 4.4: FPS attributes prepared for capacity preserving servers

If we recall how we identified the current and interfering instances in chapter 2, we derived the sequence corresponding to time t_k as shown in figure 4.7. However, at that point we were only concerned with deriving FPS attributes to fulfill the original constraints without taking into account non-periodic events.

Now, on the other hand, when we derive the priority inequalities between the task instances, we have to take into account the possibility that the execution of some instance of a periodic task can be postponed by the non-predictable execution of the server, and the execution of this particular instance can, in its turn, interfere with another one, at a later

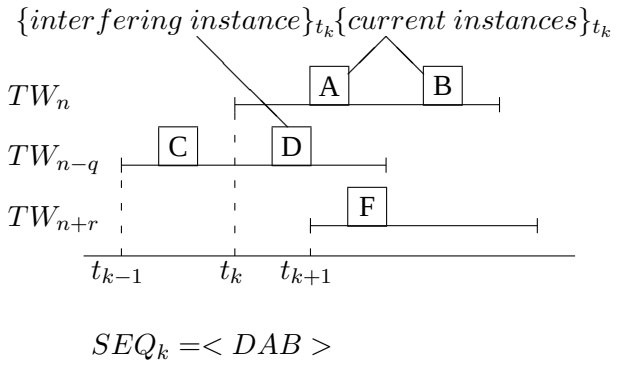


Figure 4.7: Old sequence of tasks.

point in time. In figure 4.7, the execution of C could be delayed by the execution of the server upon an aperiodic request. In that case, C will interfere with A and B, and, later on, possibly with F as well. Hence, C must be included in the sequence corresponding time t_k as an interfering instance (figure 4.8).

Once the new sequences are derived, it is straight forward to derive the priority inequalities as described in chapter 2.

At this point we are ready to add the server to our task set. In order to find the highest feasible priority for the server, we first identify the possible interference between the server and the periodic task instances. As the period and capacity of the server is decided as described in the previous section, we define the time interference between the k^{th} server instance, i.e., $Server^k$, and the j^{th} instance of task T_i, as shown in the formula 4.1

$$time\_interference(T_i^j, Server^k) = [t1, t2], \qquad (4.1)$$

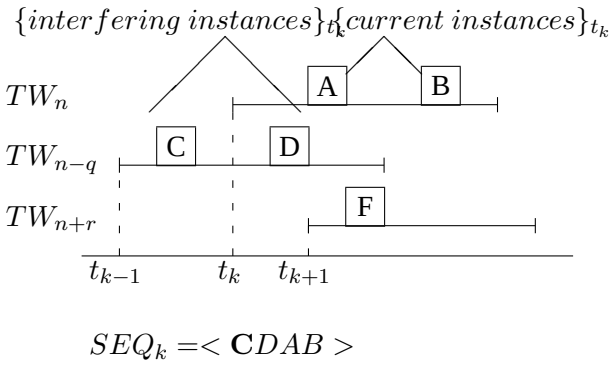


Figure 4.8: New sequence of tasks to deal with possible server execution.

where

$$t1 = max(Rel(T_i^j), Rel(Server^k))$$
$$t2 = min(dl(T_i^j), dl(Server^k))$$

We say that $time\_interference(T_i^j, Server^k) = 0$ if $t1 \geq t2$ and we denote the size of the interference intervall by

$$|time\_interference(T_i^j, Server^k)| = t2 - t1$$

However, the execution interference introduced by the server to any periodic task instance is never greater than the server capacity or the length of the interference intervall, in case the length of interference intervall is less than the server capacity. Thus, we define the execution interference introduced by the k^{th} server instance to the j^{th} instance of the task T_i, $exec\_interference(Server^k, T_i^j)$, as shown in formula 4.2.

$$
\begin{aligned}
if \quad & (time\_interference(Server^k, T_i^j)), \\
then \quad & exec\_interference(Server^k, T_i^j) = \\
& = min\{|interference(T_i^j, Server^k)|, C(Server\}
\end{aligned}
\tag{4.2}
$$

In a similar way, we can calculate the execution interference of higher priority task instances, $hp(T_i^j)$ to the execution of T_i^j (formula 4.3).

$$
\begin{aligned}
if \quad & (time\_interference(hp(T_i^j), T_i^j)), \\
then \quad & exec\_interference(hp(T_i^j), T_i^j) = \\
& = \sum_{T_m^n \in hp(T_i^j)} min\{|interference(T_m^n, T_i^j)|, C(T_m^n)\}
\end{aligned}
\tag{4.3}
$$

Note that at this point the priorities have not been assigned by ILP yet. However, we know which the interfering instances are by analyzing the sequences derived as shown in figure 4.8 and we know that the priority ordering within the same sequence will be assigned in a descending order. For example, in the sequence $SEQ_k =< ABC >$, A will get the highest priority followed by B while C will get the lowest priority. In this case, A and B will belong the set of tasks with higher priority than C, i.e., $hp(C)$.

What we have to check here is if it is possible to run the server at a higher priority than the periodic task instances it interferes with, while not causing the periodic instances to miss their deadline. At the same time, we have to take into account the interference from the other periodic tasks as well.

For every capacity preserving server, except slack stealing which we discuss later in this section, we will use the algorithm presented in equation 4.4.

At this point, we use ILP to solve the priority inequalities between the original periodic tasks by analyzing the sequences, together with the priority inequalities between the task instances and the server instances.

The ILP solver will provide us information about which priorities we should assign to the tasks and the server, such that the server priority is maximized while the original constraints on the periodic tasks fulfilled when the task set is scheduled by FPS.

$\forall T_i^j$**,**

$$
\begin{aligned}
& if \quad (interference(T_i^j, Server^k) \neq 0) \\
& \quad \{ \\
& \quad if(|exec\_interference(Server^k, T_i^j)| + \\
& \quad +|exec\_interference(hp(T_i^j), T_i^j)| + \\
& \quad +C(T_i^j) \leq dl(T_i^j)) \\
& \quad prio(Server^k) > prio(T_i^j) \\
& \quad else \\
& \quad prio(Server^k) < prio(T_i^j) \\
& \quad \} \\
& else \\
& \quad no\ interference \rightarrow no\ priority\ constraint \\
& \quad between\ Server^k\ and\ T_i^j.
\end{aligned}
\tag{4.4}
$$

In cases of attribute inconsistencies between different instances of the same task, or between different server instances, we create artifacts for the instances to achieve consistent FPS attributes, as described in chapter 2

Slack Stealing: in the case slack stealing is used as algorithm to handle aperiodics, we do not have to take its interference with the periodic tasks into consideration, as the slack stealer keeps track and uses the available slack in the system without causing any deadline misses among the periodic tasks. Since the task deadlines represent the latest point in time at which periodic task can complete their execution such that their complex constraints are fulfilled, slack stealing can be directly used to handle the non period events.

Example - deferrable server

Lets take a look at the example we shown in figure 4.6 where we used polling server and we had an aperiodic request at time 4, and lets see what happens if we use, e.g., deferrable server instead.

First, the off-line scheduler has to make sure that the task deadlines reflect the original constraint on the instances of B, i.e., the target windows of the instances of be will be [0,3] and [11,14]. Then, we assign the server a period of 5 and a capacity of 1, to allow for full exploitation of the available slack in the system and to equally distribute the capacity of the server over LCM.

Secondly, we derive the priority inequalities by analyzing the interference between the tasks and server as described in formulas 4.2, 4.3, 4.4 and we use ILP to solve the inequalities.

The task and server parameters are presented in table 4.5. Due to FP attribute inconsistency on B's instances, i.e., in this case release time inconsistency, we create artifacts for the former instances of B.

$Task$	p	c	prio	offset	dl
A	5	1	2	0	5
B1	20	3	4	0	3
B2	20	3	4	11	14
C	20	6	1	0	20
S	5	1	3	0	5

Table 4.5: FPS attributes for constrained periodic tasks and deferrable server

The run time schedule in the presence of an aperiodic request at time 4 is illustrated in figure 4.9. We can see that, by using DS, the completion time of the aperiodic task is now 11 in stead of 16 as in the case PS was used, while all task meet their deadlines. Hence, the original constraint on B's instances holds as well.

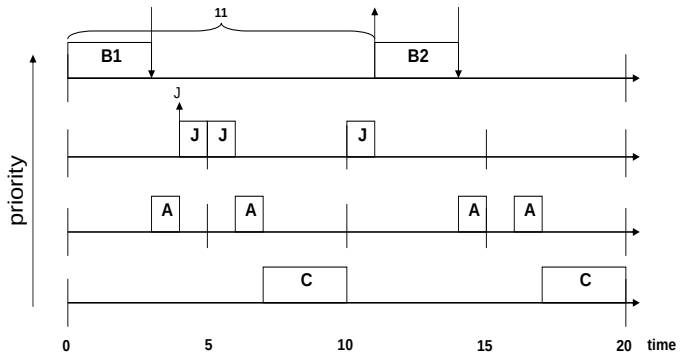


Figure 4.9: Deferrable serve - FPS schedule

4.7 Chapter summary

In this chapter, we investigated how existing FPS servers can be used together with FPS task sets obtained by using the method described in chapter 2, while guaranteeing the original complex constraints on the periodic tasks and providing a good service for non-periodic requests. We have shown that problems may arise if the servers are not aware of the complex constraints on the periodic tasks, e.g., if the server is added to the system after the transformation of the off-line schedule to task attributes for FPS. Instead, we proposed the inclusion of the server together with the rest of the periodic tasks and their constraints during the off-line schedule construction.

We have shown that servers that do not preserve their capacity during their periods, if no aperiodic requests are pending at the beginning of the period, can be easily incorporated in our method while guaranteeing the constraints of the periodic tasks, by treating them as periodic tasks that are ready to execute at the beginning of their periods, and do not suspend themselves. In this case, the only difference between the server and the periodic tasks is that the server may not execute at all if no aperiodic requests are pending at the beginning of its period.

Furthermore, we have provided a mechanism to be able to handle non-periodic events by existing servers that preserve their capacity during their periods until an aperiodic requirement occurs. The difficulty of scheduling this type of servers together with periodic, complex constrained tasks, is that the server execution, i.e., at which time the server starts its execution within its period, can not be predicted such that it can be included in the off-line schedule construction.

While some of these servers manage to schedule themselves without causing any deadline misses among the periodic tasks, e.g., slack stealer, others, e.g., deferrable server, are dependent on the set of FPS parameters assigned to them, to ensure the timeliness of periodic tasks. However, by modifying the method presented in chapter 2 we can find high priorities for these servers such that non-periodic events are promptly served while the constraints on the periodcs are still fulfilled.

The modifications to our original transformation method consist in, first, assign task deadlines to reflect the latest points in time at which periodic tasks must complete to fulfill their complex requirements. Secondly we include the interference from the tasks earlier off-line scheduled whose executions may be postponed by the server, into the derivation of the sequences and to derive the priority relations between tasks and server accordingly.

In our method we choose server periods and capacities to distribute the exploitations of the amount of slack in the system over LCM. The main advantage of equally distributing the server execution over LCM, i.e., short periods that imply an increased number of server instances up to LCM, is that we increase the chances to provide the aperiodics a prompt service by assigning the server a high priority. In some cases, we may have to create artifacts for the server instances to achieve a high priority setting. This is because the server can have temporary high priorities over short period of times, but not during the entire LCM. If, on the other hand, we choose to assign a server a period equal to LCM and a capacity up to full processor utilization together with the periodic tasks, we may end up in a situation where the server mast be assigned the lowest priority, since it interferes with all periodic task instances,

ending up with background scheduling.

While our method can feasibly schedule the existing servers together with the periodic FPS tasks created by the method described in chapter 2, any feasible attribute assignment can be used on the servers as long as the designer can guarantee the completion of the periodic tasks before their deadlines. That is because, in our off-line to FPS transformation, we specified the release times and deadline of the tasks, i.e., the target windows, in which if the tasks execute and complete, the original constraints are guaranteed.

Chapter 5

Controlling the number of preemptions in FPS

5.1 Introduction and problem description

Preemptive fixed priority scheduling (FPS) has been widely studied since the work of Liu and Layland [38]. It has gained large acceptance in a number of applications, mostly due to simple run-time scheduling and good flexibility for tasks with incompletely known attributes. However, the impact of preemption related overhead in FPS in the context of real-time systems, is well recognized [9, 44]. In multimedia applications, for example, tasks may introduce a high context switch cost [17]. In fact, preemption related overhead in FPS may cause undesired high processor utilization, high energy consumption, or, in some cases, even infeasibility. In [10], the author showed that the rate monotonic algorithm (RM) introduces a higher number of preemptions than earliest deadline first algorithm (EDF). At the same time, reducing the number of preemptions can also be beneficial from an energy point of view in systems with demands on low power consumption. When a task is preempted there is a great probability that its contents in the cache will be lost. When the execution of the task is again resumed it will cause a lot of energy consuming accesses to off-chip memory. An access to off-chip memory

is typically 10-100 times more expensive than an on-chip cache access in terms of energy consumption. Reducing the number of preemptions will reduce these additional expensive memory accesses due to reduced cache pollution.

The direct preemption cost, i.e., costs to perform context switches [24], to handle interrupts [24, 22, 9], or to manipulate task queues [9, 24], has been analyzed. Cache-related preemption, i.e., indirect cost, [31, 46], has been analyzed to incorporate it into schedulability analysis, as well as approaches to bound the cache-related preemption delay have been presented [32]. Approaches to reduce the number of preemptions in FPS have been presented [61, 25, 26], where tasks, besides their priorities, are assigned a threshold value such that they can be preempted only by other tasks with priorities higher than the threshold. This approach results, in essence, in a dual priority system which is not directly suitable for legacy systems, where exchanging the scheduler or modifying it by, e.g., adding mutexes to simulate preemption threshold, is not desirable, or not possible.

In this chapter, we introduce a method to reduce the number of preemptions in FPS systems in which modifications to the original scheduler are not desirable or not even possible. The method can be directly applied on existing FPS systems with high preemption costs as no additional modifications to the underlying scheduler are required. In particular, we provide users of FPS systems the ability to choose a user defined number of preemptions with respect to an eventual cost to pay.

Our method analyzes a set of periodic tasks with periods, priorities and offsets, scheduled by an FPS algorithm, in order to identify the number of preemptions that can occur at run-time. The basic idea is to reassign attributes to the tasks such that the tasks will execute feasibly at run-time while achieving a lower number of preemptions. To resolve a preemption between two task instances, we either swap the priorities or we force the task instances to be released simultaneously (e.g., by reassigning offsets).

Since reassigning attributes to a particular task instance may lead

to inconsistent attributes for the instances of the same task, we create artifact tasks for the task instances to solve the inconsistency. In a recent chapter 2 we transformed off-line schedules into FPS schemes. In [3], the authors derived response time bounds for tasks with offsets information and introduced an optimal priority ordering algorithm. In our approach, however, we strictly focus on reducing the number of preemptions in existing FPS systems consisting of tasks with periods, priorities and offsets, without modifying the underlying scheduler.

By choosing to eliminate preemptions one by one, we may lose optimality, since, depending on which preemption is chosen to be eliminated first, the outcome of the method may differ. In this work we use a global approach to detect preemption dependencies and to selectively choose a user-defined level of preemptions with respect to the trade-off between the number of preemptions and the cost to pay, i.e., the number of new artifacts eventually created and the level of decreased flexibility. To do so, we construct a preemption dependency graph that comprises all possible steps we can perform to eliminate preemptions, and all corresponding states representing the new number of preemptions achieved by the new task attributes, and the cost to pay.

Motivating example As a short example, let us assume we have two tasks, A and B, with periods and WCET (3,1) and (5,3) respectively, scheduled by a preemptive fixed priority scheduling algorithm. We assume that the deadlines are equal to the end of the periods and the tasks execute for WCET. Task A has a higher priority then task B. In figure 5.1(a) we illustrate a scenario where A preempts B at time t=3 and causes B to miss its deadline at time t=5, due to the context switch. This, however, would be avoided if the second instance of A would have a lower priority then B (figure 5.1(b)). On the other hand, the first instance of A has to have a higher priority then B, otherwise it would miss its deadline. In our method we solve this dilemma by transforming task A's instances into new periodic tasks with consistent FPS attributes. The new tasks will be assigned periods equals to the least common multiple of the tasks periods, LCM, and offsets and priorities to ensure the

execution and completion between the original release times (rel) and deadlines (dl).

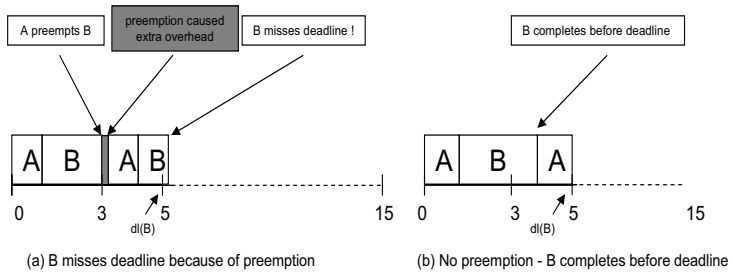


Figure 5.1: A simple example

The rest of the chapter is organized as follows: in section 5.3 we give an overview of the proposed method. Section 5.4 describes the basic approach to solve a single preemption. The algorithm proposed to reduce the number of preemptions is described in section 5.5 followed by a simple example (section 5.6) and performance evaluation (section 5.7). Section 5.8 concludes the paper.

5.2 Problem formulation and task model

Problem formulation Given a set of periodic tasks with periods, worst case execution times (WCET), offsets, and priorities, schedulable by a standard FPS algorithm, we want to provide new sets of feasible attributes such that the tasks will achieve a lower number of worst case preemption scenarios if scheduled by the same scheduling algorithm. In particular, we want to provide for the ability to chose a user-defined number of preemptions with respect to the cost to pay.

Task model In this approach, we assume that the tasks are periodic, with attributes suitable for fixed priority scheduling and schedulable by

a preemptive fixed priority scheduling algorithm. We use the same task model as described in the previous chapters:

- we denote tasks as $T_i,\ i \in \{1, 2, \ldots\}$
- T_i^j is the j^{th} instance of T_i, $j \in \{1, 2, \ldots, \frac{LCM}{p(T_i)}\}$
- $p(T_i)$ is the period of T_i
- $prio(T_i)$ and $offset(T_i)$ are the priority and offset of T_i
- $dl(T_i)$ is the deadline of T_i
- $c(T_i)$ is the worst case execution time (wcet) of T_i
- $rel(T_i^j)$ is the release time of T_i^j
- we define $start(T_i^j)$ and $finish(T_i^j)$ as the actual start and finishing time of T_i^j, derived in the off-line analysis, assuming that all tasks execute for wcet.

Furthermore, we assume that the deadline of each task is less than or equal to its period and tasks do not suspend themselves.

5.3 Method overview

Here we provide an off-line method to analyze the preemptions which can occur when a set of tasks is scheduled by FPS, and to reduce the number of preemptions by reassigning attributes to the tasks.

For a given set of periodic tasks scheduled by FPS, we first perform an off-line preemption analysis assuming the task executions for wcet. The preemption analysis takes into account even *potential preemptions* that may occur if tasks execute for less than wcet. By that, we detect the points in time at which a preemption can occur.

For each preemption we have a *preempting task instance* and a *preempted task instance*. Depending on the processor utilization during the time interval in which the preempting/preempted task instances can

feasibly execute, we attempt to eliminate the preemption by forcing the execution of either the preempted or the preempting instance such that they do not preempt each other while still executing feasibly.

In our method, we reassign attributes, i.e., priorities or offsets, to the individual instances. For example, assume that the i^{th} instance of a high priority task A, A^i, is preempting the j^{th} instance of a low priority task B, B^j, at time t. In this case, we can either reassign A^i a lower priority than B^j, or we can attempt to prevent B^j from executing before time t, i.e., to reassign B^j an offset such that B^j will be released at time t. At the same time, we have to make sure that an attribute reassignment, i.e., new priority or new offset, will not prevent any task from completing before its original deadline. If both options will cause any deadline miss, we can not solve the preemption. We perform a standard response time analysis to investigate the feasibility of the tasks after reassigning attributes, [58, 34].

When we reassign priorities, i.e., when we swap the priorities of two task instances which preempt each other, we consider the priority relations of the rest of the tasks as well. We do so by solving the new priority relation together with the old ones by using integer linear programming (ILP). Particular task priorities can be selected to not be changed (with respect to the other task priorities) by simply reformulating the ILP representation. In the same way, particular task offsets can be specified to remain unchanged. However, that will reduce the possibilities of eliminating preemptions.

5.3.1 Preemption reduction cost

Since reassigning attributes to a particular task instance may lead to inconsistent attributes for the instances of the same task, we create artifact tasks for the task instances to solve the inconsistency. The number of artifacts created to achieve a particular number of preemptions is one of the costs we have to pay. However, by using ILP, we ensure that the number of eventually created artifacts is minimal.

Another cost is given by the reduced flexibility in the case we reassign offsets and decrease the time interval in which an instance may

execute feasibly. Here, we use the term introduced in chapter 2, i.e., *target window*, to measure the loss of flexibility. In particular, We use the number of decreased target windows as a measurement unit for the decreased flexibility introduced by our method.

As prevoiusly described, the target window of a task instance is the time interval in which the instance has to execute and complete in order to execute feasibly, e.g., the target window of a particular instance T_i^j of a task T_i, is the time interval between its release time, i.e., offset or start of the period, $rel(T_i^j)$, and its deadline, $dl(T_i^j)$:

$$TW(T_i^j) = [rel(T_i^j), dl(T_i^j)]$$

Since resolving a preemption by reassigning attributes may solve or introduce another one, as the execution pattern of several tasks may change, we construct a *preemption dependency tree* to detect all feasible task sets schedulable by the original fixed priority mechanism, which will execute achieving a lower number of preemptions. By keeping track of the preemption reduction cost, we provide for the ability to chose a user-defined number of preemptions with respect to the cost to pay.

5.4 Solving a single preemption

In this section we describe our basic approach to solve one particular preemption.

Preemptions: In our off-line preemption analysis we assume that tasks execute for wcet. However, at run-time, tasks will most likely execute for less than wcet, implying a different number of preemptions compared to the ones detected by our off-line method. Our goal is to detect all possible preemptions that may occur at run-time. Hence, we divide the preemptions we attempt to solve in two major categories:

Initial preemptions - are detected in the off-line analysis, i.e., a high priority task instance is *initially preempting* a low priority task instance (figure 5.2).

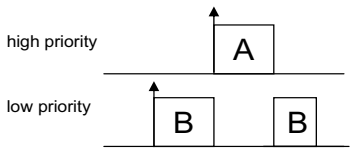


Figure 5.2: An off-line detected initial preemption

We say that a high priority task instance T_i^m is *initially preempting* a low priority task instance T_j^n if:

$$\begin{aligned} rel(T_j^n) &< rel(T_i^m)\ and \\ start(T_j^n) &< start(T_i^m)\ and \\ finish(T_j^n) &> rel(T_i^m) \end{aligned}$$

Potential preemptions - can occur at run-time due to task executions less than wcet. In figure 5.3 a) we can see that if tasks execute for WCET, no preemption will occur. However, in this situation we consider task A *potentially* preempting task B since, if task C, that delays the execution of B, is executing for less then wcet, then B can start executing earlier, i.e., before the release time of A, and will actually be preempted by A (figure 5.3 b).

We say that a high priority task instance T_i^m is *potentially preempting* a low priority task instance T_j^n if:

$$rel(T_j^n) < rel(T_i^m)\ and\ start(T_j^n) > finish(T_i^m)$$

Consequently, in order to cover both types of preemptions, we define the preemptions we attempt to solve as following:

T_i^m is preempting T_j^n if the following three conditions hold:

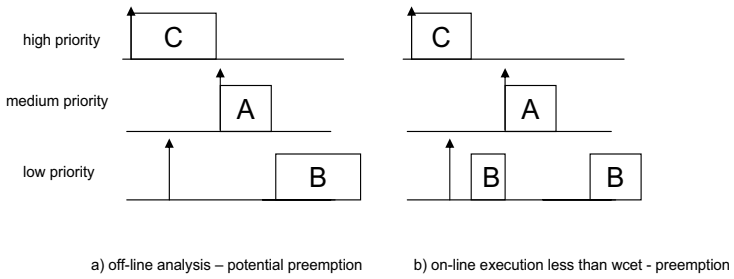


Figure 5.3: An off-line detected potential preemption

$$prio(T_i^m) > prio(T_j^n) \quad (5.1)$$
$$rel(T_i^m) > rel(T_j^n) \quad (5.2)$$
$$finish(T_j^n) > rel(T_i^m) \quad (5.3)$$

Hence, it is sufficient to eliminate any of the conditions to ensure that the preemption is avoided. However, the preemption reduction cost may vary depending on which condition we decide to eliminate.

5.4.1 Solving a preemption by eliminating the first condition

The basic idea is to swap the priorities of the preempting and preempted instances while taking into consideration the priority relations between all task instances as well. We do so by, first, breaking down the original priority relations to the instance level in form of priority inequalities, and then, we solve the inequalities by using *integer linear programming* (ILP).

Preparing step: Before attempting to solve any preemption, we break down the original priority relations, $prio(T_i) > prio(T_j)$, to the instance level. The reason for expressing the original priority relations

between the tasks by priority relations between task instances is that our method attempts to solve preemptions by reassigning attributes to individual task instances by using ILP.

We first introduce the concept of *interference* between the execution of the task instances. The condition of interference in (5.5) is used to prevent priority relations between task instances that do not interfere with each other from over constraining the inequality system, as using ILP to solve an over-constrained inequality system, will result in an sub-optimal solution.

For instance, if there is a priority relation between two tasks A and B, e.g., $prio(A) > prio(B)$, but no actual interference between, e.g., the fifth instance of A and the first instance of B, then there is no need to add the priority inequality $prio(A^5) > prio(B^1)$ to our representation of the priority relations.

Hence, we express the original priority relations among the tasks as following:

$$\begin{array}{rcll} \forall i,j,m,n,\ if & \mathbf{interference}(T_i^m, T_j^n) & == 1, & (5.4) \\ replace & prio(T_i)\ \mathbf{op}\ prio(T_j) & by & \\ & prio(T_i^m)\ \mathbf{op}\ prio(T_j^n) & & \end{array}$$

where

$$\mathbf{op} = \begin{cases} >,\ if\ prio(T_i) > prio(T_j) \\ <,\ if\ prio(T_i) < prio(T_j) \\ =,\ otherwise \end{cases}$$

The interference between two task instances, T_i^m and T_j^n is defined in (5.5).

$$\mathbf{interference}(T_i^m, T_j^n) = \tag{5.5}$$
$$\begin{cases} 1, \; if \; \begin{cases} rel(T_i^m) \leq rel(T_j^n) \; and \; dl(T_i^m) > rel(T_j^n) \\ or \\ rel(T_i^m) \geq rel(T_j^n) \; and \; dl(T_j^n) > rel(T_i^m) \end{cases} \\ 0, \; otherwise \end{cases}$$

ILP formulation Once the priority inequalities between the task instances are derived, we simply search the inequality system for the priority relation between the preempted and preempting task instances, and reverse the inequality. Then, we solve the new inequality system by using integer linear programming (ILP) in order to find a feasible priority assignment. By using ILP, we make sure that the rest of the priority relations between the task instances will not change, even though the individual priority values may change.

Let's assume, for example, that the i^{th} instance of task T_m, T_m^i, is preempting the j^{th} instance of T_n, T_n^j, i.e., the two task instances are interfering with eachother and the priority of T_m is higher than the priority of T_n. Then, we search the inequality system for the inequality $prio(T_m^i) > prio(T_n^j)$ and we replace it by $prio(T_m^i) < prio(T_n^j)$.

The next step is to solve the new inequality system, followed by performing a response time analysis to verify the feasibility of the task instances with the new attributes.

The potential cost introduced in this step is an increased number of tasks as a result of priority inconsistencies, i.e., we may have to create new tasks for the instances with eventual inconsistent priorities. However, the goal function of the ILP solver is formulated to find a solution to the new priority inequalities and to minimize (if any) the number of tasks instances with inconsistent attributes [16]. If the inequality solver yields no solution or, if the schedulability test finds the task unschedulable due to the new attributes, then we can not solve this particular preemption by reassigning priorities.

5.4.2 Solving a preemption eliminating the second condition

The next alternative we have is to force the task instances to be released simultaneously. That is, to reassign T_j^n a release time (offset) equal to $rel(T_i^m)$. From the definition of the preemptions we aim to solve, (5.2), we know that $rel(T_j^n) < rel(T_i^m)$, so, in essence, we decrease the time interval in which the instance can feasibly execute, i.e., its target window, from $[rel(T_j^n), dl(T_j^n))$ to $[rel(T_i^m), dl(T_j^n))$. Then, we perform the schedulability analysis to verify the schedulability of all task.

However, by reassigning offsets, i.e., by postponing the release time of a task instances, we decrease the flexibility of that particular instance. Thus, the cost introduced in this step is, besides eventual new tasks resulted from offset inconsistencies, a decreased flexibility for particular tasks. If the task set is found unschedulable, we can not solve this particular preemption in this step.

5.4.3 Solving a preemption eliminating the third condition

The last possibility we have to solve a particular preemption is to reassign attributes such that the third condition no longer holds.

Proposition 1 If a task instance T_j^n is off-line detected to be preempted by at least one task instance T_i^m, we define a *preemption block* of T_j^n, denoted as $p\_block(T_j^n)$, as the time interval between $start(T_j^n)$ and $finish(T_j^n)$. The length of $p\_block(T_j^n)$ is constant regardless from the execution order of the task instances in the block, assuming task executions for wcet.

Hence, a sufficient condition to ensure that the third condition no longer holds is to assign T_i^m an earliest start time equal to $finish(T_j^n) - c(T_i^m)$, i.e., to force the preempted instance to execute at the end of the preemption block.

$$rel(T_i^m) = finish(T_j^n) - c(T_i^m)$$

Proof If T_i^m is forced to execute at the end of the preemption block of T_j^n, it will leave an idle time between $start(T_j^n)$ and $(finish(T_j^n) - c(T_i^m))$ equal to its computation time, $c(T_i^m)$. That time will be used by any of the instances all ready executing in $p\_block(T_j^n)$, so the preemption block will end $c(T_i^m)$ time units earlier. Hence, T_j^n will finish before T_i^m is released and the preemption will be eliminated. □

As in the previous step (5.4.2), the cost introduced is given by both increased number of tasks as well as decreased flexibility.

However, solving another preemption may turn our 'unsolvable' preemption solvable, or may even solve it automatically as the execution behavior of several tasks changes. In section 5.5 we show how we take advantage of this phenomena by recursively investigating all possible effects caused by solving a preemption.

The pseudocode for solving one particular preemption by reassigning priorities or offsets, is shown in figure 5.4. The input consist of the original task attributes, i.e., priorities, offsets, periods and deadlines.

$R(T_i)$ in figure 5.4 represents the worst case response time of task T_i, obtained by performing a response time analysis on the set of tasks.

5.4.4 Artifact tasks

By reassigning attributes to eliminate preemptions, we may end up with inconsistent priorities or offsets for different instances of the same task. To solve this side-effect of our proposed preemption reduction method, we split the task with the inconsistent attributes into a number of new periodic, fixed priority tasks and, by that, we create a number of *artifact tasks*, each of them with FPS attributes. The number of new created artifact tasks is the cost we have to pay to reduce the number of preemptions. The instances of the new tasks comprise all instances of the original task. If at least one of the instances of a task T_i has been reassigned new attributes, i.e., priorities or offsets, and, by that, creating inconsistent task attributes, we transform the instances T_i^j, $j \in [1, \frac{LCM}{period(T_i)}]$, in new tasks τ_{ij} as described in equation 5.6.

```
solve preemption(A is preempting B ) by eliminating the first condition
{
get task_attributes;
inequalities=create_priority_inequalities;                    //add the new priority inequality to the system
replace( 'prio(A)>prio(B)' by 'prio(A)<prio(B)' in 'inequalities' );
if( LP_solve(inequalitites) )                                 //if ILP yields a solution to the inequalities
{
        if( R(T_i ) ≤ dl(T_i ),  ∀ i )                        //perform the feasibility test for all tasks T_i
        {
                create artifacts(task_attributes);            //artifacts for tasks with inconsistent attributes (if any)
                save task_attributes;                         //save the new task attributes
                exit(success);                                //preemption solved
        }
}
exit(failure);                                                //preemption unsolvable
}
solve preemption(A is preempting B ) by eliminating the second condition
{
get task_attributes;
update ( 'rel(B):=rel(A)');                          //update offset of B
        if( R(T_i ) ≤ dl(T_i ),  ∀ i )                        //perform the feasibility test for all tasks
        {
                create artifacts(task_attributes);            //artifacts for tasks with inconsistent attributes (if any)
                save task_attributes;                         //save the new task attributes
                exit(success);                                //preemption solved
        }
exit(failure);                                                //preemption not solvable
}
solve preemption(A is preempting B ) by eliminating the third condition
{
get task_attributes;
update( 'rel(A)=dl(B)- c(A )');                               //update offset of A
        if( R(T_i ) ≤ dl(T_i ),  ∀ i )                        //perform the feasibility test for all tasks
        {
                create artifacts(task_attributes);            //artifacts for tasks with inconsistent attributes (if any)
                save task_attributes;                         //save the new task attributes
                exit(success);                                //preemption solved
        }
exit(failure);                                                //preemption not solvable
}
```

Figure 5.4: Pseudocode for the solving one preemption

$$
\begin{aligned}
period(\tau_{ij}) &= LCM \\
priority(\tau_{ij}) &= \begin{cases} new\ priority,\ if\ reassigned \\ unchanged,\ otherwise \end{cases} \\
offset(\tau_{ij}) &= \begin{cases} new\ offset,\ if\ reassigned \\ (j-1) * period(T_i),\ otherwise \end{cases} \\
deadline(\tau_{ij}) &= unchanged
\end{aligned}
\tag{5.6}
$$

From the implementation point of view, when we create artifacts for the instances of a task T_i, we create copies of the task control block of T_i pointing at the same code and data segment, but with different FPS

attributes. Hence, the modified tasks can be directly scheduled without modifications to the original scheduler.

5.5 Reducing the number of preemptions - global approach

Our goal is to reduce the number of preemptions in FPS without modifying its basic mechanism. As described in section 5.4, we attempt to eliminate (solve) a preemption by task attribute reassignment, i.e., priority or offset reassignment, while guaranteeing the feasibility of the task set.

However, if we use a constructive approach, i.e., if we attempt to eliminate the preemptions in a particular order, we may lose optimality in terms of finding the minimum number of preemptions. Depending on which particular preemption we choose to resolve first, the results, in terms of the number of preemptions achieved and/or the number of artifacts created, may differ. That is due to the fact that resolving a preemption by reassigning task attributes may resolve or introduce additional ones, as the execution pattern of several tasks may change.

Therefore, for each feasible task set, we choose to solve all preemptions by all approaches, i.e., we attempt to eliminate each preemption yielded by a particular task set by all three approaches (5.4.1, 5.4.2 and 5.4.3).

5.5.1 Preemption dependency tree

To find the optimum desired level of preemptions with respect to the artifacts to be created, we construct a *preemption dependency tree* that comprises all possible steps we can perform to eliminate preemptions.

Description of the tree The root of the tree consists of the original task attributes together with the corresponding off-line detected preemptions. Each preemption point is an edge from a node, i.e., solving that particular preemption will give us another node, with another task set

and new preemption points. Thus, by eliminating each of the preemptions, we obtain one, two or three new nodes, i.e., one for each successfully eliminated condition. If all approaches yield feasible task sets, we obtain 3 new nodes.

At the same time, we keep track of the cost we have to pay for each preemption we succeed to solve, i.e., the number of artifacts and the number of reduced target windows. Thus, each new created node in the graph will contain:

- The new feasible task attributes and the new preemptions yielded by the tasks in this configuration.
- The cost to pay, i.e., the number of artifacts and the number of reduced target windows.

Since one of the advantages obtained by using FPS is flexibility, the more appropriate way to solve a preemption is to reassign priorities. That is due to the fact that solving a preemption by reassigning offsets implies reducing the target windows of the task instances. However, our goal is to find the set of task attributes that yields the minimum number of preemptions while guaranteeing feasibility. Therefore, we attempt to solve each preemption by eliminating each of the three conditions in order to find all possible preemption sets yielded by corresponding task attributes.

The dependency tree is constructed by recursively using the 3 approaches described in sections 5.4.1, 5.4.2, and 5.4.3, followed by a new preemption analysis each time a preemption is resolved. The feasible task sets contained in the nodes of the tree are schedulable by the same original fixed priority mechanism.

A user-defined state can be selected by performing a basic tree-search, with respect to the trade-off between the number of preemptions, the number of artifacts and the number of reduced target windows.

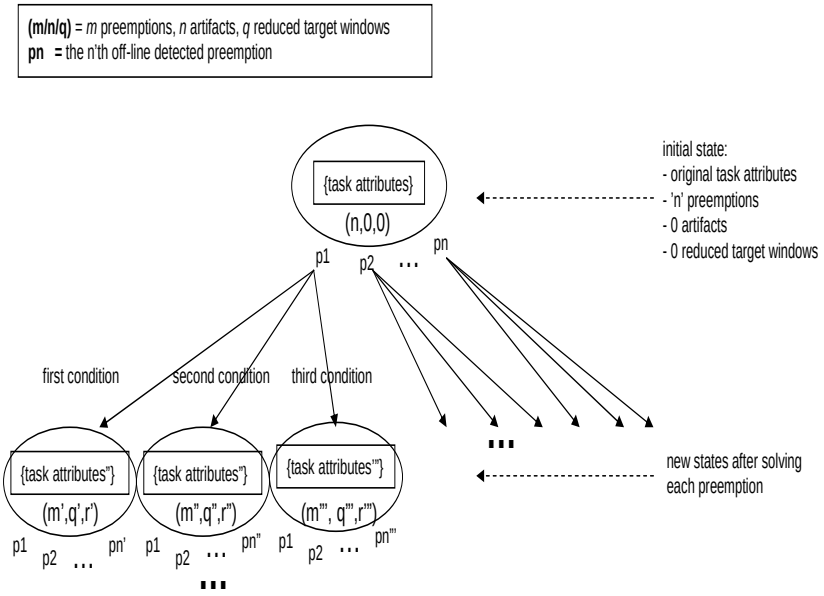


Figure 5.5: Preemption dependency tree

5.6 A simple example

We illustrate the proposed preemption reduction method with an example. We assume the set of FPS tasks described in table 5.1.

The task attributes are period (p), worst case execution time (c) and priority (prio). For an easier and more intuitive reading, we assume that the offsets are equal to the start of the period and a higher value represents a higher priority. In this example we assume that the worst case context switch time has been taken into account when calculating the worst case execution times for each task.

From the FPS schedule shown in figure 5.6, we can see that four preemptions can occur at run-time, if the tasks execute for wcet, when scheduled by FPS. Hence, the root of the preemption dependency tree

$Task$	p	c	prio
A	5	1	3
B	10	3	2
C	20	8	1

Table 5.1: Original FPS tasks

consist of 3 tasks, A, B and C with the original attributes, 4 preemptions, 0 artifacts, and 0 modified target windows.

The first preemption is detected at time t=5 when A^2 is preempting C^1. We first attempt to solve the preemption by swapping the priorities of A^2 and C^1. To do so, we first derive the target windows of the task instances (table 5.2) and, then, we break down the original priority relations between the tasks to the instance level. By analyzing the overlapings between the target windows (i.e., the interference between the task instances), we then derive the system of priority inequalities between the individual task instances according to (5.4). In our example, the original relations $prio(A) > prio(B) > prio(C)$ are transformed as following:

$$\begin{aligned}
prio(A) > prio(B) \quad &\rightarrow \quad prio(A^1) > prio(B^1), \\
&\qquad\quad prio(A^3) > prio(B^2) \\
prio(A) > prio(C) \quad &\rightarrow \quad prio(A^1) > prio(C^1), \\
&\qquad\quad prio(A^2) > prio(C^1), \\
&\qquad\quad prio(A^3) > prio(C^1), \\
&\qquad\quad prio(A^4) > prio(C^1) \\
prio(B) > prio(C) \quad &\rightarrow \quad prio(B^1) > prio(C^1), \\
&\qquad\quad prio(B^2) > prio(C^1)
\end{aligned} \tag{5.7}$$

Note that the priority inequalities $prio(A^2) > prio(B^1)$, $prio(A^2) > prio(B^2)$, and $prio(A^4) > prio(B^2)$ are not included in the inequality system, since there is no interference (as defined in (5.5)) between the mentioned instances. Hence, we do not have to take into account the

priority relation between them.

		task		
		A	B	C
$instance\ nr.$	1	[0,5]	[0,10]	[0,20]
	2	[5,10]	[10,20]	
	3	[10,15]		
	4	[15,20]		

Table 5.2: Target windows for the original task instances

At this point, we just replace the derived priority inequality between A^2 and C^1, i.e., $prio(A^2) > prio(C^1)$, by $prio(A^2) < prio(C^1)$. By solving the new inequality system using ILP, we obtain new task set consisting of 6 tasks (3 artifacts) with the following priorities: prio(A1)=5, prio(A2)=1, prio(A3)=4, prio(A4)=3, prio(B)=3, prio(C)=2. However, a schedulability test finds the new task set infeasible since the second instance of A will miss its deadline.

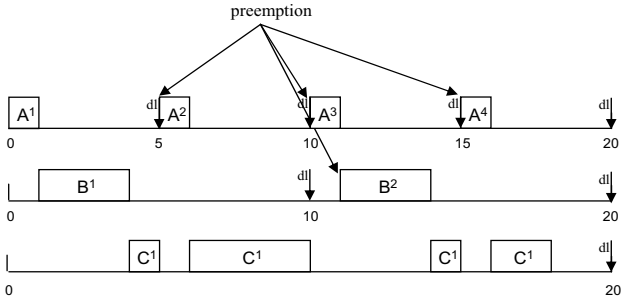


Figure 5.6: Original FPS schedule: task C is preempted by A and B

The second option we have is to reassign C^1 a release time equal to 5, which is the release time of A^2. This time, the task set is found schedulable. The new node we add to the tree consists of task attributes

for 3 tasks, i.e., *{task attributes'}* (Figure 5.7). The only difference from the original attributes in figure 5.1 is that C has now an offset equal to 5. The new number of preemptions (detected by performing the off-line preemption analysis) has decreased to 3, we have not introduced artifacts since C has only one instance in LCM and we have reduced one target window ($TW(C^1)$).

The third alternative to solving the preemption is to assign A^2 an offset equal to $finish(C^1) - wcet(A^2) = 17$. However, we can easily see that the new task set will not be feasible since A^2 has a deadline equal to 10. Hence, no new node will be added to the tree.

At this point we update (as described in (5.4)) the priority inequalities based on the modified target windows (in our case the only modified target window is $TW(C^1) = [5, 20]$).

$$\begin{aligned}
prio(A) > prio(B) \quad &\rightarrow \quad prio(A^1) > prio(B^1) \\
&\qquad\quad prio(A^3) > prio(B^2) \\
prio(A) > prio(C) \quad &\rightarrow \quad prio(A^2) > prio(C^1) \\
&\qquad\quad prio(A^3) > prio(C^1) \\
&\qquad\quad prio(A^4) > prio(C^1) \\
prio(B) > prio(C) \quad &\rightarrow \quad prio(B^1) > prio(C^1) \\
&\qquad\quad prio(B^2) > prio(C^1)
\end{aligned} \tag{5.8}$$

We can see now that the inequality $prio(A^1) > prio(C^1)$ has been removed in (5.8) since C^1 is no longer interfering with A^1.

In the same way we derive the rest of the nodes in the preemption dependency graph, by attempting to solve each preemptions by all three approaches. Due to space limitations, we show only a feasible path in the preemption dependency tree in figure 5.7. However, when running the example in our simulator, the complete tree consisted of 36 nodes while the solution presented in this section was found after building 8 nodes.

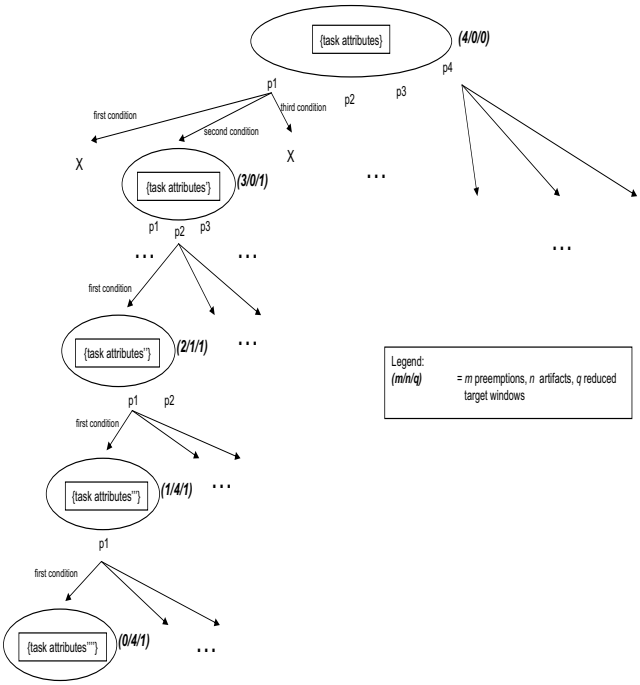


Figure 5.7: Example: preemption dependency tree

Once the tree is constructed, the user can easily traverse it to find the optimum solution. If the choice is to minimize the number of preemptions, one can follow the path shown in figure 5.7. The final node is obtained after successively eliminating 4 preemptions: A^2 preempts C^1 (the case we have described above), B^2 preempts C^1, A^3 preempts C^1 and, finally, A^4 preempts B^2. The minimum amount of preemptions, i.e., 0, is yielded by the task attributes shown in table 5.3, and the cost to pay is 4 artifacts and one reduced target window.

However, the user can choose an intermediate node if the trade-off between the number of preemptions and the number of artifacts is more suitable, e.g., 2 preemptions vs. 1 artifact and 1 reduced target window.

The final set of tasks with new attributes and the FPS schedule are shown in table 5.3 and figure 5.8 respectively. In each step, the artifact tasks are created as described in (5.4.4).

$Task$	p	c	offset	prio
A1	20	1	0	2
A2	20	1	5	5
A3	20	1	10	3
A4	20	1	15	1
B1	20	3	0	1
B2	20	3	10	2
C	20	8	5	4

Table 5.3: The new FPS attributes that yield no preemptions

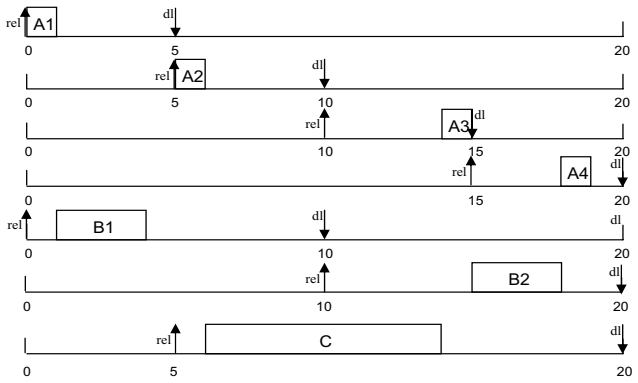


Figure 5.8: New FPS schedule, zero preemptions

5.7 Performance evaluation

We have performed a number of experiments to evaluate the efficiency of our method. We have used synthetic tasks with randomly generated

attributes, schedulable by FPS.

The experiments were run on each task set until either the tree was complete, a node containing attributes yielding 0 preemptions was found, or for a maximum duration of 2 minutes. The hardware used for the experiments consisted of a P4 PC at 1.9MHz.

Each point in the graph was created by performing computations on 50 task sets consisting of 5 to 10 tasks respectively. The LCM for each task set was randomized between 10 to 20 times the number of tasks. The periods were randomized in the interval $[\frac{LCM}{nr.\ of\ tasks}, LCM]$, and the wcet's were chosen to ensure an utilization of at least 0.6. In our experiments, the average utilization was 0.89. The priorities have been assigned according to the RM algorithm and original offsets were set to zero due to the optimality of RM priority assignment for this particualar setup. However, our method does not depend on the presence of offsets or priority assigment algorithm as it relies on release and start times.

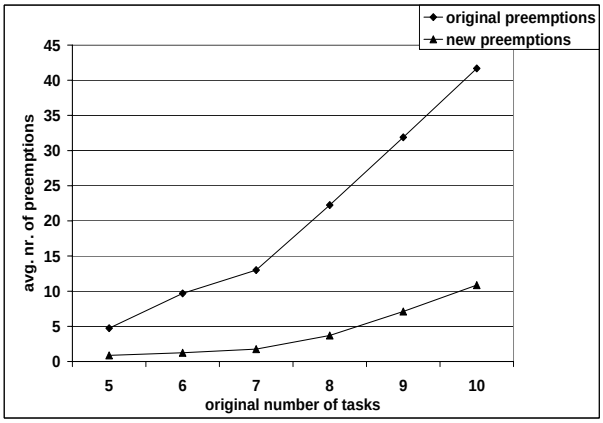


Figure 5.9: Average preemption reduction

When searching the tree for the best solution, we chose the node containing the task attributes with that yielded the lowest number of preemptions. If more than one node contained the minimum number of

preemptions, the chosen node was the one containing the least number of artifacts.

As we can see in figure 5.9, the method significantly managed to reduce the worst case number of preemptions for an average cost presented in figure 5.10.

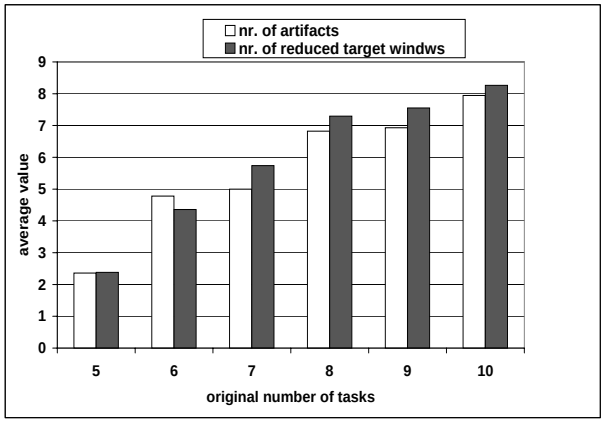


Figure 5.10: Preemption reduction cost

The average increase in the number of tasks introduced by our method is presented in figure 5.11.

5.8 Chapter summary

In this chapter, we proposed a method to reduce the number of preemptions in FPS without modifying its basic mechanism. Hence, the method is directly applicable to FPS systems in which modifications to the original scheduler are not desirable or not even possible.

Assuming the WCET for the tasks, the method analyzes off-line a set of periodic tasks with fixed priorities and offsets, scheduled by FPS, and identifies the number of preemptions that can occur at run-time. Our method reduces the number of preemptions by reassigning task at-

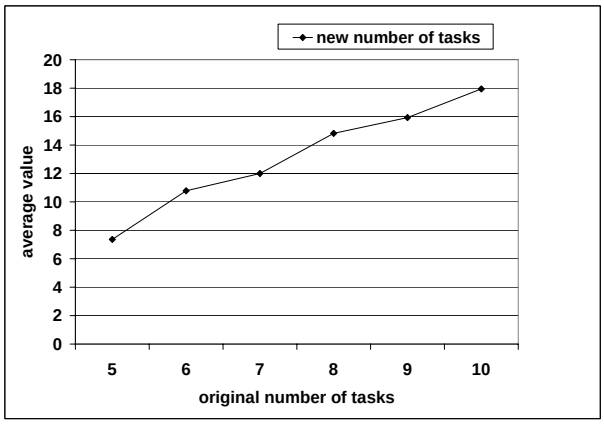


Figure 5.11: Number of FPS tasks

tributes, i.e., priorities and offsets such that the tasks are schedulability is maintained on the original scheduler, while the number of preemptions is reduced.

However, in some cases, the attribute reassignment procedure yields inconsistent attributes for instances of the same task. We solve the phenomena by creating artifacts for the task instances with inconsistent FPS attributes. Hence, no modifications are needed to the original scheduler. We keep the number of artifacts minimal by using ILP. The number of artifacts and the reduced flexibility are the costs we have to pay for reducing the number of preemptions.

Since solving a particular preemption changes the execution pattern of several tasks and, thus, may imply introducing or solving other ones, we use a global approach by constructing a preemption dependency tree, to detect all the preemption dependencies and, implicitly, the set of feasible task attributes that yields the minimum number of preemptions. The preemption dependency tree comprises all possible steps we can perform to eliminate preemptions and all corresponding states representing the new task attributes with corresponding new num-

ber of preemptions, the number of artifacts to be created and the level of reduced flexibility, i.e., the cost we have to pay. Hence, we provide for the ability to select a user-defined optimum state with respect to the trade-off between the number of preemptions and the cost to pay.

Chapter 6

Conclusions

In this thesis we present methods that combine off-line schedule construction with fixed priority run-time scheduling. In particular, we want to take advantage of all benefits provided by off-line scheduling, while using fixed priority scheduling. We use off-line schedules to express complex constraints and predictability for selected tasks. Then, we derive attributes for tasks, such that, if applying FPS at run-time, the tasks will execute flexibly while fulfilling their original constraints.

Thus, the methods solve issues arising from legacy systems, e.g., partition scheduling for avionics applications, and allows to handle constraints not covered by FPS feasibility tests, while using standard FPS at runtime. Also it provides for predictable flexibility, i.e., the restricted execution of selected tasks, e.g., for sampling and actuating in control systems, while enabling runtime flexibility for others.

Our method analyzes an off-line schedule, constructed to solve complex constraints, and derives attributes for fixed priority scheduling such that the tasks, when scheduled by FPS, execute flexibly while fulfilling the original constraints. In certain cases, the method splits tasks into instances, creating artifact tasks, as not all off-line schedules can be expressed directly with FPS. In our approach, we use standard integer linear programing to solve priority inequalities derived from the off-line

schedule and minimize the number of artifact tasks created. Finally, we assign offsets and periods to the task set provided by ILP in order to ensure the correct run-time execution.

In some cases, we have to perform additional splits, due to a violation of the periodicity in the off-line schedule, which gives different offsets for different instances of the same task. By minimizing the number of artifact tasks, our method minimizes the number of offsets in the system as well. The number of artifact tasks and offsets can be decreased by adjusting the target windows, if the resulting loss in flexibility is acceptable. For example, if one particular instance of a task has to be assigned a different offset than the others, we could adjust all of them to match the constrained instance's offset. However, in this case, we would have to reduce the target windows for all instances, resulting in a loss of flexibility.

Our methods do not introduce artifacts or reduce flexibility unless required by constraints: a set of FPS tasks, scheduled off-line according to FPS, and transformed by our method, executes in the same way as the original tasks.

We have applied the method to schedule messages with complex constraints on CAN. We use the information provided by an off-line schedule constructed to solve complex constraints on messages and we derive attributes, i.e., message identifiers, required by CAN's native protocol. At run time, the messages are transmitted and received within time intervals such that the original constraints of the messages are fulfilled.

We take advantage of the CAN particularities, i.e., constant message length and non-preemptive behavior, to relax the constraints to our optimization problem, and, thus, to provide for better solutions.

We solve the requirement of unique message identifiers by simple constraint inclusion to the ILP formulation.

To this point, we have concentrated on reconstructing the off-line schedule. Using the flexibility of the ILP solver, we can add objectives by inclusion in the goal function. In our approach, we assumed that all task dependencies have been resolved off-line.

As one of the advantages provided by FPS is run-time flexibility, we have investigated how existing FPS servers can be used together with FPS task sets obtained by using our transformation method described in chapter 2, while guaranteeing the original complex constraints on the periodic tasks and providing a good service for non-periodic requests. We have shown that problems may arise if the servers are not aware of the complex constraints on the periodic tasks, e.g., if the server is added to the system after the transformation of the off-line schedule to task attributes for FPS. Instead, we proposed the inclusion of the server together with the rest of the periodic tasks and their constraints during the off-line schedule construction.

We have provided a mechanism to handle non-periodic events by existing servers that preserve their capacity during their periods until an aperiodic requirement occurs. The difficulty of scheduling this type of servers together with periodic, complex constrained tasks, is that the server execution, i.e., at which time the server starts its execution within its period, can not be predicted such that it can be included in the off-line schedule construction.

Finally, we proposed a method to reduce the number of preemptions in FPS without modifying its basic mechanism. Hence, the method is directly applicable to FPS systems in which modifications to the original scheduler are not desirable or not even possible.

Assuming the WCET for the tasks, the proposed method analyzes off-line a set of periodic tasks with fixed priorities and offsets, scheduled by FPS, and identifies the number of preemptions that can occur at run-time. Our method reduces the number of preemptions by reassigning task attributes, i.e., priorities and offsets such that the task schedulability is maintained on the original scheduler, while the number of preemptions is reduced.

As the preemption reduction does not come for free, in this work we used a global approach to detect preemption dependencies and to provide for the ability to selectively choose a user-defined level of preemptions with respect to the trade-off between the number of preemptions

and the cost to pay.

To do so, we construct a preemption dependency graph that comprises all possible steps we can perform to eliminate preemptions, and all corresponding states representing the new number of preemptions achieved by the new task attributes, and the cost to pay.

Bibliography

[1] L. Abeni and G. Buttazzo. Integrating Multimedia Applications in Hard Real-Time Systems. In *In Proceedings of International Real-Time Systems Symposium*, 1998.

[2] L. Almeida, P. Pedreiras, and J. A. Fonseca. The FTT-CAN Protocol: Why and How. *IEEE Transactions on Industrial Electronics*, 49(6), Dec 2002.

[3] N. Audsley, K. Tindell, and A. Burns. The end of the line for static cyclic scheduling? In *Proceedings 5th Euromicro Workshop on Real-Time Systems*, pages 36–41, Oulu, Finland, June 1993.

[4] N. Audsley, K. Tindell, and A. Burns. The End Of The Line For Static Cyclic Scheduling? In *Proceedings of the Fifth Euromicro Workshop on Real-Time Systems*, pages 36–41, 1993.

[5] N.C. Audsley. Optimal Priority Assignment and Feasibility of Static Priority Tasks With Arbitrary Start Times. Technical report, Departament of Computer Science, University of York, 1991.

[6] Sanjoy Baruah, Joel Goossens, and Giuseppe Lipari. Implementing Constant-Bandwidth Servers upon Multiprocessor Platforms. In *Proceedings of the IEEE International Real-Time and Embedded Technology and Applications Symposium*, pages 154–163, Sep. 2002.

[7] Sanjoy Baruah and Giuseppe Lipari. Executing Aperiodic Jobs in a Multiprocessor Constant-Bandwidth Server Implementation. In *Euromicro Conference on Real-Time Systems*, Jun. 2004.

[8] I. J. Bate. *Scheduling and Timing Analysis for Safety Critical Real-Time Systems*. PhD thesis, University of York - Department of computer Science, UK, Nov. 1998.

[9] A. Burns, K. Tindell, and A. Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Trans. on Software Engineering*, 21(5):475–80, May 1995.

[10] G. Buttazzo. Rate monotonic vs. EDF: Judgment day. In *Proc. 3rd ACM International Conference on Embedded Software*, Philadephia, USA, Oct 2003.

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

[12] T. Carpenter, K. Driscoll, K. Hoyme, and J. Carciofini. ARINC Scheduling: Problem Definition. In *Proceedings of Real-Time Systems Symposium*, pages 165–169, 1994.

[13] 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.

[14] 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.

[15] R. Dobrin and G. Fohler. Implementing off-line message scheduling on controller area network (CAN). In *Proceedings of the 8th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2001)*, Nice, France, Oct. 2001.

[16] Radu Dobrin, Gerhard Fohler, and Peter Puschner. Translating off-line schedules into task attributes for fixed priority scheduling. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, London, UK, Dec. 2001.

[17] J. Echagüe, I. Ripol, and A Crespo. Hard real-time preemptively scheduling with high context switch cost. In *Procedings of the 7th Euromicro Workshop on Real-Time Systems*, Odense, Denmark, Mar. 1995.

[18] G. Fohler. *Flexibility in Statically Scheduled Real-Time Systems*. PhD thesis, Technische Universität Wien, Austria, Wien, Österreich, April 1994.

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

[20] R. Gerber, S. Hong, and M. Saksena. Guaranteeing Real-Time Requirements with Resource-Based Calibration of Periodic Processes. *IEEE Transactions on Software Engineering*, 21(7), July 1995.

[21] M. Gonzalez Harbour and J.P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Varying Execution Priority. In *Proceedings of Real-Time Systems Symposium*, pages 116–128, Dec. 1991.

[22] 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.

[23] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(J5):390–395, 1986.

[24] Daniel I. Katcher, Hiroshi Arakawa, and Jay K. Strosnider. Engineering and analysis of fixed priority schedulers. *Software Engineering*, 19(9):920–934, 1993.

[25] Saehwa Kim, Seongsoo Hong, and Tae-Hyung Kim. Integrating real-time synchronization schemes into preemption threshold scheduling. In *Proc. 5th IEEE International Symposium on Object-oriented Real-time Distributed Computing*, Crystal City, VA, USA, Apr. 2002.

[26] Saehwa Kim, Seongsoo Hong, and Tae-Hyung Kim. Perfecting preemption threshold scheduling for object-oriented real-time system design: From the perspective of real-time synchronization. In *Proc. Languages, Compilers, and Tools for Embedded Systems*, Berlin, Germany, Jun. 2002.

[27] 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.

[28] H. Kopetz. Why Time-Triggered Architectures will Succeed in Large Hard Real-Time Systems. In *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pages 2–9, 1995.

[29] H. Kopetz, G. Fohler, G. Grünsteidl, H. Kantz, G. Pospischil, P. Puschner, J. Reisinger, R. Schlatterbeck, W. Schütz, A. Vrchoticky, and R. Zainlinger. The distributed, fault-tolerant real-time operating system MARS. *IEEE Operating Systems Newsletter*, 6(1), 1992.

[30] H. Kopetz and G. Grunsteidl. TTP - a Protocol for Fault-Tolerant Real-Time Systems. *Computer*, 27(1):14–23, 1994.

[31] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyne Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong San Kim. Analysis of cache-related preemption delay in

fixed-priority preemtive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, June 1998.

[32] Chang-Gun Lee, Kwangpo Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Bounding cache-related preemption delay for real-time systems. *Software Engineering*, 27(9):805–826, Sep. 2001.

[33] G. Leen and D. Heffernan. Time-Triggered Controller Area Network. *Computing and Control Engineering*, 2001.

[34] J.P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceddings of the Real-Time Systems Symposium*, pages 201–212, Dec. 1990.

[35] J.P. Lehoczky and Sandra Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proceddings of the Real-Time Systems Symposium*, pages 110–123, Dec. 1992.

[36] J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceddings of the Real-Time Symposium*, pages 261–270, 1987.

[37] J.Y-T. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, 2(4):237–250, Dec. 1982.

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

[39] C. D. Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real Time Systems*, 4(1):37–54, March 1992.

[40] P. Marti, J. M. Fuertes, G. Fohler, and K. Ramamritham. Jitter Compensation for Real-Time Control Systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, Dec 2001.

[41] M. Di Natale. Scheduling the CAN Bus with Earliest Deadline Techniques. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 259–268, Dec 2000.

[42] J.C. Palencia and M. Gonzalez Harbour. Schedulability Analysis for Tasks with Static and Dynamic Offsets. In *Proceedings of 19th IEEE Real-Time Systems Symposium*, pages 26–37, 1998.

[43] J.C. Palencia and M. Gonzalez Harbour. Exploiting Precedence Relations in the Schedulability Analysis of Distributed Real-Time Systems. In *Proceedings of 20th IEEE Real-Time Systems Symposium*, 1999.

[44] K. Ramamritham and J. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *IEEE*, 82(1):55–67, Jan. 1994.

[45] Kristian Sandström. *Enforcing Temporal Constraints in Embedded Control Systems*. PhD thesis, Royal Institute of Technology, Sweden, Apr. 2002.

[46] J. Schneider. Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems, 2000.

[47] D. Seto, J.P. Lehoczky, and L. Sha. Task Period Selection and Schedulability in Real-Time Systems. In *Proceedings of Real-Time Systems Symposium*, pages 188–198, 1998.

[48] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority Inheritance Protocols: an Approach to Real-Time Synchronization. *IEEE Transactions on Computer*, 39(9):1175–1185, Sept 1990.

[49] 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.

[50] B. Sprunt, L. Sha, and J.P. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Tasks. *The Journal of Real-Time Systems*, 1989.

[51] M. Spuri and G.C. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. *The Journal of Real-Time Systems*, March 1996.

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

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

[54] T. Tia, W.S. Liu, and M. Shankar. Algorithms and optimality of scheduling aperiodic requests in fixed-priority preemptive systems. *Journal of Real-Time Systems*, 1995.

[55] K. Tindell. Adding Time Offsets to Schedulability Analysis. Technical report, Departament of Computer Science, University of York, January 1994.

[56] K. Tindell, A. Burns, and A.J. Wellings. Calculating Controller Area Network (CAN) message response times. *Contr. Eng. Practice*, 3(8):1163–1169, 1995.

[57] K. Tindell, H. Hansson, and A.J. Wellings. Analizing Real-Time Communications: Controller Area Network (CAN). In *Proceedings of Real-Time Systems Symposium*, pages 259–263, Dec. 1994.

[58] K.W. Tindell. Adding time-offsets to schedulability analysis, internal report, university of york, computer science dept, ycs-94-221.

[59] M. Torngren. Fundamentals of implementing real-time control applications in distributed computer systems. *to appear in Real-Time Systems*, 1997.

[60] TTP-OS: Time-Triggered Operating System with TTP Support.

[61] Y. Wang and M. Saksena. Scheduling fixed priority tasks with preemption threshold. In *In Proceedings, IEEE International Conference on Real-Time Computing Systems and Applications, December 1999*, Dec. 1999.

[62] J. Xu and D. L. Parnas. Priority Scheduling versus Pre-run-time Scheduling. *Real-Time Systems*, 2000.

[63] K.M. Zuberi and K.G. Shin. Scheduling Messages on Controller Area Network for Real-Time CIM Applications. *IEEE Transactions on Robotics and Automation*, 13(2):310–314, Apr. 1997.

Populärvetenskaplig svensk sammanfattning

"Combining Off-line Schedule Construction and Fixed Priority Scheduling in Real-Time Computer Systems"

Datorer har blivit lika vanliga i samhället under de senaste 10 åren som vanliga mikrovågsugnar i hemmet. Förutom hem-PC som numera finns i nästan alla hushåll, är nästan all elekronik i hemmet (till exempel dvd-spelaren eller tv apparaten) eller i bilen, datorstyrt. I de enklaste fallen består dessa system av en dator och ett antal datorprogram som körs på den.

I och med att dessa system blir allt mer avancerade så ökar kraven på datorns effektivitet också, till exempel hur många program kan köras på samma dator samtidigt, medan priserna på den färdiga produkten måste hållas så låga som möjligt för att kunna anpassas till marknaden.

Vissa program i ett datorstyrt system är viktigare än andra att de utförs korrekt med avseende på både funktionalitet och tid. I bilar, till exempel, är det ytterst viktigt att datorprogrammen som styr airbagen eller bromsarna alltid fungerar som de ska, medan cd-växlaren inte är så kritiskt för passagerarnas säkerhet. Både airbagen och cd-växlaren måste reagera på externa händelser (krock eller tryck på play knappen). Medan airbagen måste aktiveras inom en viss tidsintervall, dvs, inte före en krock, men inte för sent efter en krock heller, så spelar det ingen större roll om det tar en halv sekund extra mellan tiden man trycker på

play knappen på cd:n och tiden när låten börjar spelas upp. Alla dessa system måste kunna koexistera utan att påverka varandra på ett negativt sätt, dvs, om cd-växlaren slutar fungera, får det inte påverka bromsarnas funktionalitet.

I vissa system, är grunddesignen gjort på så sätt att det är svårt att lägga till ytterligare funktionalitet, oftast i form av nya datorprogram. Om man, till exempel, vill lägga till ett anti-sladd system i en bil, som kommer att styras av bil datorn, så måste man kunna vara säker på att resten av programmen som körs på samma dator, i synnerhet de kritiska delarna (till ex. airbag), fortfarande kommer att fungera felfritt.

Å andra sidan, ju mera program man lägger till i systemet, desto svårare blir det för datorn att hantera dem. Detta leder oftast till behovet att förnya datorn till en kraftigare modell som ska lätt hantera de gamla programmen. Samtidigt så måste man fortfarande säkerställa att programmen fungerar korrekt. Att garantera att det nya systemet som består av en ny dator och de gamla programmen uppfyller kraven på korrekt funktionalitet, kan vara ett väldigt svår uppgift.

I det här arbetet, vi förseslår metoder som gör det möjligt och lätt att utföra ovannämda uppgifter, dvs, att utöka funktionaliteten i befintliga datorsystem eller att uppgradera systemen medan den kritiska beteendet garanteras. Samtidigt, introducerar vi metoder for att förbättra efektiviteten i befintliga datorstyrda system som används i dagens läge i, till exempel, bil och flygindustrin.