Mälardalen University Doctoral Thesis
No.9

Flexible Scheduling for Media Processing in Resource Constrained Real-Time Systems

Damir Isović

November 2004

MÄLARDALEN UNIVERSITY

Department of Computer Science and Engineering
Mälardalen University
Västerås, Sweden

Copyright © Damir Isović, 2004
ISSN 1651-4238
ISBN 91-88834-70-0
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

The MPEG-2 standard for video coding is predominant in consumer electronics for DVD players, digital satellite receivers, and TVs today. MPEG-2 processing puts high demands on audio/video quality, which is achieved by continuous and synchronized playout without interrupts. At the same time, there are restrictions on the storage media, e.g.., limited size of a DVD disc, communication media, e.g., limited bandwidth of the Internet, display devices, e.g., the processing power, memory and battery life of pocket PCs or video mobile phones, and finally the users, i.e., humans ability of perceiving motion. If the available resources are not sufficient to process a full-size MPEG-2 video, then video stream adaptation must take place. However, this should be done carefully, since in high quality devices, drops in perceived video quality are not tolerated by consumers.

We propose real-time methods for resource reservation of MPEG-2 video stream processing and introduce flexible scheduling mechanisms for video decoding. Our method is a mixed offline and online approach for scheduling of periodic, aperiodic and sporadic tasks, based on slot shifting. We use the offline part of slot shifting to eliminate all types of complex task constraints before the runtime of the system. Then, we propose an online guarantee algorithm for dealing with dynamically arriving tasks. Aperiodic and sporadic tasks are incorporated into the offline schedule by making use of the unused resources and leeways in the schedule. Sporadic tasks are guaranteed offline for the worst-case arrival patterns and scheduled online, where an online algorithm keeps track of arrivals of instances of sporadic tasks to reduce pessimism about future sporadic arrivals and improve response times and acceptance of firm aperiodic tasks. At runtime, our mechanism ensures feasible execution of tasks with complex constraints in the presence of additional tasks or overloads.

We use the scheduling and resource reservation mechanism above to flexibly process MPEG-2 video streams. First, we present results from a study of realistic MPEG-2 video streams to analyze the validity of common assumptions for software decoding and identify a number of misconceptions. Then, we identify constraints imposed by frame buffer handling and discuss their implications on the decoding architecture and timing. Furthermore, we propose realistic timing constraints demanded by high quality MPEG-2 software video decoding. Based on these, we present a MPEG-2 video frame selection algorithm with focus on high video quality perceived by the users, which fully utilize limited resources. Given that not all frames in a stream can be processed, it selects those which will provide the best picture quality while matching the available resources, starting only such decoding, which is guaranteed to be completed. As a final result, we provide a real-time method for flexible scheduling of media processing in resource constrained system. Results from study based on realistic MPEG-2 video underline the effectiveness of our approach.

To my beloved wife Emina
and daughter Hanna

Preface

After finishing my Licentiate thesis in 2001, as a part of my graduate studies, I realized how confusing this exam was to other people. Not even many Swedes were familiar with this odd degree. Is it the same as ”master degree” in the USA? As ”magistrat” in my country of origin, Bosnia and Hercegovina? I have tried to explain it for people with more or less success by describing it as ”half-PhD” or a ”super-engineer” exam. I do not have that problem any more – ”Doctor of Philosophy” sounds pretty much the same all over the world!

The journey here was not easy. Without support of many people, this work, probably, would not have been possible. I am grateful to them all, not only for their technical support, but also good time I have shared with them. Here, I would like to express my gratitude to all (and please forgive me if I forget somebody, it is getting really late:)

First of all, my family: my greatest thanks go to my beloved wife Emina and my daughter Hanna for their love and support during the long night hours when completing this thesis. I am grateful to my parents and family for always encouraging me to pursue my ideas and desires and for their unfailing support and love throughout my entire education. I would never make it without you.

I would like to thank my supervisor Gerhard Fohler for believing in me and leading me all the way to this degree. Thank you, Gerhard, for always pushing me to my limits. It might not have been fun all the time, but now I know it was the right way to go. Doing anything less than my absolute best would have never given me this type of satisfaction that I

feel now. I appreciate it a lot.

My deepest gratitude goes to the rest of the SALSART group: Radu, Tomas, Larisa, and Pau, for the great time we spent together, no only in meeting rooms, but also at pubs. It's been a real pleasure to work with you.

I also want to thank my colleagues at the Department of Computer Science at Mälardalen University, especially the members of the System Design Lab. Thanks, Ylva, for giving me instructions how to describe my research in a popular-scientific way. Without you ”massacring” the first draft of my proposed description, I would never be able to explain to my parents what I do.

During the last phase of my graduate studies, I've spent some time at the University of North Carolina at Chapel Hill. I would like to express my gratitude to all members of the department, especially to Sanjoy and his family, and Shelby, for taking such a good care of me, Emina and Hanna during our visit, and to Ketan for the technical discussions and help. Thanks to Ketan's almost ridiculously extensive knowledge about the darkest secrets of MPEG, many sub-frame level issues became more clear.

I would also like to express my gratitude to all the people that I cooperated with in one way or another, especially people form ReTiS lab in Pisa: Giorgio, Peppe, Marco, Paolo, and peopple from the Philips Research, Liesbeth and Clemens. Thank you for fruitfull discussions and to the reviewers for helpful comments.

Finally, a piece of advice for all graduate students: do not wait with writing the preface until the last day before printing the thesis, as I did! After all the pain and the agony you are going to suffer while ”just writing it down”, believe me, you are not in the right mood to thank people:)

Damir Isovic
Västerås, October 17, 2004, 02:25

Publications

I have authored or co-authored the following publications:

Journals

- Damir Isović, Gerhard Fohler and Liesbeth Steffens: *Real-time issues of MPEG-2 playout in resource constrained systems*, International Journal on Embedded Systems, June 2004.

Articles in collection

- Damir Isović and Christer Norström: *Requirements for Real-Time Components*, Building Reliable Component-Based Systems, editors: Ivica Crnković and Magnus Larsson, Artech House Publishers, 2002.

Conferences and workshops

- Damir Isović and Gerhard Fohler: *Quality aware MPEG-2 stream adaptation in resource constrained systems*, 16th Euromicro Conference on Real-time Systems (ECRTS '04), Catania, Sicily, Italy, July 2004.

- Damir Isović, Gerhard Fohler and Liesbeth Steffens: *Timing constraints of MPEG-2 decoding for high quality video: misconceptions and realistic assumptions*, Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS ’03), Porto, Portugal, July 2003.

- Damir Isović, Gerhard Fohler and Liesbeth Steffens: *Some Misconceptions about Temporal Constraints of MPEG-2 Video Decoding*, WiP of the 23rd IEEE International Real-Time Systems Symposium (RTSS ’03), Austin, Texas, USA, December 2002.

- Damir Isović and Christer Norström: *Components in Real-Time Systems*, The 8th International Conference on Real-Time Computing Systems and Applications (RTCSA ’02), Tokyo, Japan, March 2002.

- Gerhard Fohler, Damir Isović, Tomas Lennvall and Roger Vuolle: *SALSART - A Web Based Cooperative Environment for Offline Real-time Schedule Design*, 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing (PDP ’02), Gran Canaria, Spain, January 2002.

- Damir Isović and Gerhard Fohler: *Efficient Scheduling of Sporadic, Aperiodic, and Periodic Tasks with Complex Constraints*, Proc. of the 21st IEEE Real-Time Systems Symposium (RTSS ’00), Walt Disney World, Orlando, Florida, USA, November 2000.

- Damir Isović, Markus Lindgren and Ivica Crnkovic: *System Development with Real-Time Components*, Proc. of ECOOP 2000 Workshop 22 - Pervasive Component-based systems, Sophia Antipolis and Cannes, France, June 2000.

- Damir Isović and Gerhard Fohler: *Online Handling of Firm Aperiodic Tasks in Time Triggered Systems*, 12th EUROMICRO Conference on Real-Time Systems (ECRTS ’00), Stockholm, Sweden, June 2000.

- Damir Isović and Gerhard Fohler: *Handling Sporadic Tasks in Off-line Scheduled Distributed Real-Time Systems*, 11th EUROMICRO Conference on Real-Time Systems (ECRTS ’99), York, England, July 1999.

Technical reports

- Damir Isović, Gerhard Fohler: *Analysis of MPEG-2 Video Streams*, MRTC Report , Mälardalen Real-Time Research Centre, Mälardalen University, August 2002

- Damir Isović, Gerhard Fohler: *Resource Aware MPEG-2 Playout Using Real-time Scheduling*, Technical Report , Mälardalen Real-Time Research Centre, March 2001.

- Damir Isović, Christer Norstrm: *Components in Real-Time Systems*, MRTC Report, Mälardalen Real-Time Research Centre, Mälardalen University, December 2001.

- Damir Isović, Gerhard Fohler: *Simulation Analysis of Sporadic and Aperiodic Task Handling*, MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-38/2001-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, May 2001

Licentiate Thesis

- Damir Isović: *Handling Sporadic Tasks in Real-time Systems - Combined Offline and Online Approach*, Licentiate Thesis, Mälardalen University Press, June 2001.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In a near future, most of the existing analog home entertainment devices, such as TVs and VCRs, will be replaced by corresponding digital products. In 2008, all analog television broadcasting in Sweden will be replaced by digital signals that better utilize the communication media and provide for a greater variety of TV channels and interactive services that are not possible in the analog domain. Examples of other sources of digital video and audio include DVDs, Video CDs, and Internet.

Compared to the analog domain, digital media introduces additional and different requirements on the environment. At the same time, there are restrictions on the communication and storage media, display devices and users. In their original form, digital multimedia streams are very big in size, while the storage and the communication media have limited resources. Thus, media files must be compressed before being stored on e.g., a DVD, or transmitted through a network, e.g., the Internet. MPEG – Moving Picture Expert Group – is the most popular compression technique for digital video and audio today. It defines a group of standards for digital storage and distribution of video and audio. Currently, the most used standard of the MPEG group is MPEG-2, used in e.g., DVDs or digital satellite broadcasting.

Display devices are also restricted, e.g., with respect to processing power, memory, and battery life. For instance, the processing power of hand held devices, such as pocket PCs or video mobile phones, is not sufficient to play out a full-size video stream without impairing video quality.

Additional requirements are imposed by the user, e.g., human's ability of perceiving motion. An MPEG-2 movie is a stream of still images displayed after each other fast enough that the human eye cannot notice the delay between consecutive pictures, i.e., the stream is perceived as motion. For example, a DVD movie is displayed on a TV set with a rate of approximately 25 frames per second. This implies there are 40 milliseconds per picture available to read the picture from the disc, decode its contents and display it on the screen. Delays in this process may result in severe video quality degradation of the played stream.

Matching video processing requirements to system limitations

One way of matching some of the requirements imposed by processing of MPEG-2 streams with the limitations of the target systems is to use dedicated hardware solutions. However, dedicated hardware cannot compensate for the limitations in the network bandwidth: in the case of video streaming, it does not matter if there is enough processing power to decode a full-size MPEG-2 video stream if there is not enough network bandwidth available for its transmission. Besides, within the next few years, MPEG-2 decoding will move from dedicated hardware to software, for reasons of cost, rapid upgradeability, and configurability. While being more flexible, software solutions are more irregular, since video processing will compete for the CPU with other applications in the system. Besides, cost-effective software media processing requires a high average resource utilization, leading to instability upon worst-case resource demands. Consequently, we need methods for decreasing the load required by media applications in resource constrained systems.

There are three ways for compensating for limited resources for media processing: decrease bit rate of the stream, use degraded decoding algorithm and frame skipping, as depicted in figure 1.1. Which methods should be used depends on the situation. If the network bandwidth is limited, the streaming server can replace the current stream with a lower

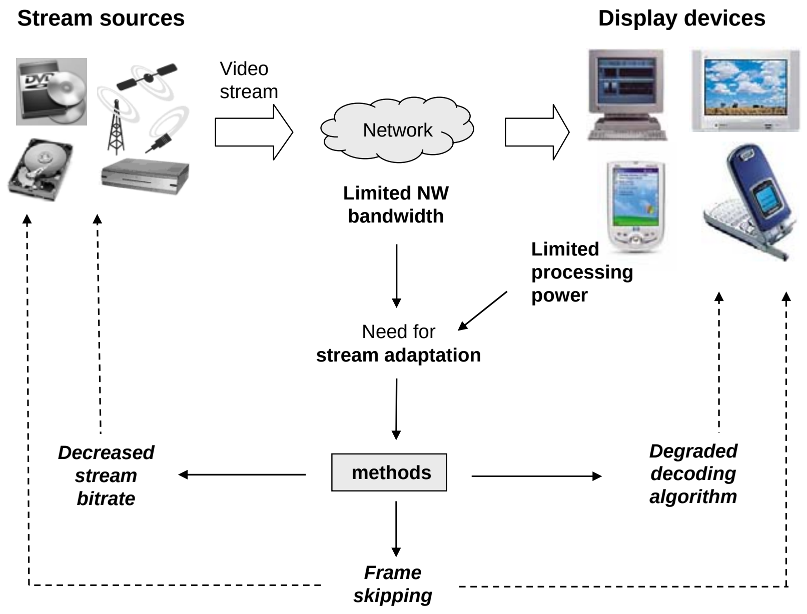


Figure 1.1: MPEG-2 video stream adaptation under limited resources

bit rate alternative. If the processing power on the display device is an issue, then downgraded decoding algorithm that uses less CPU power can be used.

The third apprach is frame skipping. It means if not all video frames in a MPEG-2 stream cannot be decoded due to limited resources, some frames are not decoded and displayed, i.e., they skipped. Frame skipping can be used in both cases above: it can take place both before sending the stream on the network, if the network bandwidth is restricted, or on the display device, if the processing power is limited.

In our work, we use the frame skipping approach with focus on high video quality perceived by the users.

Our approach

In this thesis, we propose a method for flexible scheduling of MPEG-2 video stream processing in resource constrained systems. We use real-time methods for scheduling and resource reservation to fulfill the requirements of software MPEG-2 video decoding.

In the first part of this thesis we present scheduling mechanisms for integrated offline and online scheduling, which we later apply to flexible processing of MPEG-2 video decoding. The scheduling methods presented here provide for easy access of the amount and the distribution of available resources, needed to optimize the adaptation of video streams upon limited resources. We show how to flexibly schedule mixed sets of tasks, i.e., periodic, aperiodic and sporadic, with simple and complex constraints by using an integrated offline and online approach. Offline scheduling methods can resolve many specific constraints but at the expense of runtime flexibility, in particular inability to handle dynamically arriving tasks. Online scheduling provides for flexibility, but it might introduce a high overhead for resolving complex constraints, if even possible. Our method is a combined offline and online approach. We use the offline part of slot shifting, introduced by Fohler [30], to eliminate all types of complex constraints before the runtime of the system. Then we propose a an online guarantee algorithm for dealing with dynamic tasks. Aperiodic and sporadic tasks, are incorporated into the offline schedule by making use of the unused resources and leeways in the schedule. At runtime, our mechanism ensures feasible execution of tasks with complex constraints in the presence of additional tasks or overloads.

In the second part of the thesis, we use the scheduling and available resource reservation mechanisms from the first part of the thesis to flexibly schedule MPEG-2 video streams. First, we present results from a study of realistic MPEG-2 video streams to analyze the validity of common assumptions for software decoding and identify a number of misconceptions. Then, we identify constraints imposed by frame buffer handling and discuss their implications on decoding architecture and timing. Furthermore, we propose realistic timing constraints demanded by high quality MPEG-2 software video decoding. Based on these, we present a quality aware frame selection algorithm, to fully utilize limited resources. Given that not all frames can be processed, it selects those which will provide the best picture quality while matching the available resources, starting only such decoding, which is guaranteed to be completed.

As a final result, we provide a real-time method for flexible scheduling of media processing in resource constrained system. Results from a study based on realistic MPEG-2 video underline the effectiveness of our approach.

We start by giving a general description of real-time systems and MPEG standard, followed by introduction to the two research areas and finally their interaction.

1.1 Real-Time Systems Background

Real-time systems are computing systems in which meeting timing constraints is essential to correctness. Usually, real-time systems are used to control or interact with a physical system, where timing constraints are imposed by the environment. As a consequence, the correct behaviour of these systems depends not only on the result of the computation but also at which time the results are produced [55]. If the system delivers the correct answer after a certain deadline, it can be regarded as having failed.

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

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

Real-time systems can be constructed out of sequential programs, but are typically built of concurrent programs, called *tasks*. A typical timing constraint on a real-time task is the *deadline*, i.e., the maximum time interval within which the task must complete its execution. Depending on the consequences that may occur due to a missed deadline, real-time systems are distinguished into two classes, *hard* and *soft*.

In hard real-time systems all task deadlines must be met, while in soft real-time systems the deadlines are desirable but not necessary. In hard real-time systems, late data is bad data. Soft real-time systems are constrained only by average time constraints, e.g., handling input data from the keyboard. In these systems, late data is still good data. Many systems consist of both hard and soft real-time subsystems, and from now on we will refer to them as *mixed* real-time systems.

1.1.1 Real-time scheduling

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

Offline vs online

Real-time scheduling algorithms fall into two categories [26]: *offline* and *online* scheduling.

In offline scheduling, the scheduler has complete knowledge of the task set and its constraints, such as deadlines, computation times, precedence constraints, etc. Scheduling decisions are based on fixed parameters, assigned to tasks before their activation. The offline guaranteed schedule is stored and dispatched later during runtime of the system. Offline scheduling is also referred as *static* or *pre-runtime* or *table-driven* scheduling.

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

Event-trigged vs time-trigged

There are two fundamentally different principles of how to control the activity of a real-time system, *event-trigged* and *time-trigged*.

In *event-trigged* systems all activities are carried out in response to relevant events external to the system. When a significant event in the outside world happens, it is detected by some sensor, which then causes the attached device (CPU) to get an interrupt. For soft real-time systems with lots of computing power to spare, this approach is simple, and works well. A problem with event-trigged systems is that they can fail under conditions of heavy load, i.e., when many events are happening at once. As an example of an event-trigged system we can mention the SPRING system [52], which applies an online guarantee algorithm with complex task models in distributed environments.

In a time-trigged system, all activities are carried out at certain points in time known a priori. Accordingly, all nodes in time-trigged systems have a common notion of time, based on approximately synchronized clocks. One of the most important advantages of time-trigged control are predictable temporal behaviour of the system, which eases system validation and verification considerably. An example of a time-trigged system is the MARS system [20].

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

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

Resource sufficient vs resource constrained

Scheduling can be further divided into scheduling algorithms that work in *resource sufficient* and those that work into *resource constrained* environments [53], i.e., in overload situations.

A real-time system with enough resources is a system in which we always can guarantee that all functions in the system will be able to perform in time, i.e., before their deadlines. In these systems, the CPU will never get overloaded, since those systems are designed for the worst-case scenario (peak load). In resource sufficient environments, even though tasks arrive dynamically, at any given time all tasks are schedulable. Examples include ABS brake systems, flight control systems, etc., i.e., safety-critical systems.

On the other hand, in resource constrained systems, there will be occasions when we cannot guarantee that all functions will make it in time. One example is a telephone switch system which is designed for the average case load. In most of the cases the telephone switch system works well: when we call somebody, we usually get the dial tone right away, but when we try to dial on the New Year's Eve, when all other people try to call at the same time, the system might not be able to connect our call (or we need to wait for some time).

Preemptive vs non-preemptive

Preemption is an operation of the kernel that interrupts the currently executing task and assigns the processor to a more urgent task ready to execute. Both offline and online scheduling can be either preemptive or non-preemptive. Preemption increases the schedulability of the system while non-preemtion gives an automatic mutual exclusion for access of shared resources. Most of the scheduling algorithms are preemptive.

Heuristic vs optimal

An scheduling algorithm is said to be *optimal* if it minimizes some given cost function defined over the task set. On the other hand, if an algorithm tends toward but does not guarantee to find the optimal schedule is said to be *heuristic*.

1.1.2 System model

Real-time systems span a large part of computer industry. So far most of the real-time systems research has been mostly confined to single node systems and mainly for single-processor scheduling. This needs to be extended for multiple resources and distributed nodes. In our work, we consider a distributed system, i.e., one that consist of several processing and communication nodes [54]. We assume a discrete time model [40]. Time ticks are counted globally, by a synchronized clock with granularity of slot length. Slots have uniform length and start and end at the same time for all nodes in the system. Task periods and deadlines must be multiples of the slot length.

1.1.3 Task model

Real-time systems react on events that can be predictable (e.g., sampling of pulses generated by a pulse generator) or unpredictable (e.g., interrupts). Obviously, we need different task models for different types of events. There are three major types for real-time tasks:

Periodic Tasks

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

Aperiodic Tasks

A type of task that consist of a sequence of identical instances, activated at irregular intervals. Events that triggers an aperiodic task may occur at any time, e.g., a device generates interrupts, an operator presses the emergency button, alarms, etc. In general, aperiodic tasks are viewed as being activated randomly.

Furthermore, aperiodic tasks can be hard, soft and *firm*. Hard aperiodic tasks have stringent timing constraint that must be met, while soft aperiodic do not have deadlines at all. A firm aperiodic task has a deadline that must be met once the task is guaranteed online. The difference between firm and hard aperiodic tasks is that hard tasks are guaranteed offline, while firm tasks are guaranteed online, upon their arrival. They can also be rejected by the guarantee algorithm used.

Sporadic Tasks

Sporadic tasks [45] are introduced to model external events, such as a emergency button being pushed or a train crossing a sensor. However, the interval between successive events is imposed by the environment, i.e., events arrive at the system at arbitrary points in time, but with defined maximum frequency. They are invoked repeatedly with a (non-zero) lower bound on the duration between consecutive occurrences of the same event. Therefore, each sporadic task will be invoked repeatedly with a lower bound on the interval between consecutive invocations i.e., *minimum inter-arrival time* between two consecutive invocations.

In other words, a sporadic task is a dynamic type of task characterized by a minimum inter-arrival time (mint) between consecutive instances. After the minimum inter-arrival time has elapsed, the next instance can get activated at any time – we do not know when.

In the study of mixed real-time systems, it is a common model that periodic tasks are hard and therefore deterministic, while aperiodic tasks are soft or firm. Sporadic tasks, on the other hand, provide enough knowledge to assess their timing. They can therefore be required to be hard.

1.1.4 Simple and complex constraints

We distinguish between *simple* constraints, i.e., period, start-time, and deadline, for the earliest deadline first scheduling model [23], and *complex* constraints. We refer to such relations or attributes of tasks as *complex constraints*, which cannot be expressed directly in the earliest deadline first scheduling model using period, start-time, and deadline. In most of the cases, offline transformations are needed to schedule these at runtime (some can be resolved online at the cost of the higher overhead). Here are some examples of complex constraints:

- Synchronization – Execution sequences, such as sampling - computing - actuating require a *precedence* order of task execution.

- Jitter – The execution start or end of certain tasks, e.g., sampling or actuating in control systems, is constrained by maximum variations.

- Non-periodic execution – Non-periodic constraints, such as some forms of jitter, require instances of tasks to be separated by non constant length intervals. Similar reasoning applies to constraints over more than one instance of a task, e.g., for iterations, data history or ages. A constraint can be of the type “separate the execution of instance i and $i + 4$ by no more than max and no less than min”.

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

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

1.2 Overview of PART I: Efficient Scheduling of Mixed Task Sets with Complex Constraints

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

As a first part of this thesis, we provide mechanisms to handle unpredictable, dynamic events together with predictable ones. We present a combined offline and online approach to deal with a combination of mixed sets of tasks and constraints: periodic tasks with complex and simple constraints, soft and firm aperiodic tasks, and in particular sporadic tasks.

1.2.1 Related work

A variety of algorithms have been presented to handle periodic and dynamically arriving tasks. Here we restrict ourselves to work most significant for our own work.

Aperiodic task handling

One common method to handle aperiodic task is the server-based approach. Dynamic arrivals are given access to the resources reserved for a so called server task. If an aperiodic task arrives it executes with the resources of the server task. If no dynamic task is demanding execution, the server task will not execute.

Handling of firm aperiodic requests using a Total Bandwidth Server has been presented in [51]. Online guarantees of aperiodic tasks in firm periodic environments, where tasks can skip some instances, have been described in [16].

The Deferrable Server [42] algorithm is a method to improve the average response times of aperiodic tasks with respect to polling service.

Sporadic server [50] algorithm allows to enhance the average response time of aperiodic tasks without degrading the utilization bound of periodic tasks. It aims at the shortest response time in the presence of hard real time periodic tasks executing on a fixed priority basis.

Example algorithms for the selection of tasks to reject in overload situations have been discussed in [14], [41], [7], [3]. These algorithms assume control over all tasks in the system and do not take into account the impact of offline scheduled tasks.

Sporadic task handling

There are two major techniques to handle sporadic tasks. One is to assume maximum arrival frequency and fit them in the periodic framework. The other one is to always be prepared for their unknown arrival times by performing an offline schedulability test for their worst-case arrival scenario.

In [45] sporadic tasks are fitted into the periodic framework by transforming them into *pseudo-periodic* tasks. A set of rules are applied to derive the deadline and period of a sporadic task to fulfill the required timing.

An online algorithm for scheduling sporadic tasks with shared resources in hard real-time systems has been presented in [38]. Scheduling of sporadic requests with periodic tasks on an earliest-deadline-first (EDF) basis has been presented in [59].

A schedulability test for sporadic tasks on single processors has been presented in [8]. Necessary and sufficient conditions are derived for a sporadic task set to be feasible.

An offline guarantee algorithm for sporadic tasks based on bandwidth reservation has been presented in [15] for single processor systems.

Complex timing constraints handling

An algorithm for the transformation of precedence constraints on single processor to suit the EDF scheduling model has been presented in [19]. However, many industrial applications require allocation of tasks with precedence constraints on different nodes, i.e., a distributed system with internode communication. The transformation of precedence constraints with an end-to-end deadline in this case requires subtask deadline assignment to create execution windows on the individual nodes so that precedence is fulfilled, e.g., [24].

A schedulability analysis for pairs of tasks communicating via a network instead of decomposition has been presented in [60].

In [68] static scheduling is discussed as a general technique for solving the problem of satisfying complex timing constraints in hard real-time systems. The conclusion is that pre-run-time scheduling is essential to meeting such constraints in large systems.

The application of standard timing constraints, such as deadlines and periods, can sometimes overconstrain specifications. One way to overcome this is to use *dynamic timing constraints* with a feasibility function describing temporal requirements instead of providing concrete constraints such as periods and deadlines. A method that shows how dynamic timing constraints can be used instead with standard scheduling algorithms has been presented in [30].

Combined offline and online scheduling

The slot shifting algorithm to combine offline and online scheduling was presented in [28]. It focuses on inserting aperiodic tasks into offline schedules by modifying the runtime representation of available resources. The use of information about the amount and distribution of unused resources for non-periodic activities is similar to the basic idea of slack stealing [58], [21] which applies to fixed priority scheduling. Slot shifting does not provide for easy removal of guaranteed tasks. Besides, while appropriate for including sequences of aperiodic tasks, the overhead for sporadic task handling becomes too high.

A similar approach for combined offline and online scheduling has been recently proposed in [65, 64, 63]. It creates a generalized offline *pre-schedule* for a set of time-trigged tasks, with some slack left for eventual event-driven workload competing for resources. The difference between this method and the slot shifting is that slot shifting does a reactive approach, i.e., taking an existing offline schedule and trying to accommodate aperiodic and sporadic tasks, while pre-scheduling approach is proactive, i.e., it makes a schedule that can fit the aperiodics and sporadics later on. Besides, this approach is aimed for single processor and independent tasks, whereas slot shifting can be used in distributed real-time systems, with inter-node communication and not independent tasks.

1.2.2 Motivation

The methods for handling dynamic tasks described above can generally be classified as *latest start time* methods, since they all share the characteristic of postponing the execution of hard tasks in order to give more resources to the soft tasks. Normally, as long as all guaranteed tasks meet their deadlines, it does not matter if they complete their execution in advance of their deadline or just before it.

However, these methods concentrate most on particular types of constraints. As mentioned before, a real-time system might need to fulfill some complex constraints, in addition to basic temporal constraints of tasks, such as periods, start-times and deadlines. Those complex constraints can not be expressed as generally as the simple ones. Adding complex constraints to a chosen scheduling strategy increases scheduling overhead [69] or requires new, specific schedulability tests which may have to be developed.

Constraints such as some forms of jitter, e.g., for feedback loop delay in control systems [61], require instances of tasks to be separated by *non-constant* length intervals. In order to fit these constraints into the periodic task model, it can easily happen that we end up with an over-constrained specification. At the same time, algorithms are computationally expensive [6].

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

1.2.3 Approach

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

Our method provides an *offline* schedulability test for sporadic tasks, based on slot shifting [28]. It constructs a worst case scenario for the arrival of the sporadic task set and tries to guarantee it on the top of the offline schedule. The guarantee algorithm is applied at selected slots only. At runtime, it uses the slot shifting mechanisms to feasibly schedule sporadic tasks in union with the offline scheduled periodic tasks, while allowing resources to be reclaimed for aperiodic tasks. Since the major part of preparations is performed offline, the involved online mechanisms are simple. Furthermore, the reuse of resources allows for high resource utilization.

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

Combined offline and online Scheduling

Offline scheduling methods can accommodate many specific constraints but at the expense of runtime flexibility, in particular inability to handle dynamic activities such as aperiodic and sporadic tasks. Consequently, a designer given an application composed of mixed tasks and constraints has to choose which constraints to focus on in the selection of scheduling algorithm; others have to be accommodated as well as possible. If we only use online scheduling, then we might introduce high overhead for resolving complex constraints, or, in the worst case, we cannot resolve them at all.

Our method is a combined offline and online approach: it integrates offline, time-trigged scheduling and dynamic, event-trigged scheduling. We use slot shifting to eliminate all types of complex constraints before the runtime of the system. They are transformed into a simple EDF model, i.e., periodic tasks with start times and deadlines. Dynamic activities, i.e., aperiodic and sporadic tasks, are incorporated into offline schedule by making use of the unused resources and leeways in the schedule.

We assume a resource restricted environment, where dynamic activities have to be guaranteed to fit into the offline schedule, without affecting any of previously scheduled or guaranteed activities. We provide both offline and online mechanisms for dealing with such a priori unknown activities.

The method requires a small runtime data structure, simple runtime mechanisms, going through a list with increments and decrements, provides $O(N)$ acceptance tests, N being the number of aperiodic instances, and facilitates changes in the set of tasks, for example to handle overloads. Furthermore, our method provides for handling of slack of non-periodic tasks as well, e.g., instances of tasks can be separated by intervals other than periods.

Handling aperiodic tasks

The methods presented in this thesis provide for inclusion of both firm and soft aperiodic tasks. Firm tasks must be guaranteed while the soft ones do not require any acceptance test.

Upon arrival of a firm aperiodic task, a test determines whether there are enough resources available to include it feasibly in the set of previously guaranteed tasks and if the scheduling strategy will ensure timely completion. If the task can be accepted, it is guaranteed by providing a mechanism which ensures that the resources it requires will be available for its execution.

Handling sporadic tasks

Offline scheduling is not suitable for handling sporadic tasks due to unknown arrival times. One approach could be to transform sporadic tasks into equivalent pseudo-periodic tasks [45] offline, which can be scheduled simply at runtime. However, this may lead to significant underutilization of the processor time, especially when the deadline of the pseudo-periodic task is small compared to the minimum inter-arrival time of the sporadic task. That because a great amount of time has to be reserved offline, before the runtime of the system, for servicing dynamic request from sporadic tasks. In extreme cases, a task handling an event which is rare, but has a tight deadline, may require reservation of all resources.

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

1.2.4 Contribution summary

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

Our methods use a general technique, capable of incorporating various types of constraints and their combinations. Those are resolved in the offline part of the method, without degrading the system performance at runtime. Table 1.2 gives an overview of when different types of tasks are handled by our method (simple periodic and complex periodic in the table refer to periodic tasks with simple respective complex constraints).

Periodic tasks are completely handled offline. Complex constraints are translated into simple ones, i.e., only start times and deadlines and the tasks are scheduled to execute before their deadline. Originally, the tasks are scheduled to execute as soon as possible, but they may also be shifted online within their feasibility intervals in order to accept more aperiodic or sporadic tasks.

Sporadic tasks are guaranteed offline for the worst-case arrival patterns and scheduled online, where an online algorithm keeps track of

		Periodic with constraints		**Sporadic**	**Aperiodic**	
		Simple	**Complex**		**Firm**	**Soft**
		• Periods • Deadlines • Start times	• End-to-end dl • Inst. separation • Distribution • Jitter etc.	Minimum separation between instances	Deadlines	No deadlines
Offline	Scheduling	**x**	**x**			
	Guarantee			**x**		
Online	Scheduling	**x**	**x**	**x**	**x**	**x**
	Guarantee				**x**	

Figure 1.2: Overview of handling various types of tasks and constraints

arrivals of instances of sporadic tasks to reduce pessimism about future sporadic arrivals and improve response times and acceptance of firm aperiodic tasks.

Aperiodic tasks are completely handled online. Firm aperiodics are guaranteed while the soft ones are not. We perform an online guarantee algorithm on arriving firm aperiodic tasks to see if we can accept them without violating the deadlines of any other scheduled or guaranteed task. Soft aperiodic tasks are not guaranteed: they are executed if no other task in the system executes at the moment.

Chapters 2 and 3 give details about handling of the tasks above. The final result is a method that is capable of dealing with all mentioned task types and constraints and their interactions.

We use scheduling algorithms and resource reservation mechanism from the mixed task scheduling part presented above to flexibly schedule processing of MPEG-2 video streams. We start by giving a general introduction to MPEG-2 standard for digital video coding, followed by the description of our method.

1.3 MPEG-2 Background

The Moving Picture Experts Group (MPEG) standard for coded representation of digital audio and video [1] is used in a wide range of applications. MPEG is one of the most popular audio/video digital compression technique because it is not just a single standard. Instead, it is a range of standards suitable for different applications but based on similar principles.

MPEG group defines several standards in digital video, among it the MPEG-1 standard, used e.g., in Video CDs, the MPEG-2 standard, used e.g., in DVDs, digital video broadcasting, high-definition TVs (HDTV), and the MPEG-4 standard, used e.g., in picture phones, streaming media, Internet. It also defines several audio standards – among them MP3 and AAC.

MPEG-2 is currently the most used video standard of the MPEG group. In particular, MPEG-2 has become the coding standard for digital video streams in consumer content and devices, such as DVD movies and digital television set top boxes for Digital Video Broadcasting (DVB).

It should be noted that MPEG is a standard for the format, a syntax, not for the actual encoding. The specification only defines the bit stream syntax and decoding process. Generally, this means that any decoders which conform to the specification should produce near identical output pictures. However, decoders may differ in how they respond to errors introduced in the transmission channel. For example, an advanced decoder might attempt to conceal faults in the decoded picture if it detects errors in the bit stream. As a consequence, the same content, e.g., a movie, can be encoded in many ways while adhering to the same standard. In fact, MPEG encoding has to meet diverse demands, depending, e.g., on the medium of distribution, such as overall size in the case of DVD, maximum bit rate for DVB, or speed of encoding for live broadcasts.

In this thesis we deal with MPEG-2, which is currently the most used MPEG standard. It aims to be a generic video coding system supporting a diverse range of applications. The standard covers four quality

Level	Max resolution	Max bit rate	Example application
low	352x288	4 Mbps	VHS-quality video
normal	720x576	15 Mbps	DVB, DVD
high 1440	1440x1152	80 Mbps	HDTV
high	1920x1152	100 Mbps	wide-screen HDTV

Table 1.1: MPEG-2 quality levels

levels of video resolution, each targeted at a particular application domain, see table 1.1. Note that the constraints are upper limits and that the codecs may be operated below these limits. In broadcasting terms, standard-definition TV requires main level and high-definition TV requires high-1440 level. The bit rate required to achieve a particular level of picture quality approximately scales with resolution.

1.3.1 MPEG-2 video compression

Motion video is a sequence of pictures, called *frames* in MPEG, each picture consisting of an array of pixels. For uncompressed video, its size is very large. To deal with this problem, video compression is used in order to reduce the size. The basic idea is to transform a stream of discrete samples into a bit stream of tokens, which takes less space.

The MPEG-2 video compression algorithm dramatically decreases the amount of storage space required to record video sequences by eliminating redundant and non-essential image information from the stored data.

Temporal redundancy takes advantage of similarity between successive pictures. It arises when successive pictures of video display images of the same scene. It is common for the content of the scene to remain fixed or to change only slightly between adjacent pictures. Often the only difference is that some parts of the picture have shifted slightly between the pictures. MPEG compression exploits this temporal redundancy by just sending the instructions for shifting pieces of the previous picture to their new positions in the current picture. The coding technique that exploit temporal redundancy is called *Inter*-coding (inter=between).

Spatial redundancy takes advantage of similarity among most neighboring pixels within the same picture. It occurs because parts of the picture are often replicated (with minor changes) within a single picture. For example, regions of sky or walls are almost entirely the same color. If several pixels point in the same area, MPEG compression exploits this spatial redundancy by sending the color for whole region just once, instead of sending it for each pixel. The coding technique that exploit spatial redundancy is called *Intra*-coding (intra=within).

Another way to archive higher compression ratios is *amplitude scaling*, i.e., the reduction of the color depth of each pixel in a picture and *color space scaling*, i.e., the number of colors available for displaying an image is reduced.

Furthermore, an MPEG-2 video stream can be coded with constant or variable bit rate. *Constant bit rate*, CBR, means that the rate at which the video data should be consumed is constant. It varies the quality level of the video frames in order to ensure a consistent bit rate throughout an encoded file. *Variable bit rate*, VBR, varies the amount of output data in each time segment based on the complexity of the input data in that segment. The goal is to maintain constant quality instead of maintaining a constant data rate by making intelligent bit-allocation decisions during the encoding process. CBR is useful for streaming multimedia content on limited capacity channels since CBR would make usage all of the available capacity. VBR is preferred for storage because it makes better use of storage space: more space is allocated to more complex segments while less space is allocated to less complex segments.

1.3.2 MPEG-2 stream organization

The output of a single video or audio encoder is known as *elementary stream*. MPEG standard defines ways of multiplexing more than one elementary stream (video, audio and data) into one single system stream. In MPEG-2 there are two different types of system streams, transport stream and program stream.

The *program stream* is used for combining together elementary streams that have a common time base and need to be displayed in a synchronized way. Such streams are suited for transmission in a relatively error-free environment and enable easy software processing of the received data. Program Stream packets may be of variable and relatively great length. This form of multiplexing is used for video playback and for some network applications. DVD uses Program Streams.

The *transport stream* is used for multiplexing streams that do not use a common time base. The transport streams packets have fixed length of 188 bytes. Transport streams are suited for transmission in which there may be potential packet loss or corruption by noise, or/and where there is a need to send more than one program at a time. Digital broadcasting uses Transport Streams.

In this thesis we deal with MPEG-2 Elementary Video Streams. We use real-time methods to adjust video streams in overload situations.

1.4 Overview of PART II: Real-Time Processing of MPEG-2 Video in Resource Constrained Systems

The MPEG-2 standard is predominant in consumer electronics for DVD players, digital satellite receivers, and TVs today. One common thing for all devices is that the encoded content has to be decoded and played out. Decoding can be performed in hardware or in software, or in a mix of both. Both dedicated and programmable decoders can be based on average-case requirements if they provide means to gracefully handle overload situations. If not, both must support worst-case requirements.

Within the next few years, MPEG-2 decoding will move from dedicated hardware to software, for reasons of cost, rapid upgradeability, and configurability. In a software implementation, it is possible to use the slack on the processor for other applications in average case. With dedicated hardware, there are no such possibilities. As a consequence, the behavior of a software decoder will be less regular than that of a dedicated hardware decoder. Coping with these irregularities is one of the objectives dealt with in this thesis.

Furthermore, video will not only be watched on classic TV sets, but increasingly displayed on smaller devices ranging from mobile phones to web pads, providing mobility. Consequently, MPEG-2 decoding will be performed in software under limited resources.

Most current software decoders, however, operate under the assumption of sufficient resources, using buffering and rate adjustment based on average-case assumptions. These provide acceptable quality for applications such as video transmissions over the Internet, when decreases in quality, delays, uneven motion or changes in speed are tolerable. In high quality consumer terminals, however, quality losses of such methods are not acceptable. In fact, producers of such devices have argued to mandate the use of hard real-time methods instead [10].

In this thesis, we present methods for quality aware MPEG-2 video stream adaptation under limited resources, based on realistic timing constraints for MPEG-2 decoding.

1.4.1 Related work

Here is an overview of related work relevant to ours.

Real-time multimedia processing

The Constant Bandwidth Server (CBS) algorithm for integrating multimedia and hard real-time tasks has been presented in [2]. It provides real-time guarantees for hard real-time tasks and probabilistic guarantees for soft multimedia tasks. Hard tasks are guaranteed based on worst-case execution time and minimum interarrival times, while CBS is used for soft multimedia tasks. Each multimedia task is assigned a maximum bandwidth, calculated using the *average* execution time and desired activation time. If a task needs more bandwidth, it may slow down, but still not jeopardize hard tasks. Our work shows that the average assumptions will not hold for a significant number of cases.

A method for real-time scheduling and admission control of MPEG-2 video streams that fits the need for adaptive CPU scheduling has been presented in [25]. The method qualifies for continuous re-processing and guarantees quality of service. However, it requires separate decoding tasks for different frame types. We found this to be an unnecessary restriction: one decoding task can be used for all frame types. Besides, frame skipping decisions are made based on frame type only. As a consequence, stream quality might be degraded more than neccessary.

An approach that allows close-to-average-case resource allocation to a single video processing task has been proposed in [67]. It is based on asynchronous, scalable processing, and QoS adaptation. No frame type distinction has been made and the method applies only on a special case when the display rate of the display device is equal to the frame rate of the movie. We solve this problem for the general case, where the display rates are different from the frame rates.

A method to process multimedia in fixed-priority based systems has been presented in [11]. Resource allocation for media processing is achieved by using periodic budgets provided by a budget scheduler. The method introduces notion of *conditionaly guaranteed budgets* as a way to handle structural overloads. The idea is to assign budgets to multimedia applications and if an application is not using its budget, it is given to another application. The method requires extensions to existing budget scheduler and online budget management.

A frame skipping pattern based on QoS-human has been presented in [46]. QoS-human is a measurement of video quality from a human perception, i.e., a group of people watch movies with different number of skipped frames and write down their perception of the video quality. Then, the user perception is mapped to the number of skipped frames to determine different values of QoS-human. However, only one skipping criterion, QoS human, has been applied when selecting frames, taking no consideration about frame sizes, buffer and latency requirements, or compression methods used.

Quality reduction for MPEG decoding and other video algorithms is discussed in [48], [70], [32], and [39]. The decoder reduces the load by using a downgraded decoding algorithm. This approach requires algorithms that can be downgraded, with sufficient quality levels to allow smooth degradation. Such algorithms are not yet widely available.

Worst-case execution times for multimedia tasks

It is difficult to predict WCET for decoding parts. MPEG-2 can use different bit rates which can result in large differences in decoding times for different streams. This could lead to big overestimations of the WCETs. Work on predicting MPEG execution times has been presented in [9, 12]. It assumes a linear relationship between frame size and decoding, which we show not to be the case in general.

The design and implementation of a software decoder for MPEG video bit streams is described in [44]. It shows how MPEG video could be decoded in real-time using a software-only implementation on desktop computers. Worst-case execution times for the different parts of the decoding process are reported. Frame prediction from reference frames is found to be most computationally expensive. We relate to this findings when making frame skipping decisions if the processing power is limited.

1.4.2 Motivation

MPEG-2 encoding has to meet diverse demands, depending, e.g., on the medium of distribution, such as overall size in the case of DVD, maximum bit rate for DVB, or encoding speed for live broadcasts. In the case of DVD and DVB, sophisticated provisions to apply spatial and temporal compression are applied, while a very simple, but quickly coded stream will be used for the live broadcast. Consequently, video streams, and in particular their decoding demands will vary greatly between different media.

Most standard decoders fail to satisfy the demands of MPEG-2 in overload situations as they do not consider the specifics of this compression standard. In resource limited situations the processor cannot work fast enough to decode all the frames, the workload for the software decoder has to be reduced. One way to achieve this is skip some of the frames.

Naive best-effort decoders perform frame skipping by simply running out of time at frame display time, incurring either a sudden disturbance in smoothness, as pictures are missing, or a delay of subsequent frames, disturbing motion speed. As frame decoding starts and proceeds without knowing about timely completion, it may happen that the resources are fully used, but wasted, as partially decoded video frames are generally not useful. In extreme cases, the decoding of a large and important frame might just not make it, therefore being lost and impeding quality, while simply skipping to decode a small preceding frame might have freed the resources for completion, with only slight quality reduction.

In addition, skipping a frame may affect also other frames due to inter frame dependencies. In a typical movie, a single frame skip can ruin around 0.5 seconds of motion video. Thus frame skipping needs appropriate assumptions and constraints about streams to be effective.

1.4.3 Approach

In this thesis we present a method for quality aware MPEG-2 stream adaptation in resource constrained systems. The method provides best quality by selecting frames if not all can be decoded under limited resources. It is based on a priority ordering for frame skipping taking frame importance for the overall video quality into account. It creates ensembles of decoding tasks for the video frames, each with timing constraints suited specifically for the particular frame, transforming the video stream into such tailored for actual demand and available resources.

Using a real-time system for resource management, the frame selection algorithm takes into account the actual state of the system, by determining the best frames utilizing the available resources and considering the priority ordering for skipping. Thus, our algorithm selects frames based on concrete frame knowledge and ensures that only decoding of frames which can be completed in time is started.

Below is the overview of the steps used in our method. Each of them will be described in details in separate chapters of this thesis.

MPEG-2 processing under limited resources

As a initial step in our work, we have studied MPEG-2 standard in details and looked into the requirements for timely processing of MPEG-2 video streams. We identified the stream requirements, as well as the buffer and latency requirements that need to be fulfilled for smooth playout. Furthermore, we outlined possible methods for stream adaptation when the system resources are not enough to process entire stream.

Timing constrains for MPEG-2 decoding

Video stream processing has real-time deadlines in a sense missing a decoding deadline of an important frame can result in significant visual artifact. Based on the requirements above, we present actual demands for MPEG-2 playout and derive timing constraints for frame decoding. We show that standard, fixed timing constraints are restrictive and flexible ones are better suited for MPEG-2 software decoding.

Analysis of MPEG-2 streams

As the next step towards a method for flexible processing of media stream, we have performed a proper analysis of diverse MPEG-2 video streams to identify realistic assumptions about MPEG-2. We have matched the results with common assumptions about MPEG, and found a number of misconceptions present in the literature.

Criteria for frame skipping

Frame skipping needs appropriate assumptions to be effective. Dropping the wrong frame at the wrong time can result in a noticeable disturbance in the played video stream. Here we use the realistic assumptions from our MPEG-2 analysis to propose a set of criteria for frame skipping.

Frame selection algorithm

Based on identified frame skipping criteria, we present an algorithm for quality aware frame selection when it is not possible to decode all frames in time. We apply the proposed criteria on a set of frames and assign different importance values to the frames. These will be used to make decision which frames are to be skipped first in overload situations.

Online stream adaptation

Here we unite the frame selection algorithm, decoding timing constraints and real-time resource management from our work on mixed task sets handline to provide a method for for quality aware MPEG-2 stream adaptation in resource constrained systems. The algorithm provides best video quality by selecting frames if not all can be decoded under limited resources.

While the frame selection algorithm is independent of the actual guarantee algorithm used, making it suitable to work with a variety of algorithms and paradigms, we present its use with a concrete scheduler – the one that we used to schedule mixed sets of task in the first part of the thesis.

1.4.4 Contribution summary

Here is a summary of contributions from our work on MPEG-2 video stream processing. We have:

- identified requirements for real-time MPEG-2 processing
- identified misconceptions about MPEG-2
- proposed realistic assumptions for MPEG-2

- proposed criteria for frame skipping
- derived timing constraints for MPEG-2 decoding
- showed how MPEG-2 streams can flexibly be scheduled under limited resources

We present a method for quality aware MPEG-2 stream adaptation under limited resources, based on realistic timing constraints for MPEG decoding. As an example, we show how we can adjust streams in the context of our previous work, i.e., combined offline and online scheduling. Simulation study underlines the effectiveness of our approach.

1.5 Relation between Contributions in PART I and PART II

In the first part of the thesis we show how we can flexibly schedule mixed sets of tasks, i.e., periodic, aperiodic and sporadic, with simple and complex constraints by using integrated offline and online approach based on the slot shifting method [28].

In the second part, we present a method for flexible processing of media streams under limited resources. Here we need a mechanism to access the available system resources in order to know how to adapt a video stream, i.e., how many frames can be timely decoded. We also need a real-time scheduler to schedule processing of the stream.

We use the scheduling mechanism and the resource reservation mechanism from the first part of the thesis to flexibly schedule MPEG-2 video streams. As a final result of the thesis, we provide a real-time method for flexible scheduling of media processing in resource constrained system.

1.6 Outline of the thesis

The rest of this thesis is organized as follows:

Chapter 2 describes handling of soft and firm aperiodic task together with offline scheduled tasks. It starts by giving an overview of the original slot shifting method for joint online and offline scheduling, followed by the new method for handling firm aperiodic tasks.

Chapter 3 presents a method to handle sporadic tasks by guaranteeing them offline for the worst-case, and reducing this pessimism online by keeping track on sporadic arrivals. As a final result, we present a method to schedule mixed sets of tasks with simple and complex constraints.

Chapter 4 gives an overview of MPEG-2 video streams and sets up real-time model for their processing. Latency and buffer requirements are discussed. This chapter gives a basis for the remaining work on MPEG in this thesis.

Chapter 5 uses results and finding about MPEG-2 processing and requirements from chapter 4 to derive realistic timing constraints for MPEG-2 video decoding.

Chapter 6 presents an exhaustive analysis of MPEG-2 video streams, identifies a number of misconceptions about MPEG-2 and proposes realistic assumptions about MPEG-2 processing.

Chapter 7 identifies valid criteria for frame skipping, based on assumptions from previous chapter, and proposes an algorithm for quality aware frame selection when it is not possible to decode all frames in time.

Chapter 8 presents a final method for quality aware MPEG-2 stream adaptation. It combines the two research parts by using the scheduling mechanisms from the first part to flexible schedule processing of MPEG-2 video in resource limited systems.

Chapter 9 concludes the thesis and outlines possible application areas and future work.

A significant amount of our work and results has been moved to several appendixes for readability reasons:

Appendix A presents all simulation results for scheduling of mixed tasks sets. It gives details about the simulation setup, performed experiments and confidence intervals.

Appendix B contains analysis setup and results for diverse realistic MPEG-2 video streams.

Appendix C gives an overview of all tools used for obtaining results in this thesis. Both own implemented tools and tools done by other people but used by us are described here.

– PART I –

Efficient Scheduling of Mixed Task Sets with Complex Contraints

Chapter 2

Aperiodic Task Handling

A number of industrial applications advocate the use of time-triggered approaches for reasons of predictability, cost, product reuse, and maintenance. The rigid offline scheduling schemes used for time-triggered systems, however, do not provide for flexibility. Offline scheduling methods can resolve many specific constraints but at the expense of runtime flexibility, in particular inability to handle dynamically arriving tasks, such as aperiodic and sporadic tasks. At runtime, aperiodic tasks that handle asynchronous external events can only be included into the unused resources of the offline schedule, supporting neither guarantees nor fast response times.

In this chapter we present an algorithm for flexible handling of firm aperiodic tasks in offline scheduled systems. Aperiodic tasks that arrive at runtime, are guaranteed and incorporated into the offline schedule by making use of the unused resources and leeways in the schedule. We use the offline part of slot shifting [30], to eliminate all types of complex constraints before the runtime of the system. Then we propose a new online guarantee algorithm for dealing with dynamic tasks. Our algorithm provides an $O(N)$ complexity acceptance test, where N is the number of aperiodic tasks, to determine if a set of aperiodics can be feasibly included into the offline scheduled tasks, and does not require runtime handling of resource reservation for guaranteed tasks. Thus, it supports flexible schemes for rejection and removal of aperiodic tasks, overload handling, and simple reclaiming of resources. As a result, our algorithm [34] provides for a combination of offline scheduling and online firm aperiodic task handling.

We start by presenting the basic idea for aperiodic task handling in section 2.1, followed by the description of the original slot shifting approach, section 2.2. In section 2.3, we present our new online algorithm for flexible aperiodic task handling. We compare the original slot shifting approach for aperiodic handling with the new method in section 2.4 followed by the simulation results, section 2.5 and the chapter summary in section 2.6.

2.1 Basic Idea for Aperiodic Task Guarantee

Guaranteeing and handling of firm aperiodic tasks involves three steps:

Acceptance test

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

Reservation of resources

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

Rejection strategy

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

2.2 Slot Shifting - Original Approach

In this section, we briefly describe the slot shifting method [28] which we use as a basis to combine offline and online scheduling. It provides for the efficient handling of aperiodic tasks on top of a table-driven, offline schedule with general task constraints. Slot shifting extracts information about unused resources and leeway in an offline schedule and uses this information to add tasks feasibly, i.e., without violating requirements on the already scheduled tasks.

2.2.1 Offline preparations

First, a standard offline scheduler, e.g., [49], or [29] creates scheduling tables for the periodic tasks. The scheduling tables list fixed start- and end times of task executions, eliminating all flexibility. The only assignments fixed by the specification of the tasks' feasibility, however, are the initiating and concluding tasks in the precedence graph, all other tasks may vary within the precedence order, i.e., they can be shifted.

After offline scheduling, and calculation of start-times and deadlines, the schedule is divided into a set of *disjoint execution intervals* for each node. *Spare capacities* to represent the amount of available resources are defined for these intervals.

Each deadline calculated for a task defines the end of an interval I_i. Several tasks with the same deadline constitute one interval. Note that these intervals differ from execution windows, i.e. start times and deadline: execution windows can overlap, intervals with spare capacities, as defined here, are disjoint. The deadline of an interval is identical to the deadline of the task. The start, however, is defined as the maximum of the end of the previous interval or the earliest start time of the task. The end of the previous interval may be later than the earliest start time, or earlier (empty interval). Thus it is possible that a task executes outside its interval, i.e., earlier than the interval start, but not before its earliest start time.

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

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

The length of I_i, minus the sum of the activities assigned to it, is the amount of idle time in that interval. These have to be decreased by the amount “lent” to subsequent intervals: Tasks may execute in intervals prior to the one they are assigned to. Then they “borrow” spare capacity from the “earlier” interval. See [28] for details on the borrowing mechanism of slot shifting.

2.2.2 Online mechanism

After determination of intervals and spare capacities, the offline preparations are completed and the amount and location of unused resources is available for online use, i.e., for guaranteeing firm aperiodic tasks. The basic idea is to use *two level EDF*, i.e., to schedule tasks according to - “normal level”, but give priority -“priority level” to an offline task when it needs to start at latest, similar to the basic idea of slack stealing [58] [21] for fixed priority scheduling. Thus, the CPU is not completely available for runtime tasks, but reduced by the amount allocated for offline tasks. So, we need to know the amount and location of resources available after the offline tasks are guaranteed, which we calculate in the offline part of slot shifting as described above.

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

Guarantee algorithm

Assume that an aperiodic task A with deadline $dl(A)$ is tested for guarantee at current time t. We identify three parts of the total spare capacities available:

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

If the sum of all three is larger than the worst-case execution time of A, then A can be accommodated, and therefore guaranteed.

Upon guarantee of a task, the spare capacities are updated to reflect the decrease in available resources. Also, if $dl(A)$ is not equal to the end of an interval, the interval in which $dl(A)$ occurs must be split, resulting in a creation of a new interval.

Scheduling

If the spare capacities of the current interval $sc(I_c) > 0$, EDF is applied on the set of ready tasks - “normal level”. $sc(I_c) = 0$ indicates that a guaranteed task has to be executed or else a deadline violation in the task set will occur. It will execute immediately - “priority level”. Since the amount of time spent at priority level is known and represented in spare capacity, guarantee algorithms include this information.

In original slot shifting, after each scheduling decision, the spare capacities of the affected intervals are updated. If, in the current interval I_c, an aperiodic task executes, or the CPU remains idle for one slot, current spare capacity in I_c is decreased. If an offline task assigned to I_c executes spare capacity does not change. If an offline task T assigned to a later interval $I_j, j > c$ executes, the spare capacity of I_j is increased - T was supposed to execute there but does not, and that of I_c decreased. If I_j “borrowed” spare capacity, the “lending” interval(s) will be updated. The reader is referred to [27, 28] for details on slot shifting.

2.3 Slot Shifting - New approach

The runtime mechanisms of the original version of slot shifting added tasks by modifying this data structure, creating new intervals, which is not suitable for frequent changes as required by e.g., sporadic tasks. Our new guarantee method separates acceptance and guarantee. It only modifies spare capacity, without creating new intervals, which eliminates the online modifications of intervals and spare capacities as in original slot shifting and, thus, allows rejection strategies over the entire aperiodic task set.

2.3.1 Basic idea

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

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

2.3.2 Algorithm description

Let $\mathcal{G}_{t_1}$ denote a set of guaranteed, not yet finished firm aperiodic tasks at time t_1, ordered by increasing deadlines:

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

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

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

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

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

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

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

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

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

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

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

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

2.3.3 Acceptance test for aperiodic tasks

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

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

Let $\mathcal{A} = \{A_1, A_2, ..., A_n\}$ be a set of firm aperiodic tasks that need to be scheduled together with the offline tasks. We accept the aperiodic set if each task in $\mathcal{A}$ is guaranteed to complete before its deadline, i.e., the following must hold:

$$\forall i, 1 \leq i \leq n :\ c(A_i) \leq \begin{cases} \text{sc}[t, dl(A_1)] & , i = 1 \\ \text{sc}[ft(A_{i-1}), dl(A_i)] & , i > 1 \end{cases}$$

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

Note that the spare capacities are not distributed in a uniform way throughout the schedule. Rather, as described in 2.2, the schedule is divided into intervals, each with an individual value of spare capacity. Consequently, the amount of spare capacity in a window depends on the position of that window in the schedule.

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

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

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

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

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

$$R[t_1, t_2] = (t_2 - t_1) - max(sc[t_1, t_2], 0)$$

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

2.3.4 Algorithm description

We now present the acceptance test for a firm aperiodic tsk A and algorithm for finishing time calculation in pseudo code.

Step 1: Let $\mathcal{G}_t$ denote the set of previously guaranteed aperiodic tasks at time t, with sorted deadlines, as described in subsection 2.3.2 above. Find the position of the last task, G_i, in the set $\mathcal{G}_t$, that has deadline before the currently guranteed task A, i.e., the place in $\mathcal{G}_t$ where A will be inserted:

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

Step 2: Get the finishing time of A based on the finishing time of its predecessor G_i, and A's execution demand. The start point for calculation of the finishing time is either the current time t or the finishing time of A's predecessor, whichever is later (see below for details on *getFinishingTime(..)* function).

$$ft = getFinishingTime(max(ft(G_i), t), c(A));$$

Step 3: If A can be finished before its deadline, then go through all the tasks in $\mathcal{G}_t$ with deadlines after $dl(A)$ and for each task calculate its new finishing time, with respect to the new task A (execution of all tasks with the deadline later by A will be delayed by A). New finishing times are then compared to the tasks' deadlines. If any of the investigated tasks fails to complete before its deadline, that means that adding A to $\mathcal{G}_t$ would result in a set that is not feasible, i.e., not all tasks in $\mathcal{G}_t$ will complete before their deadlines. We can either reject A or some other task(s) in $\mathcal{G}_t$. Otherwise, if all tasks can be finished by their deadlines even if A added, then insert A into the set of previously guaranteed firm aperiodic tasks $\mathcal{G}_t$.

```
$if(ft \leq dl(A))\{$

    /* check if accepting $A$ will cause any of the previously
    /* guaranteed firm aperiodic tasks to miss its deadline*/
    $for(j = i + 1; j < n; j + +)\{$
        $ft = getFT(ft, c_t(G_j));$
        $if(ft > dl(G_j))\{$
            /* not feasible! */
            $reject = true;$
            $break;$
        $\}$
    $\}$
```

```
        $if(!reject)$
            $insert(A, \mathcal{G}_t)$;
        $else$
            /* apply some rejection startegy */
    }
    $else$ reject $A$
```

Function: *getFinishingTime()* calculates the finishing time of a task based on predecessor task's finishing time, ft_p, and the remaining execution demand, c_r, of the task to investigate. First, it calculates the remaining spare capacity, sc_r, of the current interval, I_k, i.e., the interval that contains the finishing time of the previous task, ft_p, which is the earliest possible start time for the current task. Without offline tasks, finishing time is equal to the finishing time of the predecessor task increased by the computation demand of the current task. In the presence of offline tasks, however, we do not have all CPU time available, but only the spare capacities in intervals. Hence, we need to go through all the intervals between the starting point ft_p until execution demand is exhausted, i.e., we "simulate" the execution of the investigated task by going through intervals and "filling up free slots", until the remaining execution time is exhausted. The function returns the calculated finishing time of the current task.

```
$getFinishingTime(ft_p, c_r)\{$

    /* get current interval */
    $I_k = getInterval(ft_p)$;

    /* get remaining spare capacity of the current interval */
    $sc_r = start(I_k) + sc(I_k) - ft_p$;

    /* go through the intervals and "fill up" free slots
    until the remaining execution demand is exhaused */
    $while(c_r > sc_r)\{$
        $if(sc_r\ (I_k) > 0)$
                $c_r = c_r - sc_r;$
        $k++;$
        $ft_p = start(I_k)$
        $sc_r = sc(I_k);$
    $\}$
    return $(ft_p + c_r);$
$\}$
```

2.3.5 Complexity

The complexity of our algorithm is $O(N)$, N being the number of firm aperiodic tasks to be guaranteed, because we go through all tasks only once, and calculate their finishing times on the way, as depicted in figure 2.1. The *for-loop* picks a task and start the *while-loop*, which calculates its finishing time by going through the intervals. Then we pick another task, and continue traversing the intervals at the point where we got interrupted by *for-loop*, and so on. We do not have any nested loops, and we always continue forward.

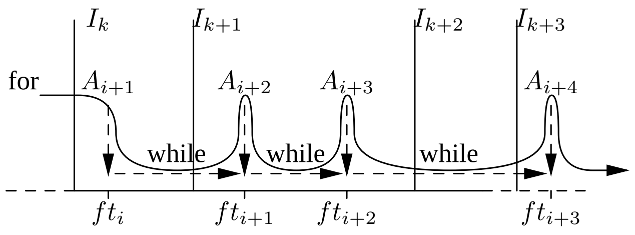


Figure 2.1: Online acceptance test for firm aperiodic tasks – algorithm complexity

2.3.6 Example

Assume an offline schedule with intervals and spare capacities as depicted in figure 2.2 (the shaded boxes represent offline tasks). Let $\mathcal{G}_3$ be

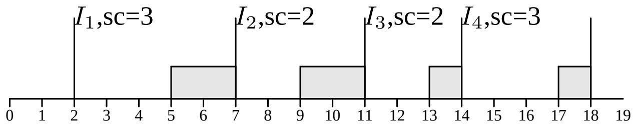


Figure 2.2: Example online firm aperiodic guarantee– static schedule

the set of previously guaranteed but not completed firm aperiodic tasks at current time $t = 3$:

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

where the first parameter is the remaining execution time, the second absolute deadline. Tasks is $\mathcal{G}_3$ are ordered by increasing deadlines. At time $t = 3$ we have the execution scenario of both offline scheduled tasks and guaranteed aperiodic tasks from $\mathcal{G}_3$ as described in figure 2.3. Guaranteed firm aperiodic tasks will execute in the first available slots, in EDF order.

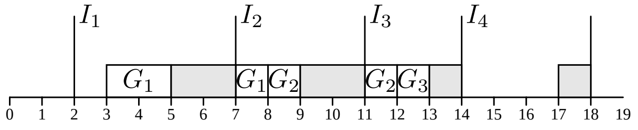
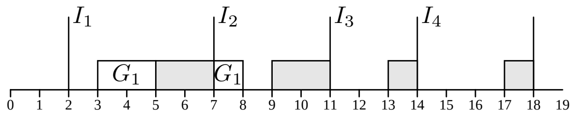


Figure 2.3: Example online firm aperiodic guarantee – execution without new task.

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

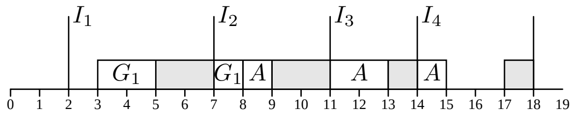
1. Task G_1 has earlier deadline than A, so G_1's position in the set $\mathcal{G}_3$ remains unchanged, i.e., before A. We do not need to guarantee it again.



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

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

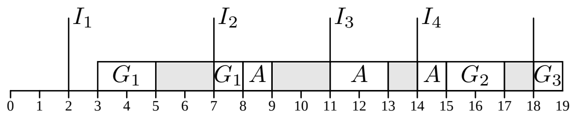
The finishing time is less than the deadline of A, which means that A could complete in time.



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

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

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



2.4 Improvements over original slot shifting

Here are the improvements over the original slot shifting acceptance test for a set of firm aperiodic tasks:

Implicit resource reservation

The method presented here reserves resources implicitly, by only accepting a new task if it can be guaranteed *together* with all previously guaranteed ones, and does not require runtime handling of resource reservation for guaranteed tasks, as the original slot shifting. Consequently, removal of guaranteed tasks and changes in the set of tasks can be handled efficiently.

Flexible rejection strategies

Our method allows for easy changes in the set of guaranteed tasks and thus supports flexible rejection strategies. It allows a new set of candidates to be submitted to the acceptance test and does not require modifications to the reserved resources for guaranteed tasks. In original slot shifting, if a firm aperiodic task cannot be feasibly included into the existing schedule, then the task is rejected. In our approach, we can chose to reject any of the guaranteed but not yet finished firm aperiodic tasks, and keep the currently guaranteed tasks instead.

Efficient resource reclaiming

Should aperiodic tasks use less resources than expressed in worst case parameters, our method directly reclaims these without recalculation of available resources. The next time the acceptance test is performed, the fact that a task has an earlier finishing time is considered in the calculations by simply starting the calculation of the finishing point of the currently guaranteed task earlier, i.e., we start to ”fill up” remaining execution time for the currently guaranteed task at an earlier point in time.

This also results in an earlier finishing time of the current task, leaving more space for other executions.

Improved overload handling

Overload handling schemes can easily be applied, since our method provide a set of firm aperiodic tasks from which any of the tasks can be rejected. One such overload handling method based on our algorithm has been presented in [17]. Even other overload handling schemes, such as presented in [14] and [3] can be used.

2.5 Simulation Analysis

We have implemented the algorithms for firm aperiodic tasks described above and have run simulations for various scenarios. We have studied the guarantee ratio for aperiodic tasks for different combinations of total system loads and aperiodic deadlines.

For the purpose of simulations we have developed a simulator to provide for detailed analysis of slot shifting. We also implemented a debugger, which provides for visual monitoring of the data structures during the simulations.

Here we present some key results, see appendix A for details about the simulation setup, performed experiments and confidence intervals. Also see appendix C for the information about the tools that we implemented for the simulation purpose.

2.5.1 Simulation setup

We have randomly generated offline and aperiodic task loads, so that the combined load of both periodic and aperiodic tasks was set between 10% and 100%. The deadlines for the aperiodic tasks were set to their maximum execution time, MAXT, two times MAXT and three times MAXT. We studied the guarantee ratio for the randomly arriving aperiodic tasks.

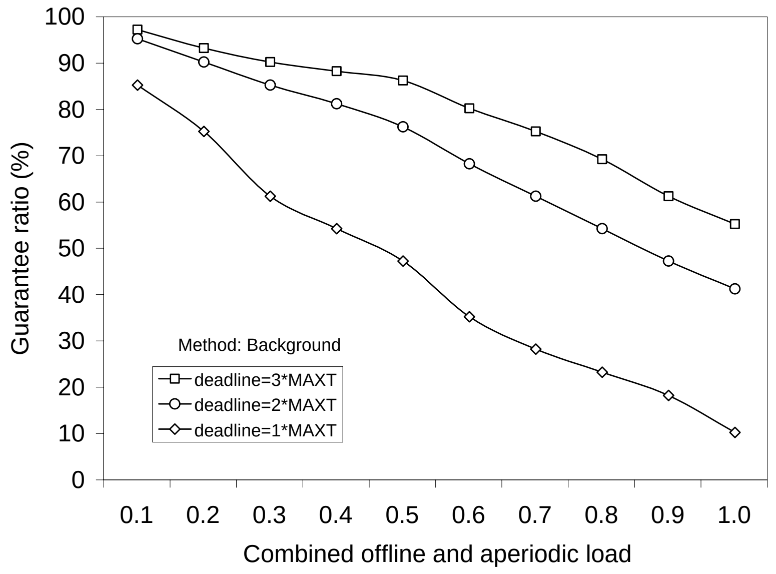


Figure 2.4: Guarantee ratio for aperiodic tasks – Background scheduling

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

2.5.2 Results

Figure 2.4 illustrates the performance of background scheduling for three different deadline settings of aperiodic tasks, while figure 2.5 depicts the performance of the extended slot shifting approach. Each point in the graphs represents a sample size of 800-3000 simulation runs, with different combinations of periodic and aperiodic tasks. 0.95 confidence intervals were smaller than 5%.

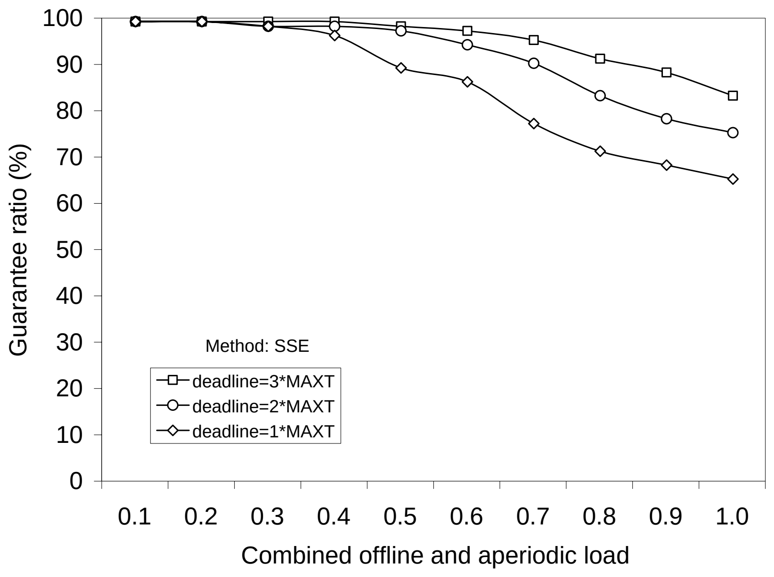


Figure 2.5: Guarantee ratio for aperiodic tasks – SSE (our approach)

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

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

2.6 Chapter summary

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

First, a standard offline scheduler constructs a schedule, resolving complex task constraints such as precedence, distribution, and end-to-end deadlines. Then, the offline schedule is analyzed for unused resources, i.e., intervals and spare capacities are calculated. Offline scheduled task can flexibly execute within their corresponding intervals. It is also possible for an offline task to executes outside its interval, i.e., earlier than the interval start, but not before its earliest start time.

The run-time scheduler uses this information to handle aperiodic tasks, shifting the execution of offline scheduled tasks to reduce response times without affecting feasibility. We provide an $O(N)$ acceptance test for a set of aperiodic tasks on a top of the offline schedule and guarantee tasks without explicit reservation of resources. Compared to the original slot shifting approach, our method supports more flexible, value based selections of tasks to reject or remove in overload situations, and simple resource reclaiming. Simulation results illustrate the effectiveness of the algorithm.

In the next chapter, we will extend this approach to handle sporadic tasks.

Chapter 3

Sporadic Task Handling

Sporadic tasks are suitable for handling events that arrive at the system at arbitrary points in time, but with defined maximum frequency. We showed in previous chapter how aperiodic tasks can be guaranteed and scheduled together with offline, periodic tasks. Here we extend that approach to handle sporadic tasks as well.

Offline scheduling is not suitable for handling sporadic tasks due to the unknown arrival times. For the same reason, online handling only can be computationally expensive. We use a combined offline and online approach. Offline we assume the worst-case scenario for arrival patterns for sporadic tasks, and online we try to reduce this pessimism by using the current information about the system. Dynamic activates are accommodated without affecting the feasible execution of offline scheduled tasks. As a final result, we provide a combined offline and online method to deal with mixed periodic, aperiodic and sporadic task sets with simple and complex constraints.

We start by presenting the basic idea for sporadic task handling in section 3.1. In section 3.2 we show how a set of sporadic tasks can be guaranteed offline for the worst-case scenario, followed by an online method to reduce this pessimism at runtime, presented in section 3.3. Section 3.4 describes the simulation results for our method, followed by the chapter conclusions in section 3.5.

3.1 Basic Idea for Sporadic Task Handling

We present a combined offline and online approach for handling sporadic tasks. Offline we assume the worst-case arrival scenario for the sporadic tasks and guarantee their feasible execution once they start invoking at runtime. Online, we reduce the pessimism assumed at design time by taking advantage of the current information about the system.

Offline part

The offline transformation of slot shifting, described in previous chapter, determines resource usage and distribution as well, which we use to handle sporadic tasks. The sporadic tasks are guaranteed offline, during design time, which allows rescheduling or redesign in the case of failure.

An offline test determines and allocates resources for sporadic tasks such that worst-case arrivals can be accommodated at any time. Since we do not know the arrival pattern of the sporadic set, we guarantee them for the worst-case, i.e., we assume all sporadic are released at the same time and with the maximum frequency between consecutive invocation.

Online part

Assuming the worst-case scenario for sporadic arrivals at design time is a necessary, but too pessimistic assumption. At runtime, we try to reduce this pessimism by using the current knowledge about the system, e.g., when a sporadic task arrives, we know that the next invocation will not occur at least for the period of its minimum inter-arrival time, which we use for firm aperiodic tasks guarantee.

An online algorithm keeps track of arrivals of instances of sporadic tasks to reduce pessimism about future sporadic arrivals and improve response times and acceptance of firm aperiodic tasks. If a sporadic task invokes its instances with less frequency than the worst-case one, then we can easily reclaim its reserved resources for other dynamic activities, i.e., firm and soft aperiodic tasks.

3.2 Offline Feasibility Test

Here we introduce an offline guarantee algorithm for a set of sporadic tasks. Firstly, the off-line periodic schedule is created and analyzed for intervals and spare capacities of slot shifting, as described in chapter 2.2.1. Secondly, the set of sporadic tasks is tried to fit into the periodic schedule. If the sporadic set is not accepted, it is up to designer to re-design the system, i.e., reschedule periodic tasks or change the sporadic set.

3.2.1 Sporadic task set

All tasks in the sporadic set are assumed to be invoked with their maximum frequency, creating the worst case scenario for the scheduler. If the deadline of a sporadic task can be guaranteed for the release with its maximum frequency, then all subsequent deadlines are guaranteed. Examples of this approach are given in [4].

The minimum time difference between successive releases of a sporadic task is its minimum inter-arrival time. It has been shown in [13] that a sporadic task which is released with its maximum frequency behaves exactly like a periodic task with period equal to its minimum inter-arrival time.

Now that we know the deadline, the maximum execution time and the 'period' of each sporadic task in the set, we can use that information to perform an offline guarantee test on the set for its worst load pattern.

3.2.2 Critical slots

One way of investigating if the sporadic set fits into the periodic schedule is to investigate if it fits at each time slot of the periodic schedule, but this is impractical. It is sufficient to investigate only some selected points in time, called *critical slots*.

Definition 1. *The critical slot, t_c, of an interval I is the time slot in I such that if a dynamic task arrives at t_c, its execution will be maximally delayed, compared to all other slots in I.*

Critical slot t_c for an interval I is calculated as:

$$t_c(I) = start(I) + max(sc(I), 0) \tag{3.1}$$

Figure 3.1 gives an example of a critical slot.

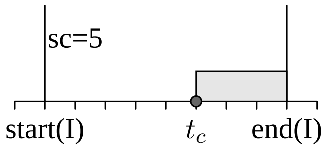


Figure 3.1: Example of a critical slot.

Critical slot implies that the execution of sporadic tasks S_i will be maximally delayed by the execution of the offline scheduled tasks, if S_i starts to invoke its instances at a critical slot of a certain interval. We prove that if the sporadic set can be guaranteed at the critical slot, it will be guaranteed at every other slot within the same interval.

Theorem 1. *Let S_i denote a sporadic task and S_i^k the k^{th} instance of S_i. If S_i^k can be guaranteed at the critical slot t_c of an interval I, it will also be guaranteed at any other slot t within the same interval:*

$$\forall t \in I, t \neq t_c : \quad S_i^k \; guaranteed\,at\,t_c \Rightarrow S_i^k \; guaranteed\,at\,t$$

Proof. Here is the proof by contradiction. Assume the following is correct:

Assumption 1: There is a time slot t in interval I, other than the critical slot t_c, such that S_i^k can be guaranteed at t_c, but not at t:

$$\exists t \in I, t \neq t_c : \quad (S_i^k \; guaranteed\,at\,t_c) \wedge (S_i^k \neg guaranteed\,at\,t)$$

When S_i^k arrives, there will be a certain amount of spare capacity available for it between its arrival time and its deadline. Let δ denote the difference between spare capacities available for S_i^k if it arrives at t and if it arrives at t_c. Assumption 1 states that S_i can be guaranteed at t_c

but not at t, which means that the amount of spare capacity available for S_i^k if it arrives at t_c must be larger than the amount of spare capacity available if it arrives at t. This implies, if assumption 1 holds, δ must be negative:

$$\delta < 0 \tag{3.2}$$

Assume S_i^k arrives at t, i.e., $ar(S_i^k) = t$. There are two possibilities for arrival of S_i^k: before or after the critical slot t_c.

Case 1 : $t > t_c$, S_i^k arrives after t_c, as depicted in figure 3.2 (shaded box in the figure represents scheduled or guaranteed tasks).

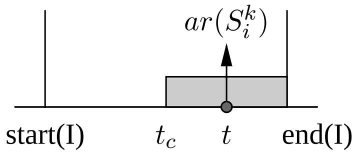


Figure 3.2: Sporadic arrival after critical slot.

The requirement for S_i^k to be accepted is that the spare capacity available for it at its arrival time has to be greater or equal to the maximum execution time of S_i.

Let I_{ar} be the interval in which S_i^k arrives and I_{dl} the interval in which S_i^k has its deadline. If S_i^k arrives at t_c instead of t, the amount of spare capacity available in I_{ar} and I_{dl} will change. Let α and β denote this change:

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

This gives:

$$\delta = \alpha + \beta \tag{3.3}$$

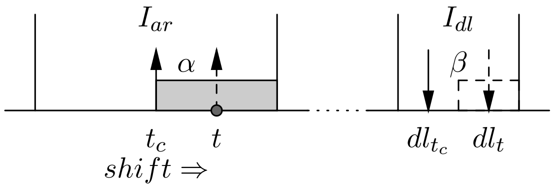


Figure 3.3: Sporadic arrival shifted to the right.

This is illustrated in figure 3.3.

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

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

$$\delta = \alpha + \beta > 0, (\alpha = 0, \beta > 0) \tag{3.4}$$

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

$$\delta = \alpha + \beta = 0, (\alpha = 0, \beta = 0) \tag{3.5}$$

(4) and (5) give:

$$\delta \geq 0$$

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

Case 2 : $t < t_c$, S_i^k arrives before t_c, as depicted in figure 3.4.

Now we shift the arrival time of S_i^k to the left, that is before the critical point t_c. This is shown in figure 3.5.

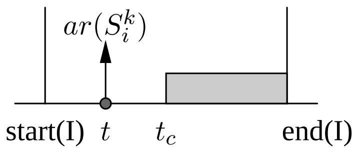


Figure 3.4: Sporadic arrival before critical slot.

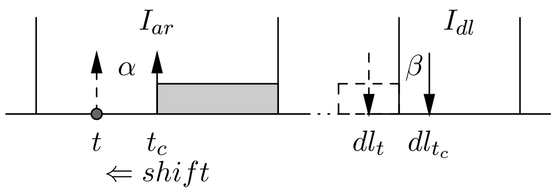


Figure 3.5: Sporadic arrival shifted to the left.

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

$$\delta = \alpha + \beta = \alpha + (-\alpha) = 0, (\beta = \beta_{worst}) \qquad (3.6)$$

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

$$\delta = \alpha + \beta > 0, (|\beta| < \alpha) \qquad (3.7)$$

(6) and (7) implies:

$$\delta \geq 0$$

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

$\square$

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

3.2.3 Offline feasibility test for sporadic tasks

The feasibility test for the set of sporadic tasks works by creating a worst case load demand of the sporadic tasks as described in section 3.2.1. We assume that all sporadic tasks arrive with their maximum frequency and test if the demand created can be accommodated into the static schedule at all critical slots.

Algorithm description

Here follows the pseudo-code for the guarantee algorithm for a set of sporadic tasks $\mathcal{S}$ (read the comment below in parallel):

Let:

sc_a	= available sc for S_i^k from $ar(S_i^k)$ to $dl(S_i^k)$
R	= an array containing slots reserved for previously guaranteed sporadic tasks
$initR()$	= initiates R to empty set
$countR(x, y)$	= number of reserved slots between slots x and y
$reserve(x, y)$	= reserves x slots as close to y as possible

```
1. $\forall t_c$
2.     $initR()$
3.     $\forall S_i^k \in \mathcal{S}$
4.         $sc_a(S_i^k) = \sum_{I_j \in [end(I_{ar}), start(I_{dl})]} max(sc(I_j), 0)$
```

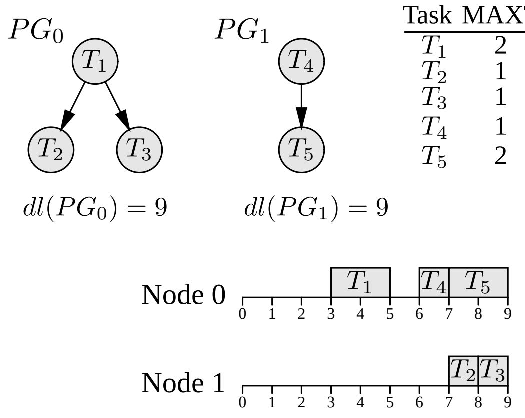
```
5.                    $+min(sc(I_{dl}), dl(S_i^k) - start(I_{dl}))$
6.                    $-countR(ar(S_i^k), dl(S_i^k))$
7.         $if \quad (sc_a(S_i^k) \geq c(S_i))$
8.         $then \quad reserve(c(S_i), dl(S_i^k))$
9.         $else$   abort (set rejected)
```

Comments:

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

Example

Assume the following periodic tasks with maximum execution times (MAXT), deadline (dl) and precedence constraints as described in figure



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

Figure 3.6: Example offline sporadic guarantee – periodic tasks and offline schedule.

3.6. We use a distributed real-time system with two computing nodes to make the example more general.

We calculate intervals and spare capacities, as described in chapter 2.2, and critical slots as described in section 3.2.2:

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

Intervals with their assigned tasks and critical slots are depicted in figure 3.7.

Assume a sporadic set $\mathcal{S} = \{S_1(1,5), S_2(3,10)\}$ where the first parameter is maximum execution time and the other one the minimum inter-arrival time at node 0. If we assume that sporadic tasks arrive with their maximum frequencies, then the deadline of each invocation is equal to the release of the next invocation.

Now we apply the off-line guarantee algorithm on each task in the sporadic set $\mathcal{S}$. First, we try to guarantee S_1 and S_2 at critical slot 3,

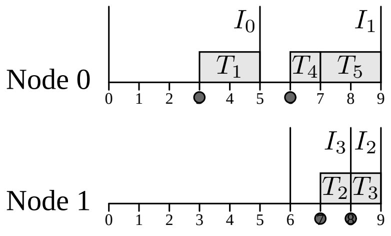


Figure 3.7: Example offline sporadic guarantee – schedule with intervals.

and if they can be guaranteed, we proceed with investigation of slot 6. The LCM of $\mathcal{S}$ is 10, which means that S_1 is invoked twice and S_2 once before the worst-case pattern is repeated.

We now illustrate the guarantee test for sporadic tasks S_1 and S_2 in figure 3.8.

t_c	Task	Invocation	$sc_a \leq MAXT$?	R
3	S_1	1	$1 \geq 1 \Rightarrow true$	$\{5\}$
		2	$3 \geq 1 \Rightarrow true$	$\{5,11\}$
	S_2	1	$2 \geq 3 \Rightarrow false$	*abort!*

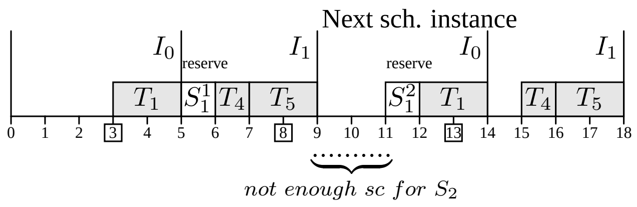

Next sch. instance
I_0
I_1
I_0
I_1
reserve
reserve
T_1
S_1^1
T_4
T_5
S_1^2
T_1
T_4
T_5
0 1 2 $\boxed{3}$ 4 5 6 7 $\boxed{8}$ 9 10 11 12 $\boxed{13}$ 14 15 16 17 18
$not\ enough\ sc\ for\ S_2$

Figure 3.8: Example failed offline sporadic guarantee.

We start by checking the first instance of S_1, i.e., S_1^1. The amount of available spare capacities before the deadline of S_1^1 is 1, which is enough to execute it. Event second instance of S_1 can be guaranteed since the available spare capacity is greater than the execution demand of S_1. Hence, we reserve slots 5 and 11 for the instances of S_1. However, the first instance of S_2 cannot be guaranteed since the available spare capacity is less than its maximum execution time, hence the sporadic set cannot be guaranteed at critical slot 3.

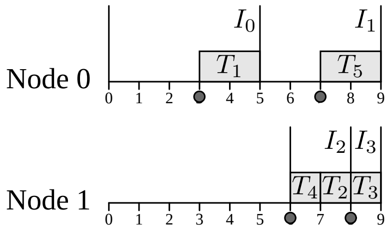


Figure 3.9: Example offline sporadic guarantee – schedule redesign.

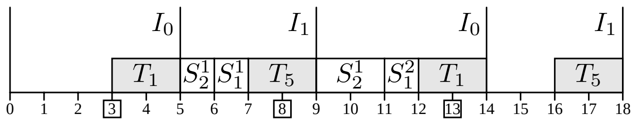
What we can do now is to redesign the system and try again. Since we support distributed systems, we could reallocate some of the periodic tasks from node 0 to node 1, or allocate some of sporadics on node 1. In this example, we decide to schedule the periodic task T_4 on node 1 instead of node 0. The new offline schedule is depicted in figure 3.9.

Intervals remain the same, spare capacities and critical slots have to be recalculated for I_1 and I_2:

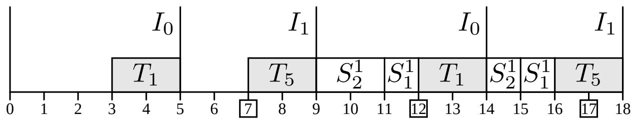
I_1 : $sc(I_1)$ = 1+1=2 I_2 : $sc(I_2)$ = 1-1=0
$t_c(I_1)$ = 5+2=7 $t_c(I_2)$ = 6+0=6

We try to guarantee $\mathcal{S}$ on node 0 again, in recalculated critical slots. This time both S_1 and S_2 are guaranteed, see figure 3.10.

t_c	Task	Invocation	$sc_a \geq MAXT?$	R
3	S_1	1	$2 \geq 1 \Rightarrow true$	$\{6\}$
		2	$3 \geq 1 \Rightarrow true$	$\{6,11\}$
	S_2	1	$3 \geq 3 \Rightarrow true$	$\{5,6,9,10,11\}$
7	S_1	1	$3 \geq 1 \Rightarrow true$	$\{11\}$
		2	$2 \geq 1 \Rightarrow true$	$\{11,15\}$
	S_2	1	$3 \geq 3 \Rightarrow true$	$\{9,10,11,14,15\}$



a) Critical slot 3



b) Critical slot 7

Figure 3.10: Example successful offline sporadic guarantee.

3.3 Online Handling

We showed above how a sporadic set can be guaranteed offline. In this section we show how this set can be scheduled online together with periodic and aperiodic tasks. The algorithm presented here keeps track of sporadic arrivals and reduces pessimism introduces by the worst-case assumption in the offline phase, something that improves guarantees and response times of aperiodic tasks.

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

3.3.1 Acceptance test for aperiodic tasks in presence of sporadic tasks

In section 2.3.3 we provided a method to guarantee firm aperiodic tasks on top of an offline schedule. Now we will see how to perform the same guarantee for firm aperiodic tasks in the presence of offline guaranteed sporadic tasks.

Interference window

When guaranteeing a firm aperiodic task A_j, we need to take into consideration the preemptions from offline guaranteed sporadic tasks that can execute their instances between the arrival time and the deadline of A_j. The time interval in which a sporadic task S_i can preempt and hence interfere with the execution of an aperiodic task A_j is called the *interference window* of A_j by S_i and it is denoted as $IW(A_j, S_i)$. As the first step of the aperiodic acceptance test we need to determine this interval for each sporadic task in the offline guaranteed sporadic set $\mathcal{S}$.

We do not know when a sporadic task $S_i \in \mathcal{S}$ will start to invoke its instances, but once it starts, we do know the minimum time between its invocations – the minimum inter-arrival time of S_i. We also know the worst case execution time of S_i, $c(S_i)$. We use this information for the acceptance test of A_j.

Assume S_i invokes an instance at time t (see figure 3.11). Let S_i^k denote current invocation of S_i, and S_i^{k+1} the successive one. At time t we know that S_i^{k+1} will arrive no sooner than $t+\lambda$, where λ is the minimum inter-arrival time of S_j. So, when S_i^k has finished its execution, S_i will not interfere with any of the firm aperiodic tasks until S_i^{k+1} arrives, which is at least λ time units. This means, when calculating the amount of resources available for a firm aperiodic task A_j with an execution that intersects with S_i's execution window, we do not need to take into account the interference from S_i at least between the finishing time of its current invocation, S_i^k, and the start time on the next invocation, S_i^{k+1}, as depicted in figure 3.11.

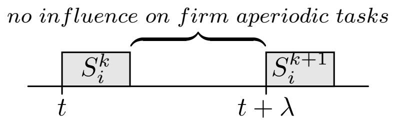


Figure 3.11: A sporadic task.

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

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

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

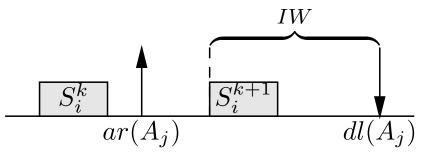
Assume A_j arrives at system at time t, i.e., $ar(A_j) = t$. Let S_i^k the last invocation of S_i before t. There are two cases to consider when calculating the interference window $IW(A_j, S_i)$:

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

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

a) $start(S_i^k) + \lambda \leq t$, i.e., the last invocation completed before A_j arrived, and the next invocation, S_i^{k+1}, could have arrived but it has not yet. This means S_i^{k+1} can enter A_j's execution window at any time, thus the same as in case 1: $IW(A_j, S_i) = EW(A_j)$.

b) $end(S_i^k) \leq t < start(S_i^k) + \lambda$, i.e., the current invocation S_i^k has completed before t, and the next one has not arrived yet. But now we know that the next one, S_i^{k+1} will not arrive until λ time slots, counted from the start time of S_i^k.



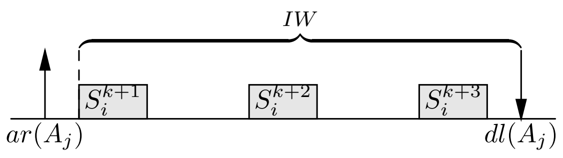
This means the interference window can be decreased with the amount of time slots in EW for which we know that S_i^{k+1} cannot possibly arrive:

$$IW(A_j, S_i) = [start(S_i^k) + \lambda, dl(A_j)]$$

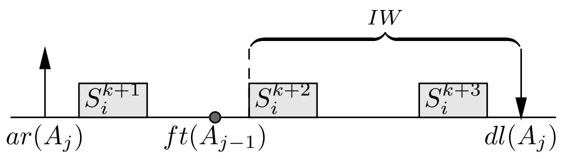
c) $t < end(S_i^k)$, i.e., the current invocation is still executing. In the worst case, the interference window is entire EW, $IW = EW$.

Now we will see how the interference window can actually "shrunk" when guaranteeing a firm aperiodic task A under runtime. It is usually not the case that A will start to execute as soon it arrives. This because of the offline tasks and previously guaranteed firm aperiodic tasks. In section 2.3.3, we presented a method for guaranteeing firm aperiodic tasks on top of offline tasks. The start time of the firm aperiodic task A_j, which is currently tested for acceptance, is based on the finishing time of its predecessor, A_{j-1}, i.e., another firm aperiodic task with earlier deadline. Hence, in some cases the start of the interference window $IW(A_j, S_i)$ is set to the finishing time of A_{j-1}.

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

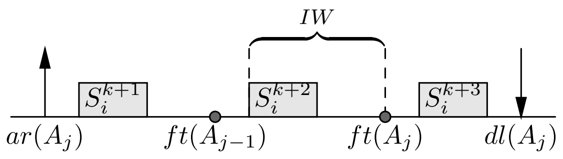


Assume another previously accepted firm aperiodic task A_{j-1} which will delay the execution of A_j:



We see that the earliest time A_j can start is set to the finishing time of its predecessor, $ft(A_{j-1})$. So, all invocations of S_i that occurred before earliest start time of A_j, $est(A_j)$, have been taken care of when calculating $ft(A_{j-1})$, and are not needed to be considered when calculating $ft(A_j)$. The start of the interference window is now set to the start time of the first possible instance of S_i that can interfere with A_j, that is S_i^{k+2}.

Now we calculate the finishing time of A_j using the algorithm described in section 2.3.3. Without sporadic tasks, A_j would be guaranteed to finish at time $ft(A_j)$. Since A is guaranteed to finish before its deadline, we do not need to take into consideration the impact from S_i after the finishing time of A_j. Hence, the end of the interference window $IW(A_j, S_i))$ is set to $ft(A_j)$.



So, what actually happens in this example is that only one instance of S_i is considered when calculating $ft(A_j)$.

At this point, we can formalize the impact of a sporadic task S_i on a firm aperiodic task A_j:

If S_i has not yet started to invoke its instances at the time we start with the acceptance test for A_j, we must assume the worst case, that is the first instance of S_i will start at the same time as the earliest start time of A_i:

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

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

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

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

To conclude, the time interval $IW(A_j, S_i)$ in which a sporadic task S_i may preempt and interfere with the execution of a firm aperiodic task A_j is obtained as:

$$IW_i = [\delta, ft(A_j)] \tag{3.8}$$

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

$$\delta = \begin{cases} est(S_i^{k+m}) & \text{if } S_i \text{ known} \\ max(t, ft(A_{j-1})) & \text{otherwise} \end{cases} \tag{3.9}$$

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

The processor demand approach, [8], can be used to determine the total processing time, $c^T(\mathcal{S})$, needed for all sporadic tasks in $\mathcal{S}$ that will interfere with A_j:

$$c^T(S) = \sum_{i=1}^{n} \left\lfloor \frac{|IW(A_j, S_i)|}{\lambda(S_i)} \right\rfloor c(S_i) \tag{3.10}$$

Algorithm description

Assume a firm aperiodic task A_i that is tested for acceptance upon its arrival time, current time t. We want to make sure that A_i will complete before its deadline, without causing any of the guaranteed tasks to miss its deadline. A guaranteed task is either an offline task, a previously guaranteed firm aperiodic task or a sporadic task. Offline and sporadic tasks are guaranteed before the run-time of the system, see sections 2.2 and 3.2, while firm aperiodic tasks are guaranteed online, upon their arrival. The guarantee algorithm is performed as follows:

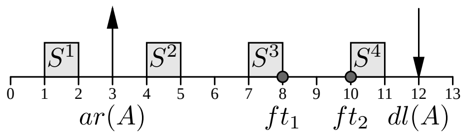
step 1: Assume no sporadic tasks and calculate the finishing time of A_i based only on offline tasks and previously guaranteed firm aperiodic tasks (as described in section 2.3.3).

step 2: Calculate the impact from all sporadic tasks that could preempt A_i before its finishing time calculated in the previous step (equation 3.10).

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

Example

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



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

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

Implementation

The first part of the algorithm is the same as described in 2.3.2: first we locate the position of hard aperiodic task to be guaranteed, calculate its finishing time and check if any of previously guaranteed hard aperiodic tasks will miss its deadline. The second part, that calculates the finishing time, is extended to handle the impact from the sporadic tasks as follows:

```
$getFinishingTime(ft_{pred}, c)$
    /*"fill up" free slots until the $c$ is exhausted.*/
    $\forall S_i \in \mathcal{S}$
            if $S_i$ started to invoke
                $\delta = est(S_i^{k+m})$                /*eq (3.9)*/
            else
                $\delta = max(t, ft_{pred})$
            $IW_i = [\delta, ft]$                        /*eq (3.8)*/
            $sum = sum + \left\lfloor \frac{|IW_i|}{\lambda(S_i)} \right\rfloor c(S_i)$   /*eq (3.10)*/
    if $sum \neq 0$
            $getFinishingTime(ft, sum)$
    else
            return $ft$
```

The recursive formulation was chosen for simplicity of explanation: our implementation uses a loop. In the loop, time is increased from current to finish time, without going back. Thus the complexity remains linear, similar to the finishing time algorithm in 2.3.2.

3.4 Simulation Analysis

We have tested the acceptance ratio for firm aperiodic tasks with the methods to handle sporadic tasks: worst case arrivals without knowledge about sporadic invocations (refered as "no info") and updated worst

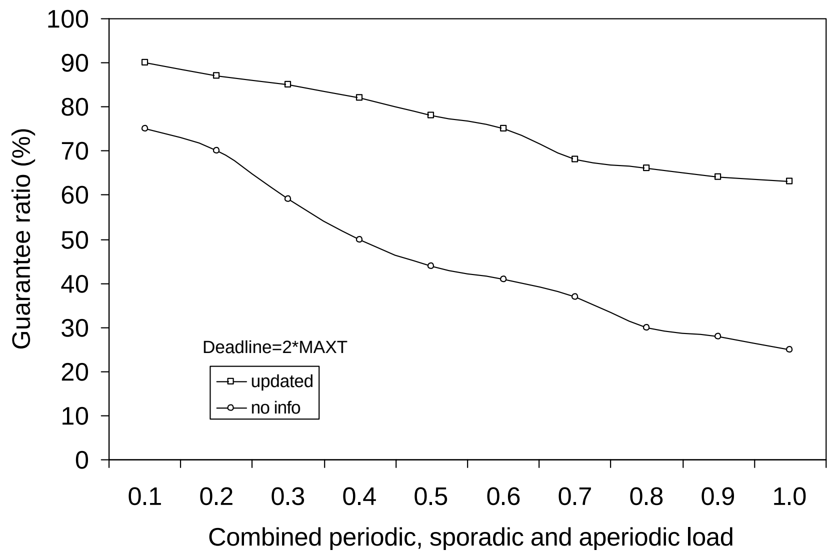


Figure 3.12: Guarantee ratio for aperiodic tasks in the presence of sporadics tasks: load variation

case with arrival info ("updated"), as described in section 3.3, case 1 and case 2.

As in the previous chapter, here we present only some key results, see appendix A and C for details about performed experiments and implemented simulation tools.

3.4.1 Simulation setup

We studied the guarantee ratio of randomly arriving aperiodic tasks under randomly generated arrival patterns for the sporadic tasks. First we investigated the guarantee ratio for firm aperiodic tasks with combined loads 10% - 100%. The deadlines for the aperiodic tasks were varyed between their maximumum execution time and three times the maximum execution time, i.e., between MAXT and 3*MAXT. The combined

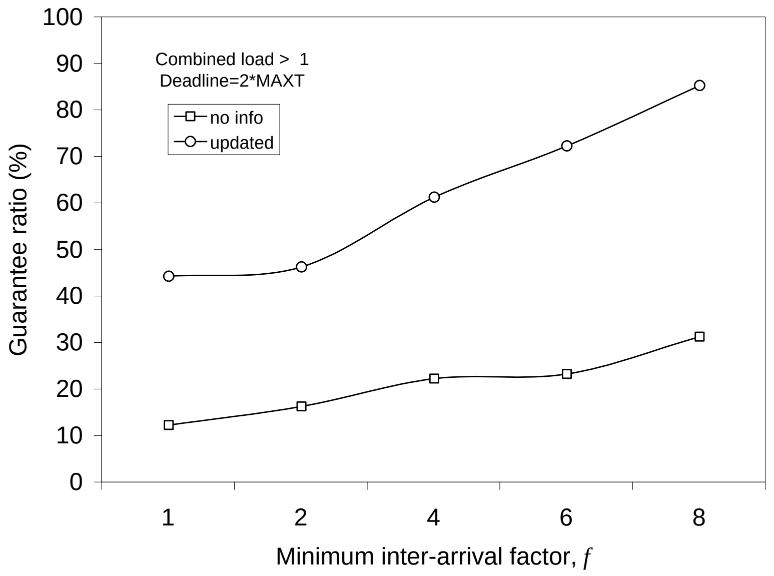


Figure 3.13: Guarantee ratio for aperiodic tasks in the presence of sporadics tasks: variation of MINT

load was set to 100%.

In the second part of the experiment we varied the arrival frequencies of sporadic tasks according to a factor, f, such that the separation between instances $averageMINT$ is equal to $averageMINT = f * MINT$. This means that if $f = 1$ then the instances are invoked with the maximum frequency, and if $f = 2$, the distance between two consecutive invocations is $2 * MINT$ on average.

3.4.2 Results

The results from the first part of the experiment are summarized in figure 3.12, while the results from the second one are presented in figures 3.13. In both cases our method improves the acceptance ratio of firm aperiodic

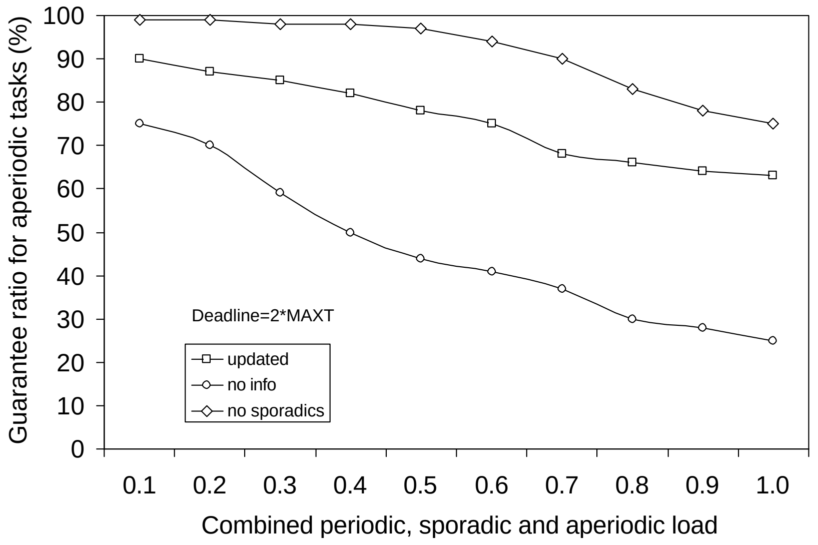


Figure 3.14: Guarantee ratio for aperiodic tasks with sporadics - Final results

tasks. This results from the fact that our methods reduce pessimism about sporadic arrivals by keeping track of them.

Figure 3.14 summarizes the simulation. We can see that guarantee ratio for firm aperiodic tasks is very high, even when we have sporadic tasks in the system. By keeping track off sporadic arrivals, we can accept firm tasks that otherwise would be rejected.

3.5 Chapter summary

In this chapter we presented a method for integrated offline and online scheduling of mixed sets of tasks and constraints. In particular, we presented an efficient method to handle sporadic tasks, providing for $O(N)$ complexity online acceptance test for firm aperiodic tasks.

During offline analysis we determine the amount and location of unused resources, which we use to include dynamic activities during the runtime of the system. The sporadic tasks are guaranteed during design time, allowing rescheduling or redesign in the case of failure. At runtime, resources reserved for sporadic tasks can be reclaimed and used for efficient aperiodic task handling.

Thus, our method combines handling of complex constraints, efficient and flexible runtime scheduling, as well as offline and online scheduling, providing a basis for predictably flexible real-time systems. Results of simulation study show the effectiveness of the algorithms.

In the second part of the thesis, we will use the scheduling and resource reservation mechanism presented here to flexibly schedule media processing in resource constrained systems.

– PART II –

Real-Time Processing of MPEG-2 Video in Resource Constrained Systems

Chapter 4

MPEG-2 Video Processing under Limited Resources

Media files are very large in size in their original form, thus, they must be compressed before being stored on e.g., a DVD, or transmitted through a network, e.g., the Internet. MPEG-2 is the most popular compression techniques for digital video and audio today, widely used in consumer electronics for DVD players, digital satellite receivers, and TVs today.

In this chapter we give an overview of MPEG-2 and set up a processing model for handling MPEG-2 video streams that is going to be used in the rest of this thesis. In particular, we give a detailed description of MPEG-2 video stream, and show how it is processed. Furthermore, we extend the work by Liesbeth Steffens[1] presented in [56] to identify buffer and latency requirements for continuous MPEG-2 playout, the work that has been published in our joint papers with her [36, 35, 37].

We start by a giving a description of MPEG-2 video stream in section 4.1. Here we discuss different MPEG-2 video layers, coding techniques for the frames and the stream organization. In section 4.2 we present the task model needed for video processing, followed by the

[1]Liesbeth Steffens from Philips Research The Netherlands, Eindhoven, has co-authored three of my MPEG publications. However, any figures on buffers and latency done by Liesbeth are not included in this thesis. We refer to our joint papers.

latency and buffer requirements in section 4.3. We discuss the different techniques for video processing under limited system resources in section 4.4. Section 4.5 summarizes the chapter.

4.1 MPEG-2 Video Stream

Here we present the main characteristics of MPEG-2 video stream. A complete description of the MPEG-2 compression scheme is beyond the scope of this thesis. For details on MPEG see e.g., [1, 66, 57].

4.1.1 MPEG-2 video layers

An MPEG-2 video stream is a sequence of compressed frame pictures. Henceforth we will use the terms *picture* and *frame* interchangeably.

MPEG-2 video is broken up into a hierarchy of layers to help with error handling, random search and editing, and synchronization. The layers are depicted in figure 4.1. From the top, the first layer is known as the *video sequence layer*, and it is any self-contained video bit-stream, e.g., a part of a movie. A sequence layer begins with a sequence header, which contains the information about the picture size (width and height), overall display aspect ratio (for example, 4:3 for regular TV, or 16:9 for widescreen), the intended display rate (e.g., 24 frames per second) and the stream bit-rate.

Each sequence consists of one or more *groups of pictures*, which consist of a header and a series of several pictures, and it is intended to allow random access into the sequence.

A *picture* is the primary coding unit of a video sequence. It consist of *slices*. Slices are important for error handling. If the bit stream contains an error, the decoder can skip to the start of the next slice. Having more slices in the bit stream allows better error handling, but use space that could otherwise be used to improve picture quality.

Each slice consists of one or more adjacent *macroblocks*, which are 16x16 arrays of luminance pixels, or picture data elements, grouped in

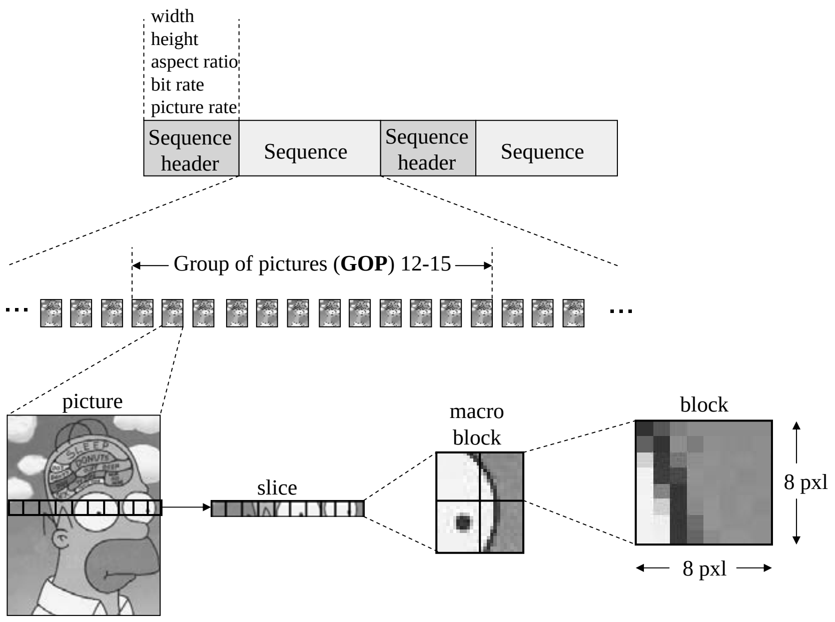


Figure 4.1: MPEG-2 video layers

four 8x8 *blocks*, for further processing such as transform coding.

4.1.2 Frame types

Some frames are encoded with pure intra-picture compression techniques, i.e., the picture can be reconstructed from the frame itself only. Other frames are encoded using motion compensation, which is an inter-picture technique. Instead of the complete picture, only the differences with one or two nearby pictures are encoded using the intra-picture techniques. In other words, macroblocks in a frame can be coded as intra and non-intra, i.e., there are intra, forward-predicted, backward-predicted and forward-and-backward predicted macroblocks in a MPEG bit stream.

To decode frames that are encoded with motion compensation, the nearby frames that were used in the encoding, have to be available for reference. These frames are called *reference* frames. We distinguish *forward* references and *backward* references, to past and future frames, i.e., frames that are displayed earlier and later, respectively.

The MPEG-2 standard defines three types of frames, I, P and B frames.

I frames

The I frames or *intra* frames are simply frames coded as still images. They contain absolute picture data and are self-contained, meaning that they require no additional information for decoding. I frames have only spatial redundancy providing the least compression among all frame types. Therefore they are not transmitted more frequently than necessary.

P frames

The second kind of frames are P or *predicted* frames. They are forward predicted from the most recently reconstructed I or P frame, i.e., they contain a set of instructions to convert the previous picture into the current one. P frames are not self-contained, i.e., if the previous reference frame is lost, decoding is impossible. On average, P frames require roughly half the data of an I frame, but our analysis in chapter 6 also showed that this is not the case for a significant number of cases.

B frames

The third type is B or *bi-directionally* predicted frames. They use both forward and backward prediction, i.e., decoding a B frame requires previous I or P frame, and *next* I or P frame, see figure 4.2. Forward and backward references are always to the nearest I or P picture in the intended direction. B frames contain vectors describing where in an earlier or a later picture data should be taken from. They also contain transformation coefficients that provide the correction. B frames are never predicted from each other, only from I or P frames. As a consequence, no other frames depend on B frames.

B frames require resource-intensive compression techniques but they also exhibit the highest compression ratio, on average typically requiring one quarter of the data of an I picture. Once again, our analysis showed that this does not hold for a significant number of cases.

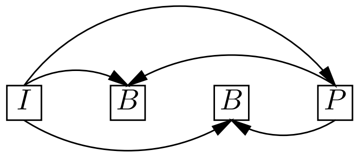


Figure 4.2: Forward (P) and bidirectional (B) prediction

An encoded video stream can consist of I frames only, of I and P frames, or of I, P, and B frames. In our work we focus on the last category, streams consisting of I, P and B frames.

4.1.3 Group of Pictures

Predictive coding, i.e., the current frame is predicted from the previous one, cannot be used indefinitely, as it is prone to error propagation. A further problem is that it becomes impossible to decode the transmission if reception begins part-way through. In real video signals, cuts or edits can be present across which there is little redundancy. In the absence of redundancy over a cut, there is nothing to be done but to send from time to time a new reference picture information in absolute form, i.e., an I frame. As I decoding needs no previous frame, decoding can begin at I coded information, for example, allowing the viewer to switch channels.

An I frame, together with all of the frames before the next I frame, form a *Group of Pictures (GOP)*, as shown in figure 4.2. The GOP length is flexible, but 12 or 15 frames is a common value. Furthermore, it is common industrial practice to have a fixed pattern (e.g., $I\ BB\ P\ BB\ P\ BB\ P\ BB$). However, more advanced encoders will attempt to optimize the placement of the three frame types according to local sequence characteristics in the context of more global characteristics.

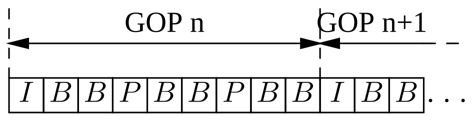


Figure 4.3: Frame types and Group of Pictures

Note that the last B frame in a GOP requires the I frame in the next GOP for decoding and so the GOPs are not truly independent. Such GOPs are called *open* GOPs. Independence can be obtained by creating a *closed* GOPs which may contain B frames but end with a P frame. In a closed GOP, all references are within the GOP, because the GOP starts and ends with a reference frame.

4.1.4 Decoding and display order

As mentioned above, B frames are predicted from two I or P frames, one in the past and one in the future. Clearly, information in the future has yet to be transmitted and so is not normally available to the decoder. MPEG-2 gets around the problem by sending and decoding frames in the “wrong” order. The frames are sent out of sequence and temporarily stored. Figure 4.4 shows that although the original frame sequence is $I\ BB\ P$ ..., this is transmitted and decoded as $I\ P\ BB$ ..., so that the future frame is already in the decoder before bi-directional decoding begins.

Here is an example that involves three consecutive (open) GOPs. The second GOP is presented in bold face font for clarity reason:

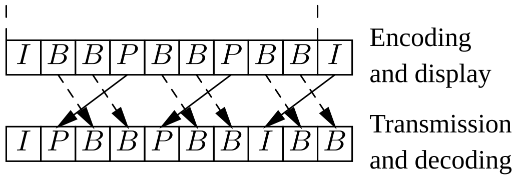


Figure 4.4: Changes in frame sequence

Display order:

$IB_1B_2P_1B_3B_4P_2B_5B_6\mathbf{I\ B_1B_2\ P_1\ B_3B_4\ P_2\ B_5B_6}IB_1B_2P1...$

Decoding order:

$I\,P_1\,B_1B_2\,P_2\,B_3B_4\mathbf{I}B_5B_6\mathbf{P_1\ B_1B_2\ P_2\ B_3B_4}I\,\mathbf{B_5B_6}\,P_1\,B_1B_2...$

In the decoding order, the two B frames after the second I frame are part of the first GOP, and the last two B frames of the second GOP come after the third I frame. Note that regularity is not intrinsic in the MPEG standard. Generally, an encoder will follow a certain scheme, like this one, but this is not required by the standard.

Picture reordering requires additional memory at the encoder and decoder and delay in both of them to put the order right again. The number of bi-directionally coded frames between I and P frames must be restricted to reduce cost and minimize delay, if delay is an issue.

4.2 MPEG-2 Processing Model

In its simplest form, playing out an MPEG-2 video stream requires three activities: input, decoding, and display. These activities are performed by three tasks, which are separated by an *input* buffer and a set of *frame* buffers, see figure 4.5.

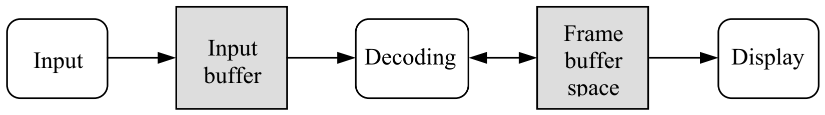


Figure 4.5: MPEG-2 processing model – tasks and buffers

4.2.1 Input task

The *input task* directly responds to the incoming stream. It places en encoded video stream in the input buffer at a certain rate, expressed in bits per second, the *bit rate, BR*. In the simple case, the input activity is very regular, and only determined by the fixed, constant bit rate. In a more general case, the input may be of a more bursty character due to an irregular source (e.g., the Internet), or due to a varying multiplex in the transport stream.

4.2.2 Decoding task

The *decoding task* extract the video data from the input buffer at a specific *frame rate, FR,* which is the number of frames per second, *fps*, and it is specified in the MPEG stream. Some common frame rate values are e.g., 24 fps, 25 fps and 30 fps (or to be more precise, 23.999.., 24.999... and 29.999..fps). It decodes extracted frames and puts the result (decoded frames) in the frame buffers. The decoding times for frames can vary, depending on the frame bit size and the used compression technique.

If sufficient buffer space is available, the decoders may work asynchronously, spreading the load more evenly over time. Its deadline is determined by the requirements of the display task. If B frames are present in the stream, the decoder performs frame reordering, i.e., the display order differs from the decoding order. This means that the frames are offered to the display task at irregular intervals. Reference frames are offered to the display task *after* the B frames they helped to decode.

4.2.3 Display task

The *display task* is IO bounded, and often performed by a dedicated co-processor. It is driven by the refresh rate of the screen, the *display rate, DR*. The display task, once started, must always find a frame to be displayed. In the simple case, the display rate equals the frame rate, but we will also consider situations where the display rate is higher than the frame rate.

4.3 Latency and Buffer Requirements for video processing of MPEG-2

Once we start to play out a video stream, the *end-to-end latency* is fixed and it is measured from the arrival of the first bit at the input task to the display of the first pixel or line on the screen. If this latency is not fixed, the system cannot work correctly over time [56].

The end-to-end latency is the sum of the *decoding latency*, and the *display latency*, see figure 4.6. The decoding latency and the display latency are not necessarily fixed.

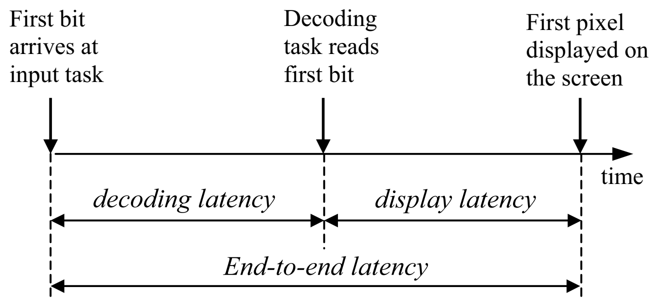

First bit arrives at input task
Decoding task reads first bit
First pixel displayed on the screen
time
decoding latency
display latency
End-to-end latency

Figure 4.6: End-to-end latency for MPEG playout

The initial decoding latency is measured from the arrival of the first bit at the input task to the reading of the first bit of the first frame, after the header, by the decoder.

The initial display latency is measured from the reading of the first bit of the first frame, after the header, by the decoder, to the display of the first pixel on the screen.

If the decoding task is strictly periodic, the decoding and display latencies are constant. If the decoder is asynchronous, i.e., if its activity is determined by the buffer fillings, both latencies can vary due to different decoding times for frames.

4.3.1 Input buffer requirements

We have mentioned earlier that the input task reads the MPEG stream and puts the video data in the input buffer. The buffer occupancy rises linearly during the decoding of each frame, and drops vertically at the start of a new frame, when the picture data are removed from the input buffer.

The input buffer serves several purposes. First, it has to compensate for the irregular data size for different frames. This irregularity is bounded, and the bounding is encoded in the stream, in the form of a parameter called *VBV buffer size*. VBV stands for Video Buffering Verifier, a hypothetical *reference decoder* that is conceptually connected to the output of the encoder. It has an input buffer known as the VBV buffer. VBV's purpose is to provide a constraint on the variability of the data rate that an encoder or editing process may produce. In VBV, decoding starts when the first frame has completely arrived in its input buffer, and retrieves a complete encoded frame out of the input buffer at the start of a new frame period. The contents of the VBV input buffer never exceeds VBV buffer size, thus, part of the definition of a compliant video stream is that it does not cause underflow or overflow of this model buffer, see MPEG video standard [1] for details.

Second, the input buffer has to compensate for varying decoding times, which are not foreseen by the encoder. Therefore, this compensation cannot be bounded a priori.

Third, a realistic decoder retrieves the data from the input buffer according to its processing. The resulting non-zero retrieval time relaxes the buffer requirement, but can also not be bounded a priori. Therefore, the input buffer size is essentially a design choice, closely related to the initial decoding latency and the desired end-to-end latency.

Once the size of the input buffer is chosen, the maximum decoding latency of the reference decoder, RDL_{max}, is fixed:

$$RDL_{max} = \frac{IBS}{BR}$$

where IBS is the input buffer size, and BR the bit rate.

4.3.2 Frame buffers requirements

The frame buffers serve a dual purpose. They serve as reference buffers for the decoder and as input buffers for the display task, or output buffer for the decoding task. It is possible that a certain frame buffer is used in both capacities at the same time. This makes frame buffer management somewhat more complicated than input buffer management. The display task cannot start until the first frame has been placed in the output buffer, and does not release the current output buffer until a second output buffer is available (double buffering scheme). In this way, the display task always has a frame to display.

If the stream contains two or more B frames in sequence, the minimum number of frame buffers needed is four: two for the reference frames, one for the B frame being displayed, one for the B frame being decoded. The use of four frame buffers allows a certain irregularity in the delivery of output frames by the decoder. For example, if we have a GOP structure with two B frames between each pair of reference frames, i.e., $IBBPBBP...$, the second B frame can be decoded in the same frame period as the first B frame is being displayed, since they are using different frame buffers. In general, when the n-th frame is being displayed, the $(n+1)$-th frame is decoded. Therefore, the minimum display latency equals two frame periods. If there are no B frames, there is no frame reordering, and the minimum display latency will be one frame period instead of two.

In the example above, the decoding cannot be done with less than four frame buffers, but these four frame buffers do allow a larger display latency. For example, we can maximize the display latency by not displaying B frames when they are completely decoded, but when the buffer is needed to decode the next frame. Now the n-th frame is being displayed while the $(n + 3)$-th frame is being decoded, i.e., the display latency equals three frame periods. Thus the display latency is bounded between the minimum of two frame periods and a maximum of three frame periods.

4.3.3 Buffer overflow and underflow

Since the decoder is asynchronous, there is a risk of buffer overflow and underflow, which could result in severe visual artifacts.

Buffer underflow and overflow are illustrated in figure 4.7. Input buffer underflow, and frame buffer overflow occur when the decoder is too fast, i.e., when the decoding latency is too small and/or the display latency too large. The decoder is blocked until the input and/or output task catches up. This can be prevented by synchronization.

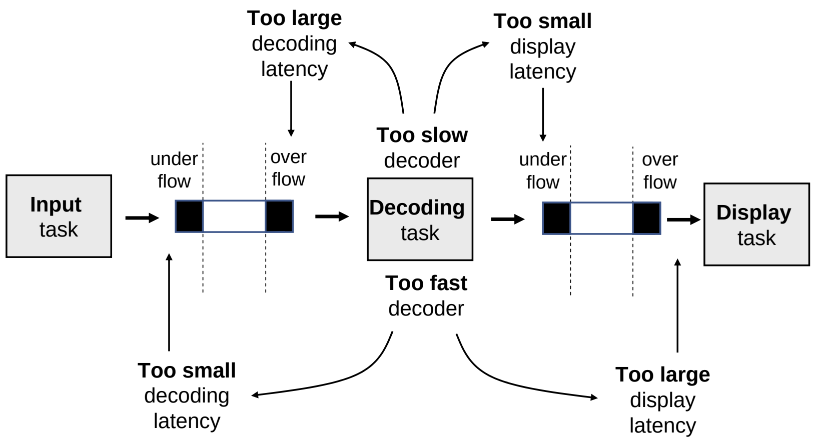


Figure 4.7: MPEG buffer overflow and underflow

Input overflow and output underflow occur when the decoder is too slow, i.e., when the decoding latency is too large and/or the display latency is too small. In case of output underflow, the display does not have a new frame to display, but this has been foreseen by retaining the previous frame for display until a new one arrives. Input overflow can be much more serious. In some cases, the input can be delayed, e.g., in case of a DVD player. In other cases, the input task cannot be blocked, especially in case of a broadcast input, where the input buffer must be made large enough to accommodate at least the variation that is allowed by the frame buffers.

The overflow is most likely to occur close to the end of a GOP. The decoder reads from the head of the buffer queue, the input task writes to the tail of the buffer queue. When the input buffer is full, two options are open for the input task: overwrite data at the head of the queue, or drop incoming data. In both cases, reference data will be destroyed, which will lead to a very serious artifact, because the remainder of the GOP cannot be decoded without these reference data. Therefore, preventing overflow at the input is imperative.

There are three measures that contribute to preventing overflow: judicious choice of end-to-end latency and input buffer size, speeding up the processing by allocating more processing resources, and preventive load reduction, e.g., by skipping frames. This will be discussed in details in the next section.

4.4 Playout under Limited Resources

The latency variation allowed is a design decision, based on the maximum allowed end-to-end latency, and the available buffer space. If the processor cannot work fast enough to meet the time constraints, the decoder has to speed up. There are two ways to do this: *quality reduction*, and *frame skipping*.

Whichever strategy is chosen, we assume that the system organization is such that the display task is never without data to display. This is not difficult to achieve. If a decoded frame does not arrive on time, and the display task has to redisplay the previous frame, this is a deadline miss for the decoder. With the given arrangement deadline misses have a penalty, in the form of a perceived quality reduction. Moreover, since the frame count has to remain consistent, the decoder must skip one frame.

4.4.1 Quality reduction

With the quality reduction strategy, the decoder reduces the load by using a downgraded decoding algorithm. This approach has two advantages over frame skipping. In general the decoding load is higher when there is more motion, but in that case, skipping frames may be more visible than reducing the quality of individual pictures. Moreover, quality reduction can be more subtle, whereas skipping frames is rather coarse grained.

The main disadvantage of the quality reduction approach is that it requires algorithms that can be downgraded, with sufficient quality levels to allow smooth degradation. Such algorithms are not yet widely available.

4.4.2 Frame skipping

Frame skipping means not decoding and displaying some of the frames. Frame skips speed up the decoder, and increase the display latency, i.e., the display latency is increased by a complete frame period when a frame is skipped.

There are two forms of frame skips, reactive and preventive. A *reactive frame skip* is a frame skip at or after a deadline miss to restore the frame count consistency. In case of a deadline miss, there are two options, aborting the late frame, which is probably almost completely decoded, or completing the late frame, and skipping the decoding of a later frame. In the former case, the display latency stays low, and a next deadline miss is to be expected soon. In the latter case, the display latency is drastically reduced, because the decoder will be blocked due to output buffer overflow. An additional frame buffer would give more freedom, and a more stable system, at the cost of using additional memory. In both cases, we have to make sure that the input buffer is large enough to allow the minimal display latency.

A *preventive frame skip* preventively increases the display latency. Skipping a frame takes a certain time, but much less than decoding it. Instead of rising, which is normal for B frames, the buffer occupancy drops during the frame skipping. The decision to skip preventively is taken at the start of a new frame, and is based on an measurement of the lateness of the decoder.

4.5 Chapter summary

MPEG-2 video stream is a sequence of frame pictures. I frames are self-contained, while P and B frames are predicted from other frames; the first one from a previous reference frame and the second one both from a previous and a next reference frame.

MPEG video processing consists of three tasks: input, decoding, and display. Input and display are usually IO bound and are executed on specialized co-processors. Decoding is computation-bound, and is executed on the CPU. The three tasks are separated by buffers, one input buffer, and a frame buffer space that contains at least two frame buffers.

The input task accepts the incoming stream at a certain bit rate, which can be constant and variable. The display task operates at fixed display rate. Both are hard timed. As a consequence, there is a fixed end-to-end latency between input and display. The decoding task is squeezed between the input task, which pushes the encoded data into the input buffer, and the display task, which pulls the decoded data out of the frame buffers. Hence, the decoding latency and the display latency can vary.

Underflow and overflow can occur in each of the buffers, having different consequences. For example, output underflow is less severe and it can be compensated for by redisplaying the last decoded frame. Input overflow can be much more serious: wrong data overwrite can ruin an entire GOP resulting in a severe visible artifact. Thus, preventing overflow at the input is imperative.

In this thesis, we use preventive frame skipping to speed up decoding and hence prevent this problem. In forthcoming chapters we will present criteria for frame skipping and use those to propose a quality aware MPEG-2 stream adaptation upon overload situations, but first will look into requirements for decoding frames and propose in next chapter realistic timing constraints for MPEG-2 video processing.

Chapter 5

Timing Constraints for Real-Time MPEG-2 Video Processing

Video and audio, as well as stream processing in general, have throughput requirements and real-time deadlines. For example, decoding a 25 fps video stream requires periodically a newly decoded frame every 40 ms. These deadlines are hard in the sense that missing a deadline causes an error, which can render a whole GOP unusable: if the decoding of an I frame is aborted due to a deadline miss, then no other frames in the same GOP can be reconstructed.

One way of meeting deadlines for MPEG-2 processing is to perform a guarantee test for frame decoding, based on the amount of available system resources. As a first step toward such a guarantee algorithm, we need to know the timing constrains imposed by MPEG-2 processing, i.e., we need start times and deadlines for frame decoding in order to be able to guarantee its timely execution.

In this chapter we derive realistic timing constraints for MPEG-2 video decoding. We start by outlining the sources of the constraints in section 5.1, followed by derivation of start time constraints and finishing time constraints for the decoding of video frames, in sections 5.2 and

5.3. We use these constraints in section 5.4 to propose earliest start times and deadlines for frame decoding. Section 5.5 concludes the chapter.

5.1 Sources of constraints

Timing constraints for an MPEG video decoder stem from roughly three sources:

- MPEG stream constraints – in particular frame ordering and their dependencies, poses mostly *relative* constraints. For example, in order to decode a B frame, its reference frames need to be decoded first. Besides, the backward reference frame must be transmitted and decoded before the B frame, meaning that it will have earlier decoding start-time but later display time than the B frame.

- Display rate constraints – related to the refresh rate of the screen, defines mostly *absolute* constraints. It depends on hardware characteristics, which in turn define when a picture should be ready to be displayed. Consumer TV sets typically have refresh rates of 50, 60, or 100Hz, computer screens may have more diverse values.

- Resource and synchronization constraints – incurred by the frame buffers. The number and handling of frame buffers depends on hardware and architecture design, i.e., the constraints will be implementation dependent. Therefore we do not include specific constraints, which would change with design decisions.

5.2 Start time constraints

Let f_i^j denote a frame with the decoding number i and the display number j. Note that, as outlined in chapter 4.1.4, the decoding order will differ from the display order, i.e., $i \neq j$, if the stream contains B frames.

For B frames $j = i-1$, for I and P frames, the display number depends on the MPEG stream and has to be determined via look-ahead.

The earliest time at which decoding a frame f_i^j can begin is the earliest point in time at which all of the following start time conditions, *STC*, hold:

Start Time Constraint 1: *Frame header parsed and analyzed*:

$$STC_1(f_i^j) \geq t_H(f_i^j) \tag{5.1}$$

where t_H is the time it takes to parse the frame header and extract relevant information needed for frame decoding. This time is platform dependent.

Start Time Constraint 2: *For B and P frames: the decoding finishing time (ft) time of the forward/backward reference frame*:

$$STC_2(f_i^j) \geq ft(f_k^l),\ l < j \tag{5.2}$$
$$STC_2(f_i^j) \geq ft(f_m^n),\ n > j \tag{5.3}$$

where f_k^l is the backward reference frame and f_m^n the forward reference frame of f_i^j. We will see in next section how latest finishing times for frames can be calculated.

Start Time Constraint 3: *Frame data available in input buffer*:

$$STC_3(f_i^j) \geq CIT(f_i^j) = \sum_{n=1}^{j} \frac{FS(f_k^n)}{BR(f_k^n)} \tag{5.4}$$

where CIT is the cumulative input time of a frame f_i^j, and it depends on the frame size, FS, and the bitrate, BR, of all previously decoded frames (k is correspoding display index of a frame with the decoding index n and it depends on the stream structure, i.e., the number of B frames between two consecutive reference frames).

Start Time Constraint 4: *Free frame buffer available.*

$$STC_4(f_i^j) \geq t_B(f_i^j) \tag{5.5}$$

where t_B is the earliest time when a frame buffer is available. This is always naturally true for reference frames: they require at least two buffers, one for the current frame and one for the previous reference frame it references to, see section 4.3. When a new reference frame is being decoded, at most one of them is needed for reference. As a consequence, for reference frames, STC_4 becomes true one frame period earlier than it would for B frames.

The last two constraints are necessary for unblocked video stream processing.

5.3 Finishing time constraints

The latest time at which decoding a frame has to be completed is the earliest point in time at which any of the following finishing time conditions, *FTC*, holds:

Finishing Time Constraint 1: *Required display time of the frame*.

$$FTC_1(f_i^j) \leq RDT(f_i^j) \tag{5.6}$$

where RDT denotes the required display time of the frame. We will see now how it can be obtained.

If we have a TV set displaying a digital broadcast stream, DTV, the input frame rate is equal to the display frame rate: 50 - 60 Hz, depending on the region. Other input streams may have different frame rates, and other displays may have different display rates, i.e., the display rate is a multiple of the frame rate:

$$DR = \rho * FR$$

The frame period, T_{fr}, is equal to $1/FR$, while the display period, T_{dis}, is equal to $1/DR$. This means that ρ can be expressed as:

$$\rho = \frac{DR}{FR} = \frac{T_{fr}}{T_{dis}}$$

If the display rate is an integer multiple of the frame rate, i.e., ρ is an integer, the solution is simple, since the frame period and the display period will be harmonic. If this is not the case, things are more complicated. We will discuss both cases.

Case 1: *Display rate is an integer multiple of the frame rate:*

$$\rho \in Z^+$$

where Z^+ is a set of positive integers (i.e., $\rho = 1, 2, 3, ...$).

In this case, the required display time, RDT, of a frame with the decoding number i and the display number j is given by:

$$RDT(f_i^j) = IDL + (j-1)T_{fr} \tag{5.7}$$

where IDL stands for initial display latency, i.e., the display time of the first frame, as described in section 4.3. IDL includes ”catching in” on the display period.

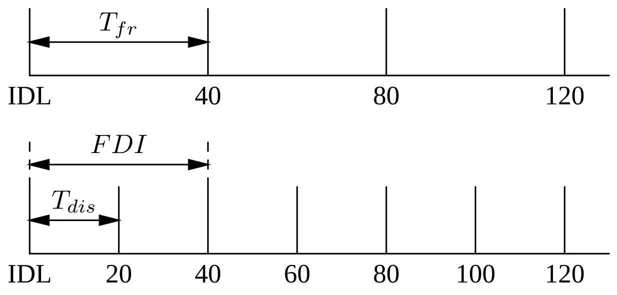
The length of the time interval in which a frame can be displayed is, in this case, the same for each frame, i.e., the length of the *frame display interval*, FDI, is equal to the frame period:

$$|FDI(f_i^j)| = T_{fr}$$

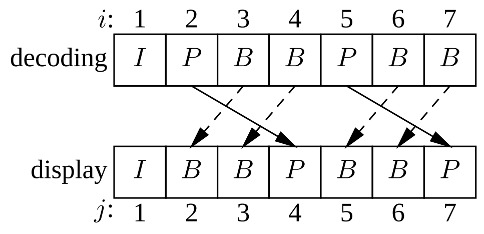
This implies that each frame will be re-displayed the same number of times, i.e., the repetition rate for the frames, R, is constant and it is equal to:

$$R(f_i^j) = \frac{|FDI(f_i^j)|}{T_{dis}} = \frac{T_{fr}}{T_{dis}} = \rho$$

Figure 5.1 depicts a simple example: assume an MPEG stream with a GOP structure $I\ BB\ P\ BB\ P$. If the frame rate is 25 fps, and the display rate is 50 fps, then we will have two invocations of the display task per frame. The frame period, T_{fr}, is equal to $1/25 = 40ms$, while the display period, T_{dis}, is equal to $1/50 = 20ms$, as shown in figure 5.1-a.



a) Frame rate and display rate, $DR = 2 * FR$



b) Decoding and display numbers of the frames

f	i	j	RDT	$\|FDI\|$	R
I	1	1	IDL + 0	40	2
B	3	2	IDL + 40	40	2
B	4	3	IDL + 80	40	2
P	2	4	IDL + 120	40	2
B	6	5	IDL + 160	40	2
B	7	6	IDL + 200	40	2
P	5	7	IDL + 240	40	2
...	...	...	...	...	...

c) Required display times, frame rates and intervals

Figure 5.1: Case 1 - display rate is an integer multiple of the frame rate

Decoding and display numbers are depicted in figure 5.1-b. For B frames, $j = i - 1$, e.g., the first B frame will have the decoding number 3 and the display number 2. For reference frames in this example, $j = i + 2$ (except for the first I frame in the stream which will have the same display and decoding number).

Finally, figure 5.1-c presents the corresponding frame intervals, repetition rates and required display times for the frames. Note different decoding and display numbers for the frames, e.g., the first P frame will have the decoding number 2, but its display time is 4, since we must display the two B frames first.

Case 2: *Display rate is not an integer multiple of the frame rate:*

$$\rho \notin Z^{+}$$

If the display rate is not an integer multiple of the frame rate, than we can only find approximate solutions. Here is an example: assume that we have an input frame rate of 24 Hz (original film material), and a display rate of 80 Hz (computer display). The decoding period is proportional to the frame rate, i.e., $T_{dec} = 1/24 = 41.666...$ ms, whereas the display period is $T_{dis} = 1/80 = 12.5$ ms, as illustrated in figure 5.2-a.

Since the decoder task is not in phase with the display task, the required display times will not overlap with starts of new frame periods, as in case 1. There are two ways to display frames:

Approach 1: Always postpone. The required display time for a frame is always *after* start of the corresponding frame period.

For example, the required display time of the first B frame in the example from figure 5.2 (the one with $i = 3$ and $j = 2$) is equal to the start of the first display period that occurs *after* the start of B's frame period ($IDL + 41.666...$), which is $IDL + 50$. Similarly, RDT of the second B frame is the start of the first display period after $IDL + 83.333..$, which is $IDL + 87.5$ and so on, as shown in figure 5.2-b.

In this case, the required display time of the frames is calculated as:

$$RDT(f_i^j) = IDL + \lceil (j-1)\rho \rceil \, T_{dis} \tag{5.8}$$

Approach 2: Take the closest one. The required display time for a frame can be *before or after* start of the frame period, whichever is closest.

Let $\Delta_L(f_i^j)$ and $\Delta_R(f_i^j)$ denote the time distance from the start of f's frame period to the closest left respective right start of display period, i.e.,:

$$\begin{aligned}\Delta_L(f_i^j) &= (j-1)T_{fr} - \lfloor (j-1)\rho \rfloor T_{dis} \\ \Delta_R(f_i^j) &= \lceil (j-1)\rho \rceil T_{dis} - (j-1)T_{fr}\end{aligned}$$

The required display time for this approach is given by:

$$RDT(f_i^j) = IDL + \begin{cases} \lfloor (j-1)\rho \rfloor T_{dis}, \text{ if } \Delta_L(f_i^j) < \Delta_R(f_i^j) \\ \\ \lceil (j-1)\rho \rceil T_{dis}, \text{ otherwise} \end{cases} \tag{5.9}$$

For example, for the first B frame, $\Delta_L(f_3^2) = 41.666 - 37.5 = 4.166$ and $\Delta_R(f_3^2) = 50 - 41.666 = 8.333$. Since Δ_L is less than Δ_R, the required display time is equal to $IDL + 37.5$ and not $IDL + 50$, as we would have in approach 1. Required display times for the other frames is shown in figure 5.2-c.

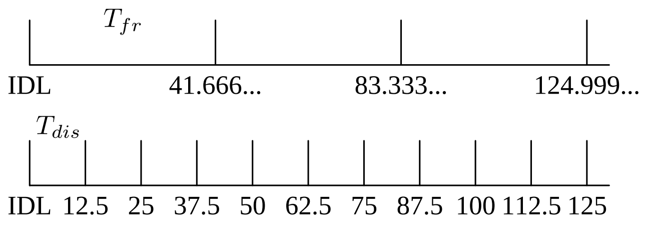
The repetition rate for the frames (both in approach 1 and approach 2) will not be constant for each frame, since the frame display intervals will have different length. For example, if we use approach 1, $FDI(f_1^1)$ in the example above will have length 50, while $FDI(f_3^2)$ will have length $87.5 - 50 = 37.5$.

The length of the frame display interval for the both appraches in this case is equal to the required display time of the frame that is to be displayed next, i.e., the one with the display number $j+1$ (and some decoding number k):

$$|FDI(f_i^j)| = RDT(f_k^{j+1}) - RDT(f_i^j)$$

The repetition rates are calculated as:

$$R(f_i^j) = \frac{|FDI(f_i^j)|}{T_{dis}}$$



a) Frame rate and display rate, $DR = 3.333.. * FR$

f	i	j	RDT	$\|FDI\|$	R
I	1	1	IDL + 0	50	4
B	3	2	IDL + 50	37.5	3
B	4	3	IDL + 87.5	37.5	3
P	2	4	IDL + 125	50	4
B	6	5	IDL + 175	37.5	3
B	7	6	IDL + 212.5	37.5	3
P	5	7	IDL + 262.5	50	4
...	...	...	...	...	...

b) Approach 1: always pospone

f	i	j	RDT	$\|FDI\|$	R
I	1	1	IDL + 0	37.5	3
B	3	2	IDL + 37.5	50	4
B	4	3	IDL + 87.5	37.5	3
P	2	4	IDL + 125	37.5	3
B	6	5	IDL + 162.5	50	4
B	7	6	IDL + 212.5	37.5	3
P	5	7	IDL + 250	37.5	3
...	...	...	...	...	...

c) Approach 2: closest instance

Figure 5.2: Case 2 - display rate is not an integer multiple of the frame rate

Approach 1 is a little more relaxed in terms of precise latencies, and thus deadlines. Apparently, the choice between approach 1 and 2 does not really matter with respect to relative frame jitter. In both cases, we get a cycle of three frame intervals: 50, 37.5, 37.5. However, the relative frame jitter is important for perception. In high quality video where the jitter is not accepted, this problem has been solved by using interpolation, i.e., making new frames. This feature is known as *natural motion* [22].

Finishing Time Constraint 2: *Imminent overflow of input buffer*.

$$FTC_2(f_i^j) \leq t_O(f_i^j) \tag{5.10}$$

where t_O is the time at which the input buffer overflow occurs. By a judicious choice of input buffer size, as outlined in section 4.3, FTC_2 will always be met. Should the completion constraint be missed, though, data loss at the input buffer will occur, with the risk of having to recapture the stream, which will take at least the complete GOP or until the next sequence header.

5.4 Earliest start times and deadlines for frame decoding

We use the timing constraints presented above to propose start times and deadlines for frame decoding. Simply, we set the earliest start time and the deadline for decoding a frame to be the most strict start time and finishing time constraint for that frame. Thus, the earliest start time, est, and the deadline, dl, for decoding the frame f_i^j are equal to:

$$\begin{aligned} est(f_i^j) &= max\{STC_1, STC_2, STC_3, STC_4\} \\ dl(f_i^j) &= min\{FTC_1, FTC_2\} \end{aligned}$$

We will use those start times and deadlines in chapter 8 to propose a quality aware guarantee algorithm for frame decoding.

5.5 Chapter summary

MPEG streams are played on different display devices with different screen refresh rates. The challenge here is to match frame rates encoded in the stream with display rates used by the display devices. While this is quite straight forward for the case where the frame rate is an integer multiple of the display rate, e.g., frame rate is 25fps and the display rate is 50Hz, things gets more complicated when this is not the case, e.g., frame rate is 24 fps and display rate is 80Hz.

In this chapter we derived realistic timing constraints for MPEG-2 decoding. We proposed two ways of dealing with the later case; the first one is to display the current frame in the first instance of the display task that occurs after the frame deadline, i.e., always postpone, and the second method is to use closest instance of the display task, either before or after the frame deadline, whichever is closest.

As a final result, we proposed a set of start time and completion time constraints for MPEG video decoding and used these to set the earliest start times and the deadlines for frame decoding. These will be used later to provide a real-time guarantee algorithm for frame decoding upon limited resources.

Chapter 6

Misconceptions and Realistic Assumptions about MPEG-2

Frame skipping needs appropriate assumptions to be effective. Skipping the wrong - even small - frame at the wrong time can ruin a whole GOP. As one of the initial steps towards a quality aware frame selection method upon resricted system resources, we have performed an exhausted analysis of diverse realistic MPEG-2 video streams, both on the frame and the sub-frame level.

In this chapter we report the results from our analysis and match those with common assumptions about MPEG. The objective was to check the validity of common assumptions for software MPEG-2 decoding and to propose realistic assumptions about MPEG-2 stream and processing, needed for frame skipping with focus on high video quality. We found a number of misconceptions present in the real-time community.

We start by presenting the analysis results in section 6.1. In section 6.2 we discuss the validity of common assumptions about MPEG-2 by comparing them to our results and findings. Furthermore, we present the sub-frame level analysis results in section 6.3, followed by conclusions in section 6.5.

Genre	Length	Fps	Resolution	Mbit/s	GOP	Frames
Action	118 min	25	720x576	9800	(12,3)	179412
Drama	107 min	25	720x576	8700	(12,3)	173054
Cartoon	104 min	25	720x576	6000	(12,3)	121406

Table 6.1: Some representative MPEG streams

6.1 Analysis of Realistic MPEG-2 Video Streams

We have analyzed a number of diverse streams taken from original DVDs and matched our results with the common MPEG assumptions. For the readability sake, in this chapter we report only representative results for selected MPEG-2 movies. The complete results for all analyzed movies can be found in appendix B.

6.1.1 The objective

We have measured frame sizes, decoding execution times, and GOP statistics such as total GOP sizes, the number of open and closed GOPs, the number of GOPs where the I frame is not the largest one, I,B,P frame patterns etc. Then we matched the obtained results with some common assumptions about MPEG video stream.

Since some video contents are more sensitive for quality reduction than others [47], we have analyzed different types of movies; action movies, dramas, and cartoons, see table B.1. Column GOP in the table B.1 represents the GOP structure of the streams, i.e., it refer to the length and distance between reference frames respectively, e.g., GOP structure (12,3) means I-to-I distance is 12, while I-to-P and P-to-P distance is 3.

6.1.2 Simulation environment

The MPEG video streams have been extracted from original DVD movies. To extract the data out of an MPEG video stream, we have implemented a C-program, see appendix C for details.

Frame type	Bytesize	Action movie	Drama	Cartoon
I	min	11	17	7178
	max	247073	183721	140152
	average	63263	58985	84318
P	min	2	4	159
	max	152000	126229	137167
	average	29352	28893	31943
B	min	4	4	159
	max	96131	79552	111405
	average	18525	19054	14398

Table 6.2: Frame size statistics

The decoding execution time measurements were performed on several PC computers, with different CPU speed (in the range 0.5-2.0 GHz). The time for measuring decoding execution times was equivalent to the length of the movies.

6.1.3 Analysis results

GOP and frame size statistics of the selected movies are presented in table 6.2. We have also analyzed the relations between frame sizes on the individual GOP basis, see table 6.3. “GOPs with the Same length = 82%” in the table 6.3 means that in the analysed movie 82% of the GOP had the same length, e.g. 12 frames per GOP, while 18% of the GOPs did not follow that pattern i.e., contains less or more frames. “9%” in the column “some P larger than I” of the table 6.3 means that in 9% of the GOPs there are at least one P frame that is larger then the I frame. The other columns in the tables are quite self-explanatory.

Furthermore, we have looked into size distribution for different frame types. Figure 6.1 depicts the frame size distribution for the action movie. The size range between the minimum frame size and the maximum frame size for different frame types has been divided into ten size intervals and the number of frames with sizes within respective interval is

Number of GOPs:	Action	Drama	Cartoon
Closed GOPs	17%	2%	1%
Open GOPs	83%	98%	99%
GOP with the same length	82%	92%	98%
I largest in GOP	90%	94%	92%
P largest in GOP	9%	5%	7%
B largest in GOP	1%	1%	1%
some P larger than I	9%	6%	8%
some B larger than I	5%	3%	1%
some B larger than P	39%	37%	12%
some P larger than a previous P	81%	84%	81%
some B larger than a previous B	97%	100%	100%

Table 6.3: GOP statistics

shown. For example, we can see from the figure that 50.4% of P frames have sizes between 37869 and 50492 bytes.

Finally, we have measured the decoding times for different frame types, see figures 6.2,6.3 and 6.4 for an example.

6.2 Common Assumptions about MPEG - Revised

Here we present some common assumptions about MPEG and match them with our analysis results. We have looked into stream assumptions, frame size assumptions, and a decoding time assumption.

Stream assumptions

Assumption 1: The sequence structure of all GOPs in the same video stream is fixed to a specific I,P,B frame pattern.

This is not true. For example, in 18% of the GOPs in the action movie the GOP length was not 12 frames. Not all GOPs consist of the same fixed number of P and B frames following the I frame in a fixed pat-

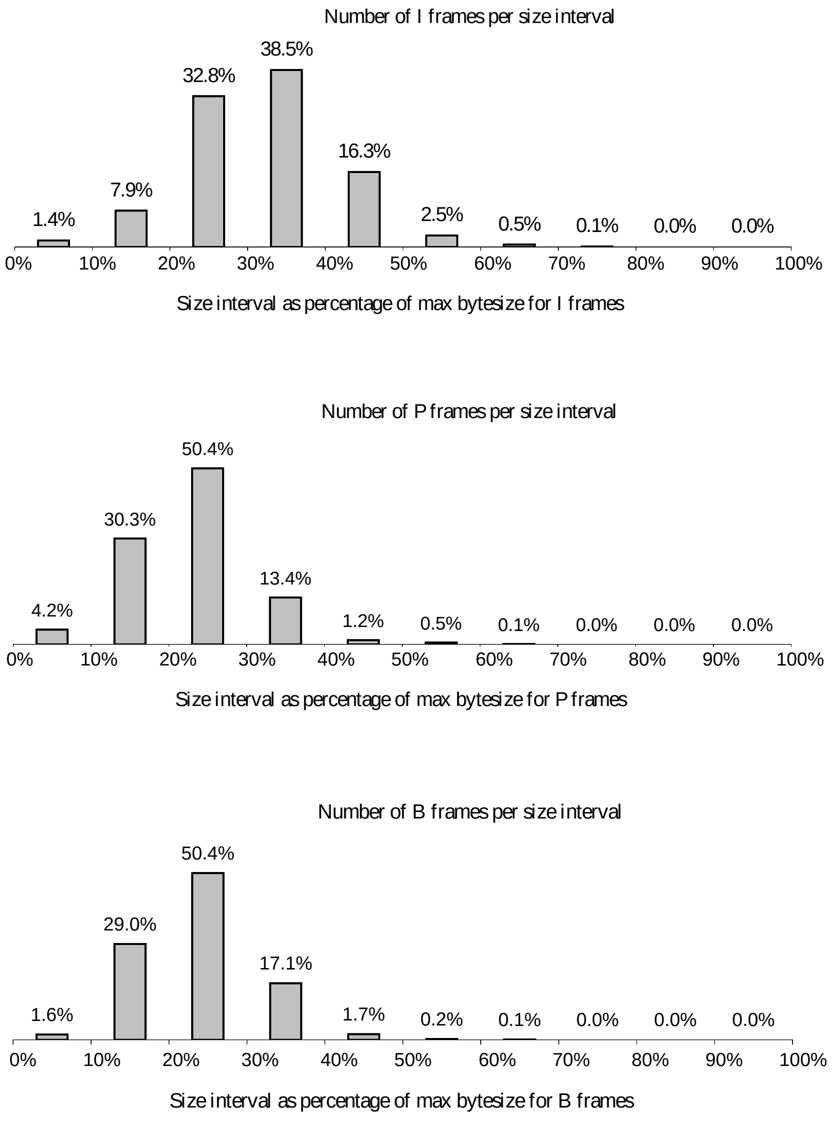


Figure 6.1: Frame size distribution for the action movie

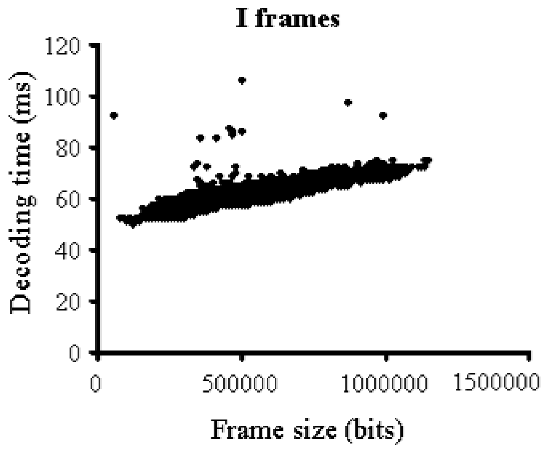


Figure 6.2: I frames: decoding execution times as a function of frame bitsize

tern. That is because more advanced encoders will attempt to optimize the placement of the three picture types according to local sequence characteristics in the context of more global characteristics.

Assumption 2: MPEG streams always contain B frames.

Not true. We have been able to identify MPEG streams that contain only I and P frames (IPP), or even only the I frames in some rare cases. Video streams that use only I frames exploit an older MPEG-2 technology that does not take advantage of MPEG-2 compression techniques. The IPP technology provides high quality digital video and storage, making it suitable for professional video editing. B frames provide the highest compression ratio, making the MPEG file smaller and hence more suitable for video streaming, but if the file size is not an issue, they can be excluded from the stream.

Assumption 3: All B frames are coded as bi-directional.

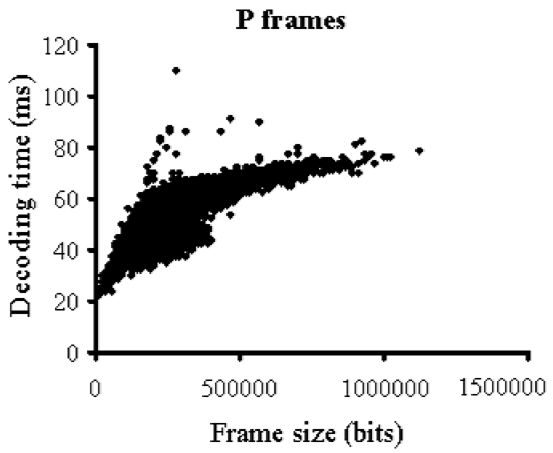


Figure 6.3: P frames: decoding execution times as a function of frame bitsize

This is not true. There are B frames that do have bi-directional references, but in which the majority of the macroblocks are I blocks. If the encoder cannot find a sufficiently similar block in the reference frames, it simply creates an I block.

Assumption 4: All P frames contribute equally to the GOP reconstruction.

Not true. The closer the P frame is to the start of the GOP, the more other frames depend on it. For example, without the first P frame in the GOP, P_1, it would be impossible to decode the next P frame, P_2, as well as all the B frames that depends on both P_1 and P_2. In other words, P_2 depends on P_1, while the opposite is not the case. Besides, all B frames that depend on P_2 will also (indirectly) depend on P_1, giving more frames that depend on P_1 than P_2.

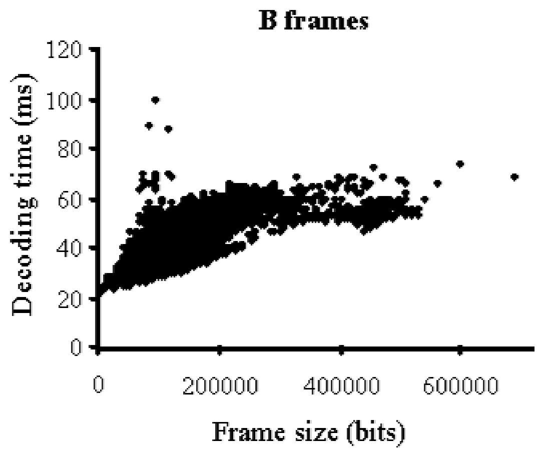


Figure 6.4: B frames: decoding execution times as a function of frame bitsize

Frame size assumptions

Assumption 5: I frames are the largest and B frames are the smallest.

It holds on average. In all the movies that we analyzed, the average sizes of the I frames were larger than the average sizes of the P frames, and P frames were larger than B frames on average. However, our analysis showed that this assumption is not valid for a significant number of cases. For example, in the action movie we have a case with 9% GOPs in which P have the largest size, and 1% of GOPs where a B frame is the largest one (see table 6.3), which corresponds roughly to 8 and 1 minutes respectively in a 90 minute film. Such deviations from average cannot be ignored.

Assumption 6: An I frame is always the largest one in a GOP.

This is not true. For example in the action movie the I frame was not the largest in 12% of the cases (in 9% of the cases some P frame was larger than the I frame, and in 3% of the GOPs, a B frame was larger than the I frame).

Assumption 7: B frames are always the smallest ones in a GOP.

Not true. For example, in the drama movie, a B frame was larger than the I frame in 3% of the cases, and larger than a P frame in 37% of the cases. As a consequence, even the assumption that P frames are always larger than B frames is also not valid. As another example, we found a GOP where the B frame is almost 100 times larger than the I frame ($B \approx 1MB, I \approx 12kB$).

Assumption 8: I,P and B frame sizes vary with minor deviations from the average value of I,P and B.

Not true. In the action movie, frame sizes vary greatly around an average, see frame size intervals in figure 6.1. For example, for B frames, the interval between 0.5 and 1.5 of average holds only some 70% of frames.

Decoding time assumptions

Assumption 9: Decoding time depends on the frame size and it is linear.

While some results on execution times for special kinds of frames have been presented, e.g., [12], a (linear) relationship between frame size and decoding time cannot be assumed in general. Our analysis shows, that the relation between frame size and decoding follows roughly a linear trend. The variations in decoding times for similar frame sizes, however, are significant for the majority of cases, e.g., in the order of 50-100% of the minimum value for B frames. As expected, the frame types exhibit varying decoding time behavior (see figures 6.2,6.3 and 6.4): I frames vary least, since the whole frame is decoded with few options only. On the other hand, B frames, utilizing most compression options, vary most.

6.3 Sub-frame Analysis

We have looked into MPEG-2 video streams on the sub-frame level. We investigated the type of macroblocks for each frame type. As explained in chapter 4.1.2 there are intra, forward-predicted, backward-predicted and forward-and-backward predicted macroblocks in a MPEG video stream. We investigated the amount of different macroblocks for the three frame types. The analysis result for an example movie is depicted in figure 6.5.

Furthermore, we have looked into so called *skipped macroblocks*[1], i.e., macroblocks for which no data is encoded. Skipped macroblocks are used to achive higher compression ratio. When a macroblock is skipped, it is implicitly defined by the standard in the following way: in a P frame, a skipped macroblock is a direct copy of the corresponding macroblock from the previous I or P frame. In a B frame, a skipped macroblock is reconstructed by assuming the motion vectors and motion type (i.e., forward, backward, or bidirectional) are the same as the last encoded macroblock. In this case, skipped macroblocks can not follow intra-coded macroblocks because then there would be not motion type or motion vectors defined.

The average numbers of skipped macroblocks per frame type for some example streams are presented in table 6.4. Skipped macroblocks per P and B frame for an example video stream are depicted in figure 6.6. We do not illustrate skipped macroblocks for I frames simply because we could not identify any stream with skipped I macroblocks.

Based on the sub-frame analysis we could check the correctness of some additional assumptions about MPEG-2:

Sub-frame asumptions

Assumption 10: Frame coding types (I,B,P) all consist of the same macroblocks types.

[1]The term ”skipped” should not be confused with ”frame skipping”

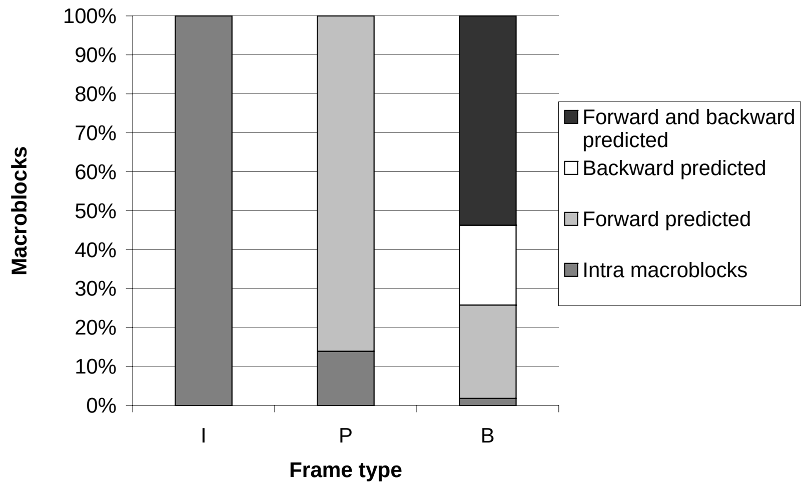


Figure 6.5: Macroblock types in an example MPEG stream

Not true. All macroblocks within an I frame are coded as intra. However, macroblocks within a P frame may either be coded as intra or inter (temporally predicted from a previously reconstructed frame). Macroblocks in a B frame can be independently selected as either intra, forward-predicted, backward-predicted or both forward and backward predicted. One example of this is given in figure 6.5.

Assumption 11: The presence of skipped macroblocks is content dependent.

This seems to be true. Depending on the content and the specific encoder algorithms used, the ability to employ skipped macroblocks may be highly variable from one stream to the next. So, different streams have very different usage of skipped macroblocks. For example, we can see from figure 6.4 that the number of skipped macroblocks for stream 3 is 25% while for stream 2 is only 1%.

Assumption 12: Skipped macroblocks are used in all three frame types.

Video Streams	I frames	P frames	B frames
stream 1	0%	19%	15%
stream 2	0%	1%	2%
stream 3	0%	25%	22%
stream 4	0%	12%	27%
stream 5	0%	3%	5%

Table 6.4: Skipped macroblocks per frame type

According to the MPEG standard [1], even I frames can have skipped macroblocks (that use only spatial redundancy), but we could not find any skipped I macroblocks in any of the analysed streams. The conclusion we make that skipped macroblocks are very seldomly used I frames.

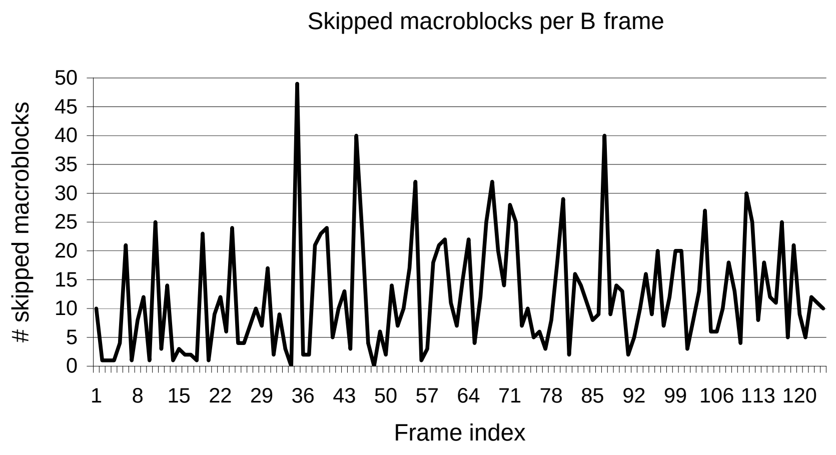
Furthermore, we can see from the figure 6.6 that the number of skipped macroblocks within the same video stream vary between frames.

By performing the sub-frame analysis we could make an interesting observation: the total number of macroblocks for some P frame is not equal to the sum of all intra macroblocks and forward-predicted macroblocks for the frame. P frames do not exploit backward prediction, i.e., they do not contain any backward-predicted macroblocks, hence the total sum off all macroblocks per a P frame should be the sum of all intra and forward-predicted macrolocks.

The explanation [2] is: in P frames (and only P frames), there are some macroblocks which are not intra (i.e., motion compensation is in use) but also do not define any forward motion vectors. By definition, these macroblocks are interpreted as using motion compensation with a motion vector defined as (0,0). It is a special case in the standard because it happens so often.

That what makes it interesting is those macroblocks are very good candidates for skipping on sub-frame level, since they other macroblocks

[2]Thanks goes to Ketan Patel from University of Nort Carolina for claryfing this phenomenon.



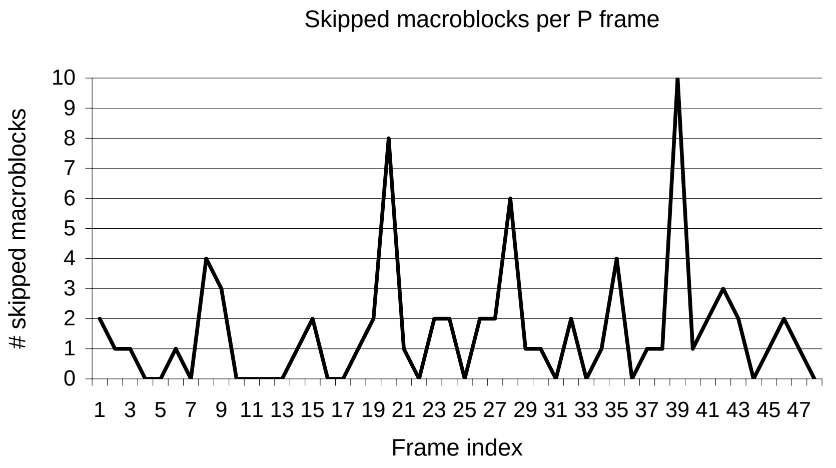


Figure 6.6: Skipped macroblocks variations

depend on them.

6.4 Valid assumptions about MPEG-2

Previously, we presented a number of common assumptions about MPEG-2 and then commented on their correctness. Here is a summary of identified valid assumptions that we are going to use in the reminder of this thesis:

- GOP sequences are not fixed to a specific I, P, B pattern.
- Not all MPEG-2 video streams contain B frames.
- Not all B frames are coded as bi-directional.
- The closer P frame to the start of the GOP, the more other frames depend on it.
- I frames are not necessarily the largest in a GOP.
- B frames are not necessarily the smallest in a GOP.
- Fame sizes can vary a lot from the average value (for a certain frame type).
- Decoding times for frames do not necessarily follow a linear trend with respect to the frame sizes.
- A frame can contain several macroblock types.
- The presence of skipped macroblocks is content dependent.
- Skipped macroblocks are used very rarely in I frames.

6.5 Chapter summary

In this chapter, we presented a study of realistic MPEG-2 video streams and showed a number of misconceptions for software decoding, in particular about the relation of frame structures and sizes.

For example, an intuitive conclusion is that I will be the largest frames, followed by P and B frames, and frames have similar sizes within their respective frame type. While true on average, such assumptions do not hold for a considerable number of cases. The analysis of realistic streams, movie DVDs, shows a case with 9% GOPs in which P frames have the largest size, and 1% GOPs where the largest frame is a B frame. This corresponds to roughly 8 and 1 minutes in a 90 minute movie. Clearly, such deviations from average cannot be ignored.

Algorithms based on average behavior, regarding the variations in frame sizes as small deviations will not provide acceptable quality.

Chapter 7

Quality Aware Frame Selection in MPEG-2

MPEG-2 video playout requires adequate system resources to be timely processed. This is especially true in software decoding, where video processing compete for the CPU with other tasks in the system. If we cannot provide enough resources to process a full-size MPEG-2 video, then video stream adaptation must take place. In the case of limited network bandwidth, we need to decrease the amount of transmitted video data but still ensure that enough *relevant* video data delivered in time to provide continuous and synchronized playout. If the processing power of the display device is restricted, we need to speed up decoding by, for example, not processing all video frames.

Frame skipping is a way to adapt video streams to the available system resources. Frames in a video stream can be skipped both before sending the stream on the network, if the network bandwidth is restricted, and on the display device, if the processing power is limited. However, frame skipping needs appropriate assumptions about the video stream to be effective. Skipping the wrong frame at the wrong time can result in a noticeable disturbance in the played video stream. On the other hand, if frames are skipped properly, we can provide high video quality while achieving good resource utilization.

In this chapter we provide a frame skipping approach for MPEG-2 with focus on high video quality perceived by users, that fully utilize the available system resource. In section 7.1, we use the identified valid assumptions about MPEG-2 processing from chapter 6 to propose a set of criteria for frame skipping. Based on these criteria, we propose in section 7.2 an algorithm for quality aware frame selection when it is not possible to decode all frames in time. Section 7.4 summarizes the chapter.

7.1 Criteria for Preventive Frame Skipping

Not all frames are equally important for the overall video quality. Skipping some of them will result in more degradation than others. Based on the analysis presented in the previous chapter, we here identify some criteria to decide the relative importance of frames.

Criterion 1: *Frame type*

According to this criterion, the I frame is the most important one in a GOP since all other frames depend on it, see chapter 4.1.2. If we lose the I frame in a GOP, then the decoding of all consecutive frames in the GOP will not be possible, since all other frames in the GOP depend directly or indirectly on the I frame, see chapter 4.1.2. B frames are the least important ones because they are not reference frames. Skipping one B frame will not make any other frame undecodable, while skipping one P frame will cause the loss of all its subsequent frames and the two preceding B frames within the same GOP. If we would apply this criterion only, then we would pull out all B frames first, then P frames and finally the I frame.

Criterion 2: *Frame position in the GOP*

This is applied to P frames. Not all P frames are equally important, see assumption 4 in the previous chapter, section 6.2. Skipping a P frame will cause the loss of *all* its subsequent frames, and the two preceding B frames within the GOP. For instance, skipping the first P frame (P_1) would make it impossible to reconstruct the next P frame (P_2), as well as all B frames that depends on both P_1 and P_2. And if we skip P_2 then we cannot decode P_3 and so on.

Criterion 3: *Frame size*

Applies to B frames. According to the previously presented analysis results, see assumption 9 in 6.2, there is a relation between frame size and decoding time, and thus between size and gain in display latency. The purpose of skipping is to increase display latency. So, the bigger the size of the frame we skip, the larger display latency obtained.

However, skipping large B frames might not always be the best option. Small B frames might exploit complex compression techniques which minimize frame size, but are more expensive to decode, in terms of needed processing power. Frame prediction from reference frames is found to be most computationally expensive [44]. Hence, if the network bandwidth is limited, then large B frames should be skipped first, and if the objective is to decrease the CPU load, small, more compressed frames should be skipped.

Criterion 4: *Skipping distribution*

With the same number of skipped B frames, a GOP with *evenly* skipped B frames will be smoother than a GOP with uneven skipped B frames, e.g if we have a GOP=$IBBPBBPBBPBB$ then even skipping $I-BP-BP-BP-B$ will give smoother video than uneven skipping $I--PBBPBB--$, since the picture information loss will be more spread [47].

Criterion 5: *Buffer size*

Buffer requirements has to be taken into account when designing a frame skipping algorithm. There is no point in having a nice skipping algorithm without having sufficient space to store input data and decoded frames, see chapter 4.3.3.

Criterion 6: ***Latency***

This is not really a criterion, but one must be aware of the fact that an algorithm that takes entire GOP into account requires a large end-to-end latency, and corresponding buffer size, see chapter 4.3.

When deciding the relative importance of a set of frames, e.g. a GOP, for the overall video quality, we assign values to them according to all criteria collectively applied, rather than applying a single criterion. Since the criterion 1 is the strongest one, the I frame will always get the highest priority, as well as the reference frames in the beginning of the GOP, while in some cases we would prefer to skip a P frame towards the end of the GOP than a big B frame close to the GOP start.

7.2 Frame Priority Assignment Algorithm

In this section we present our algorithm to select frames based on the criteria above. We apply the skipping criteria on a set of frames to assign different importance values to the frames. The lower the value for a frame, the sooner the frame will be skipped. The number of importance values is equal to the number of frames in the chosen set. That will provide for unique priorities between frames.

We apply our algorithm on a per GOP basis. However, our method can be applied even on larger frame sequences that consist of several GOPs. We chose GOP length since MPEG-2 video stream are divided in GOPs which provides for easier access of the relevant information. Besides, running the algorithm on a smaller number of frames, decreases the run-time overhead of the importance value assignment.

Note that, even if we need to check entire GOP to assign values to the frames, we do not need to buffer the entire GOP, since we only do a look-ahead in the stream where we check the GOP structure and count frame sizes.

7.2.1 Algorithm description

Here is the description and the pseudo-code for the assignment of the importance values among frames in a GOP:

Let:

N	= GOP length
M	= distance between reference frames
$\mathcal{P}$	= a set containing all P frames in the GOP
$\mathcal{B}$	= a set containing all B frames in the GOP
$v(f)$	= importance value of frame f
ESC_i	= i^{th} even-skip chain of B frames

Step 1 Assign the highest value to the I frame (equal to the number of frames in the GOP)

$$v(I) = N$$

Step 2 The set $\mathcal{P}$ contains all P frames, $\mathcal{P} = \{P_1, P_2, ..., P_k\}$, sorted according to their position in GOP (P_1 is closest to the I frame, i.e, the first P frame in the GOP, while P_k is the last frame in the GOP). The longer the distance from the I frame, the lower the importance value (P_1 will get the highest value and P_k the lowest one).

```
$\forall P_i \in \mathcal{P}, 1 \leq i \leq |\mathcal{P}|$
    $v(P_i) = N - i$
```

Step 3 Initially set all values for B frames to the lowest P value (e.g. P_k above)

```
$\forall B_k \in \mathcal{B}, 1 \leq k \leq |\mathcal{B}|$
    $v(B_k) = min[v(P_i) \mid 1 \leq i \leq |\mathcal{P}|] - 1$
```

Step 4 Identify all “even-skip” chains for B frames and sort them according to the total byte size. Decrease the importance values of the B frames, depending on which chain they belong to. The less the total byte of a chain, the less the values are assigned to belonging B frames.

```
4:  $ESC_1 = \{B_1\} \cup \{B_{1+j*M} \mid 1 \leq j \leq \frac{N}{M-1}\}$
    $\forall i, 2 \leq i \leq |\mathcal{B}^*|$
        $ESC_i = \{B_i\} \cup \{B_{i+j*M} \mid 1 \leq j \leq \frac{N}{M-1}\}$
        $if \;\; sum(ESC_i) > sum(ESC_{i-1})$
            $swap(ESC_i, ESC_{i-1})$
        $\forall B_k \in ESC_i$
            $v(B_k) = v(B_k) - |ES_{i-1}|$
```

The presented algorithm skips small B frames first. If the objective is to utilize limited network bandwidth, then the ”even-skip” chains above should be sorted in acceding order, i.e., large B frames should be skipped first, see discussion for criterion 3 in section 7.1.

7.2.2 Example

Assume the following GOP with respective bit sizes (taken from the action movie):

$$\begin{aligned} &I \;\; BB \; P \; BB \; P \; BB \; P \; BB = \\ &\quad \{734136, 89656, 96640, 119368, 89232, 74048, \\ &\quad 100680, 32112, 87080, 92064, 18336, 142008\} \end{aligned}$$

We want to assign importance values to frames according to our method. The number of frames in the GOP is 12, so the values will be between 1 and 12, 12 being the highest priority. The assigned values after each step are depicted on top of the frames. The frames with values that differ from the previous step will be highlighted by filled style. Also, P and B frames are indexed in order to distinguish between different frames of the same type.

We start by applying criterion 1:

12	10	10	11	10	10	11	10	10	11	10	10
I	B_1	B_2	P_1	B_3	B_4	P_2	B_5	B_6	P_3	B_7	B_8

According to this criterion, the I frame got the highest value 12, three P frames got the same value 11, and B frames are the least important, with value 10.

We continue by applying the criterion 2 on the P frames:

12	10	10	11	10	10	10	10	10	9	10	10
I	B_1	B_2	P_1	B_3	B_4	P_2	B_5	B_6	P_3	B_7	B_8

P_1 is closest to the I frame among all P frames. Hence P_1 will keep its assigned value (11), while the values of P_2 and P_3 will get decreased. Since P_2 is closer to the I frame than P_3, it will get higher value than P_3. By this we ensure that in overload situations P_3 will be dropped first, P_2 second and P_1 will be the last one among P frames to drop.

Since the value of P_3 is now the same as the values of B frames, we even need to decrease the B values to make sure that all P frames will be prioritized before any of the B frames:

12	8	8	11	8	8	10	8	8	9	8	8
I	B_1	B_2	P_1	B_3	B_4	P_2	B_5	B_6	P_3	B_7	B_8

We mentioned earlier that the criteria 3 and 4 should not be applied separately. For the criterion 3 we need to compare sizes for B frames. Let $s(f)$ denote the size in bits for a frame f. For the chosen GOP the following holds:

$$s(B_8)>s(B_2)>s(B_1)>s(B_3)>s(B_6)>s(B_4)>s(B_5)>s(B_7)$$

If we apply the frame size alone, then B_{1-8} frames would be assigned values 6, 7, 5, 3, 2, 4, 1 and 8 respectively (B_8 would get the highest value, 8, because it is largest). Assume now that we need to skip 4 frames. According to the assigned values, the skipping mechanism would produce the pattern: $I\ BB\ P\ B-\ P\ --\ P\ -B$, which is not the optimum for the video smoothness, as discussed before. Instead, we need to apply criterion 3 together with criterion 4 to obtain the best possible value assignment with respect to both frame sizes and even distribution of skipped frames. We start by identifying all “even-skip” chains (ESC) of B frames:

$$\begin{aligned} ESC_1: &\quad B_1 \rightarrow B_3 \rightarrow B_5 \rightarrow B_7 \\ ESC_2: &\quad B_2 \rightarrow B_4 \rightarrow B_6 \rightarrow B_8 \end{aligned}$$

We compare the total byte size in both chains, and we assign greatest values to the B frames in the chain with larger size:

$$\begin{aligned} size\ ESC_1: &\quad s(B_1) + s(B_3) + s(B_5) + s(B_7) = 229336 \\ size\ ESC_2: &\quad s(B_2) + s(B_4) + s(B_6) + s(B_8) = 402238 \end{aligned}$$

Since the total size of ESC_2 is larger than the size of ESC_1, we first decrease the values of ESC_1 by the number of frames in ESC_2, i.e., 4; we need those four values for frames in ESC_2. The new assignment is:

12	4	8	11	4	8	10	4	8	9	4	8
I	B_1	B_2	P_1	B_3	B_4	P_2	B_5	B_6	P_3	B_7	B_8

Next we do internal value distribution according to the frame sizes, in both ESC_1 and ESC_2. The largest frame in the chain gets the highest value. In ESC_1, B_1 will get value 4 because it is the largest in the chain, and B_7 gets the smallest value 1. Similarly, in ESC_2, B_8 keeps the value 8 and B_2 gets the lowest value in the chain, that is 4. The final value assignment is:

12	4	7	11	3	5	10	2	6	9	1	8
I	B_1	B_2	P_1	B_3	B_4	P_2	B_5	B_6	P_3	B_7	B_8

So, the frame skipping according to the assigned values is performed as showed below:

													(GOP size)
1)	I	B	B	P	B	B	P	B	B	P	B	B	1677822
2)	I	B	B	P	B	B	P	B	B	P	–	B	1659486
3)	I	B	B	P	B	B	P	–	B	P	–	B	1627374
4)	I	B	B	P	–	B	P	–	B	P	–	B	1538142
5)	I	–	B	P	–	B	P	–	B	P	–	B	1448486
6)	I	–	B	P	–	–	P	–	B	P	–	B	1374438

...and so on...

By doing this kind of value assignments for B frames we find the compromise between even skipping and frame sizes, because we make skipping decision based not only on the frame size but also on the relation to the other B frames in the GOP. i.e., the influence on the entire GOP.

7.3 Offline and online usage

The frame selection algorithm presented here can be used both offline and online:

Offline usage

We can apply our algorithm on a MPEG-2 stream to create several quality levels, i.e., several instances of the stream, each with different number of frames. An offline MPEG-2 transcoder transforms the stream into different qualities suitable for different receivers, and store them on a disc. When a user requests a video stream, the most suitable one, i.e., the one with quality level that matches best the receives resources, is sent back to the user.

If we transcode streams offline, we do not need to consider the complexity of the frame selection algorithm. Thus, we can apply more advanced selections strategies that involve longer frame sequences than a single GOP.

However, a problem with this approach is the granularity, i.e., how many quality levels should be create. Clearly, it is not feasible to precompute a stream for each possible combination of the available bandwidth and processing power. Therefore, in practice, we can only create streams for a small number of resource budgets, and use approximation when choosing a stream for certain resource budget. An example of such linear interpolation to approximate a policy for choosing quality levels to cope with load fluctuations has been proposed in [62].

Onine usage

The algorithm can also be used online, either by tailoring a certain stream before sending it on the network, or responding directly to an incoming stream and adapting it on the display device. In both cases, the stream is adapted according to the current system limitations, rather than producing offline streams for some bandwidth values.

This way, we can get more accurate stream adaptation based on real demands at a moment. At the same time, we do not need to store any pre-computed streams, since we do stream adaptation on the fly, mapping the current load to the number of frames to be skipped. However, runtime complexity must be kept low, which can be achieved by operation on per GOP basis, i.e., a small set of frames.

We also need mechanism to access the amount of available system resources online. We propose in the next chapter a method for online stream adaptation upon limited resources, that uses real-time methods for scheduling and resource reservation.

7.4 Chapter summary

When the system resources, such as available network bandwidth for transmitting or the processing power of the display device, are not sufficient to handle a full-size MPEG-2 stream, we decrease the load imposed by the stream by skipping frames. Based on valid assumptions about MPEG-2 in chapter 6, we proposed a set of criteria to be applied when determining the importance values of different frames for the overall video quality perceived by the user. We use those values to determine in which order frames should be skipped. The algorithm operates on a GOP basis, but it can easily be generalized for longer frame sequences.

While some other methods for quality-of-service in software MPEG-2 decoding make skipping decision based only on the frame type, e.g., [25],[46], or make no distinction between frames at all [67], we look into several important properties and relationships between frames. For example, we distinguish between P frames in a GOP, and assign different importance values to different P frames. Furthermore, we make difference between B frames, with respect to several criteria such as frame size, buffer and latency requirements, and the type of resource restriction, e.g., the processing power or the network bandwidth.

In the next chapter, we will use the real-time scheduling mechanisms presented in chapters 2 and 3 together with the frame selection algorithm presented here to flexibly schedule processing of MPEG-2 video in resource limited systems.

Chapter 8

Online Stream Adaptation

In the first part of the thesis we showed how to flexibly schedule mixed sets of tasks, i.e., periodic, aperiodic and sporadic, by using integrated offline and online approach. In the second part, we developed an algorithm for quality aware frame skipping, based on valid assumptions about MPEG-2 and realistic timing constraints for MPEG-2 video decoding.

In this chapter we combine both for a method for quality aware MPEG-2 stream adaptation in resource constrained systems. It uses the real-time scheduling mechanism presented in chapters 2 and 3 to flexibly schedule online processing of MPEG-2 video streams under limited resources. The frame selection algorithm takes into account the actual state of the system and determines the best set of frames utilizing the available resources and considering the quality based priority order for skipping. Thus, our algorithm selects frames based on concrete frame and system load knowledge and ensures that only decoding of frames which can be completed in time is started. While the frame skipping algorithm is independent of the actual guarantee algorithm used, making it suitable to work with a variety of algorithms and paradigms, we present its use with a concrete scheduler.

We start by giving an overview of our method in section 8.1. In section 8.2 we present the task model for the decoding tasks, based on the timing constraints from chapter 5. Section 8.3 describes the guarantee mechanism for frame decoding. Finally, we present results from a study underlining the effectiveness in section 8.4

8.1 Method Overview

Figure 8.1 gives an overview and the system architecture of our approach. We deal with systems with limited resources where frame skipping has to take place. Our method takes a MPEG-2 stream, the amount of available system resources and frame importance values as input, and produces a tailored MPEG-2 stream that can be timely processed with respect to available resources.

In chapter 7.1 we proposed a number of criteria to be applied when selecting the frames to be decoded. The frame priority assignment algorithm, proposed in the same chapter, uses those criteria to assign importance values to frames. The lower the value for a frame, the sooner the frame will be skipped (compared to the other frames). We also mentioned that the algorithm can be used for an arbitrary set of frames, but we chose here to perform frame selection on per GOP basis, for increased runtime efficiency, see chapter 7 for details.

The frame set with assigned importance values is then examined for feasibility by our guarantee algorithm. If not all frames in the current set can be decoded in time with respect to available resources, the guarantee algorithm uses the frame priorities to select which frames to skip first. It receives feedback from the system and decides how many and which frames can be decoded in time, depending on the current system load. If the guarantee algorithm fails to ensure decoding of all the frames in the set, it will start skipping some of them, starting with the frames that have the least impact on the high overall video quality. The output from the algorithm is the information about which frames can be successfully decoded in time, without causing any of other guaranteed tasks in system to miss its deadline.

The method above needs a mechanism to access the system load online. Video streams may vary significantly in their request for sys-

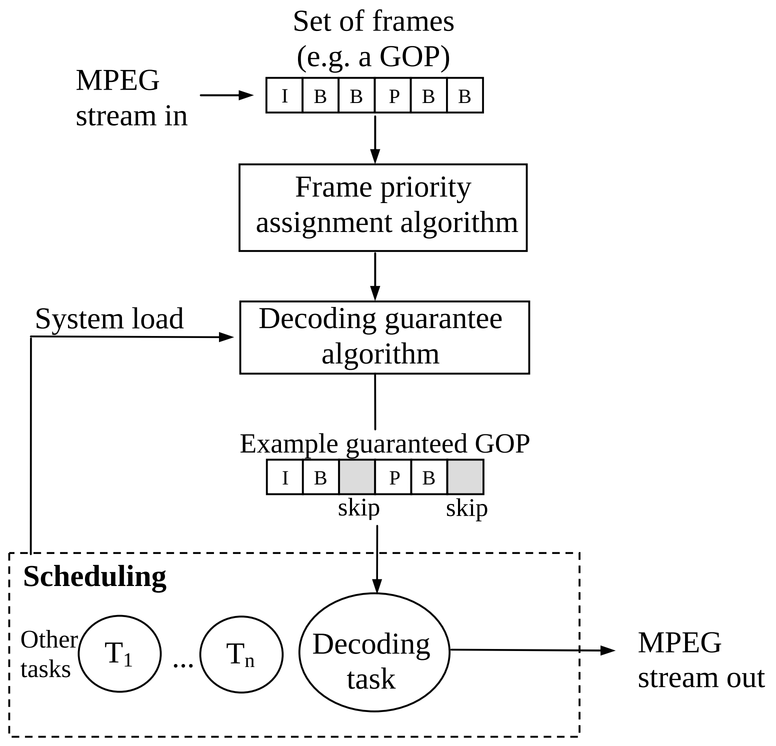


Figure 8.1: Method overview and system architecture

tem resources which causes the workload on display devices to change dramatically. General purpose operating systems are not capable of coping with these irregularities [5]. We use real-time methods presented in chapters 2 and 3 for scheduling and resource reservation. The amount and the distribution of available resources is calculated offline, and can easily be accessed at runtime through the slot shifting mechanism. However, other mechanisms for scheduling and resource reservation can be used as well. For example, we believe that any type of server-based algorithms, e.g., Total Bandwidth Server [51], Deferrable Server [42], Sporadic Server [50], candidate well as real-time scheduling methods to be applied with our frame selection algorithm.

8.2 Decoding Task Model

The decoding task reads the video data from the input buffer, decodes it and puts it into the frame buffers. It executes asynchronously, since the decoding latency and the display latency can vary, as discussed in chapter 4.3. In order to perform guarantee algorithm for timely decoding of frames, we need to know the timing constraints for the decoding tasks.

8.2.1 Start times and deadlines

Decoding task start time and deadline are determined by the requirements of the display task, and the buffer fillings. We use earliest start time and deadlines presented in chapter 5.4, which we calculated based on proposed start time and finishing time constraints, see chapter 5 on timing constraints for decoding task.

8.2.2 Execution times

It is difficult to predict worst-case execution times (WCET) for frame decoding. MPEG-2 can use different bit rates which can result in large differences in decoding times for different streams. This could lead to big overestimations of the WCETs. Our analysis shows that the relation between frame size and decoding follows roughly a linear trend. The variations in decoding times for similar frame sizes, however, are significant for the majority of cases, e.g., in the order of 50-100% of the minimum value for B frames. As expected, the frame types exhibit varying decoding time behavior: I frames vary least, since the whole frame is decoded with few options only. On the other hand, B frames, utilizing most compression options, vary most.

The guarantee algorithm for frame decoding that we present in next section requires known decoding times for optimal resource usage. There are two ways to make this information available at the guarantee time: offline analysis of the stream which gives exact decoding times, or, in more realistic scenario, prediction at runtime. The focus of this thesis is not to predict frame decoding times. Instead, we refer to some previous work. Predicting MPEG execution times has been presented in [9, 12], where the frame decoding time is predicted by frame type and size, and the corresponding predictor is shown to have less than 25% of prediction error.

However, it should be noted that our algorithm performs well even when the decoding times for frame are not accurate, or event based on the average decoding time for different frame types, as the evaluation in section 8.4 shows.

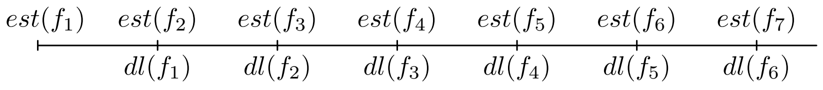
8.3 Guarantee Algorithm for Frame Decoding

Best-effort software decoders usually perform badly in the case of a deadline miss; they simply skip the current frame, without taking any consideration about the frame importance. In the worst case, the current frame could be an I frame, which would ruin the entire GOP. We base our skipping decision on the assigned importance values between frames. If the current frame is an important one, we do not skip it, instead we skip a less important frame.

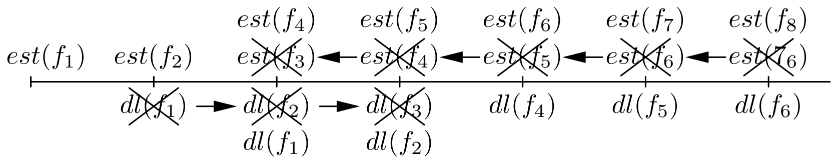
8.3.1 Basic idea

For each frame in a set of frames, we check how much available resources there are between the earliest start time and the deadline for the frame decoding. If the amount of available resources is less than the decoding execution demand of the frame, we must skip frames. When a frame is skipped, timing constraints for other frames in the set can be relaxed.

Assume, for example, the following earliest start times and decoding deadlines:



If we skip frame f_3, then the required display times (and hence the decoding deadlines) of all preceding frames can be relaxed, since we are neither decoding nor displaying f_3. We shift deadlines of all preceding frames to the right, i.e., $dl(f_2)$ becomes equal to $dl(f_3)$ and $dl(f_1)$ becomes $dl(f_2)$, as illustrated below:



Similarly, the earliest start times of the successor frames are shifted to the left, since the frames will become available in the input buffer earlier if we skip the current frame.

8.3.2 Algorithm description

Let $\mathcal{F}$ denote the currently guaranteed set of frames, and let f_s be the frame that is to be skipped, i.e., the one with currently lowest importance value among remaining frames in $\mathcal{F}$. Also let $\mathcal{P}$ and $\mathcal{S}$ denote subsets of $\mathcal{F}$, containing predecessor and successor frames of f_s respectively:

$$\begin{aligned}
\mathcal{F} &= \{\overbrace{f_1, f_2, ...,}^{\mathcal{P}} f_s, \overbrace{f_{s+1}, f_{s+2}, ..., f_n}^{\mathcal{S}}\} \\
\mathcal{P} &= \{\mathcal{P} \subseteq \mathcal{F} \mid \forall f_j \in \mathcal{P},\ 1 < j < s\} \\
\mathcal{S} &= \{\mathcal{S} \subseteq \mathcal{F} \mid \forall f_k \in \mathcal{S},\ s < k < n\}
\end{aligned}$$

Here follows a description:

1. Go through the current set frame by frame.
2. Compare the execution demand of the current frame with the available resources between the earliest start time and the deadline of the frame.

3. Get the frame with the currently minimum importance value.
4. Remove skipped frame from the current set of frames.
5. Relax deadlines for the predecessor frames and start times of the successor frames. The current frame f_i belongs either to $\mathcal{P}$ or $\mathcal{S}$, i.e., either its deadline or start time constraint is relaxed, which means in the next step of the while loop the amount of available resources for f_j will be bigger compared to the previous step (before skipping f_s).

Formalized:

```
1: $\forall f_i \in \mathcal{F}$
2:     while $availableResources[est(f_i), dl(f_i)] \leq c(f_i)$
3:         $f_s = minImportanceValue(\mathcal{F})$
4:         $\mathcal{F} = \mathcal{F} - f_s$
5:         $\forall f_j \in \mathcal{P}$
               $dl(f_j) = dl(f_{j+1})$
           $\forall f_k \in \mathcal{S}$
               $est(f_k) = est(f_{k-1})$
```

8.3.3 Alternative solutions for increased runtime efficiency

The complexity of the presented guarantee algorithm is polynomial with respect to number of frames in the guaranteed set: for each frame we skip (in the while-loop), the amount of frames in $\mathcal{F}$ will decrease (the for-loop), giving the worst-case complexity $O(N^2)$, where N is the number of frames in the set. As motivated in 7.2 we operate on per GOP basis, i.e., we chose $\mathcal{F}$ to be the current GOP. Considering very small values for N (in most cases 12-15 frames per GOP, see analysis in chapter 6), the algorithm is cost-efficient to run online.

If the computation cost is an issue, despite the simplicity of the guarantee algorithm above, less accurate, but faster solutions can be used instead. Assume the following GOP, with assigned importance values:

9	3	6	8	1	4	7	2	5
I	B_1	B_2	P_1	B_3	B_4	P_2	B_5	B_6

We check frame by frame and find that P_2 will not make it, unless we skip a frame.

Solution 1 - *Relax timing constraints only where needed.* When we discover that there are not enough resources for the decoding of the current frame f_i, we will skip the lowest priority frame f_s among all remaining frames. When we skip f_s, we know that all frames that precede f_i have already been confirmed to have enough resources, and we do not need to update their start times or deadlines. We only need to go through the frames between f_s and f_i and adjust their timing constraints. If $s < i$, we update start times, if $s > i$ we update deadlines, and if $s = i$ we skip the current frame f_i and continue with the other frames. In the example above, the skipping candidates are all frames in the GOP, and we choose to skip B_3, since it has the lowest priority. Then we update the earliest start times of B_4 and P_2, which will increase their execution windows.

This solution is sub-optimal in a sense that we do not relax timing constraints for frames that actually could be relaxed, i.e., already guaranteed frames I, B_1, B_2 and P_1. This will reflect on the flexibility of the scheduling of other tasks, since the decoding task will be more restricted than necessary, i.e., it will have a shorter execution window than needed. Still, this approach is better that just skipping the current frame, as naive decoders would do.

Solution 2 - *Skip only successor frames.* The second sub-optimal solution is to always skip either f_i or some of its successor frames, whichever has the lowest value. The gain here is that we do not need to go back, since all frames prior to f_i have already been checked. In the example above, the skipping candidates are frames P_2, B_5 and B_6, and we chose to skip B_5. Then we relax deadlines for P_2 and B_6. This solution is sub-optimal in a sense that we can skip a frame that has not the lowest importance value, e.g., B_3 has lower priority than skipped B_5, but this solution is faster than the one above.

8.4 Evaluation

Our frame decoding algorithm is applicable on a variety of methods which provide mechanisms for online access of the available resources. As an example, we show how we can adapt streams in the context of our previous work, i.e., combined offline and online scheduling.

8.4.1 Available system resources

We get the amount of available CPU resources by using the slot shifting mechanism, see chapter 2, and apply our guarantee algorithm to create a feasible stream.

First, an offline scheduler [27] creates scheduling tables for the selected periodic tasks with complex constraints. It allocates tasks to nodes and resolves complex constraints by constructing sequences of task executions. The resulting offline schedule consist of independent tasks with start times and deadlines, which can be re-scheduled by EDF at runtime, preserving original constraints.

Second, the offline schedule is divided into a set of disjoint execution intervals. Offline scheduled tasks can be executed flexibly within their intervals, i.e., they can be "shifted" in order to accommodate for tasks that arrive at runtime, e.g., aperiodic and sporadic tasks.

Third, we need to know amount and location of resources available after offline tasks are guaranteed. Spare capacities to represent available resources are then calculated for each interval.

8.4.2 Online access of available system resources

At runtime, we can access the amount and the distribution of available resources via intervals and spare capacities. For example, at any point in time at runtime we can easily calculate the amount of spare capacity between two time slots, t_1 and t_2, by simply summing up the spare capacities of the intervals between them.

Let I_{start} denote the interval that contains $t1$ and I_{end} the one containing t_2. Then, the total amount of spare capacities between t_1 and t_2 it is equal to the remaining spare capacity I_{start}, plus the sum of spare capacities of all full intervals between I_{start} and I_{end}, plus the remaining spare capacity of I_{end}:

$$sc[t_1, t_2] = max(sc_r(I_{start}) + \sum_{I_i \in (t1,t2)} sc(I_i) + sc_r(I_{end}), 0) \quad (8.1)$$

This mechanism suits very well for online stream adaptation, providing a simple and efficient way to access the amount of available CPU time at runtime, but as mentioned before, other mechanisms can be used as well.

8.4.3 Simulation setup

We have implemented and analyzed the quality aware frame selection algorithm (referred as *QAFS* from now on) and compared our algorithm with a naive, best-effort approach (*BE*), i.e., the one that has no guarantee mechanism for frame decoding. We have compared *QAFS* and *BE* with respect to the useful resource consumption per GOP and the total number of decoded frames.

By useful resource consumption we mean the time per GOP spent on useful decoding, i.e., fully decoded frames that contribute to the overall picture quality (as picture data and/or reference data). Wasted decoding time is the one spent on partial decoding frames that must be aborted due to decoding deadline misses.

We have also looked into the total number of decoded frames and compared it to the total number of frames successfully decoded by respective algorithm. Partialy decoded frames are not counted.

We have run the simulations on about 15000 GOPs. The system load per GOP was randomly distributed between the frames.

8.4.4 Simulation results

Figure 8.2 summarizes the comparison of *QAFS* and *BE* with respect to useful resource consumption. The *x*-axis in the figure represents the GOP *satisfaction degree*, which is the ratio between the resources needed for timely decoding of a GOP and the available system resource given to the GOP by the stream. E.g., if GOP satisfaction is 30%, then the GOP is given 30% resources of what it needs for decoding of all its frames. The *y*-axis shows how much of the granted time is spent on the useful decoding.

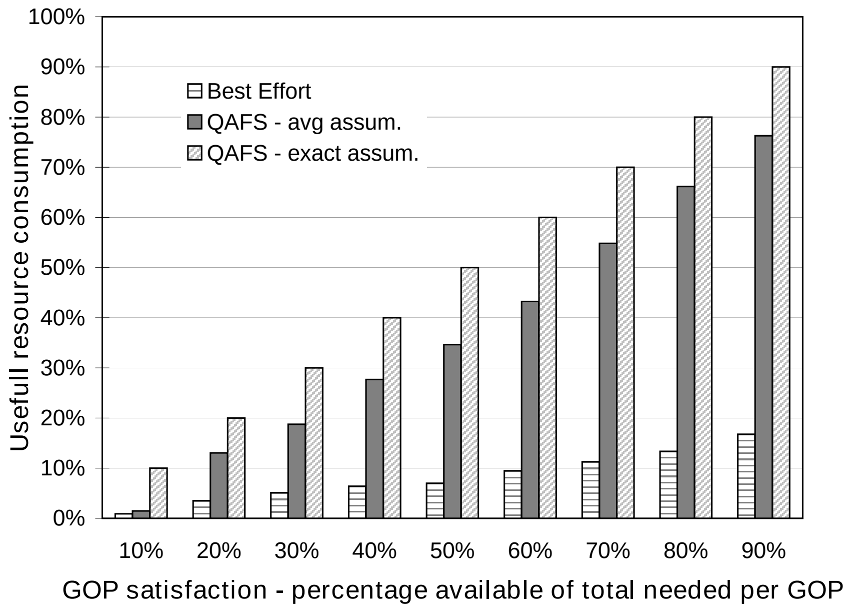


Figure 8.2: Simulation results - Useful resource consumption

We have simulated the case with known exact execution times for the decoding of frames, measured offline, and with the average decoding times for the respective frame types. As expected, the analysis shows that *QAFS* will not waste any resources at all if the exact execution times are known upon guaranteeing. Although there are not yet completely accurate method to predict decoding times online, we have shown the efficiency of our algorithm (once when such method is available).

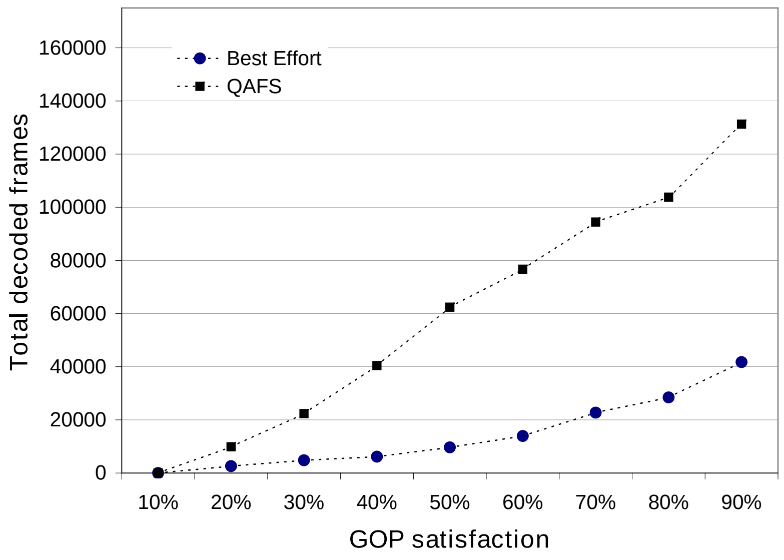


Figure 8.3: Simulation results - Successfully decoded frames

In a more realistic case with the average decoding times, *QAFS* will waste some resources due to the fact that it will guarantee decoding for some frames that cannot be decoded in time, since it performs frame guarantee based on the average decoding times and not the exact decoding times. So, whenever it accepts a frame that has the exact execution time that is larger than the average execution time, some resources will be wasted. Still, it performs much better than the naive, best-effort algorithm, as can be seen in the figure.

Since the GOP load is distributed between all the frames, the best effort algorithm will miss decoding deadlines whenever the total load in the execution window of the currently decoded frame is larger than the execution demand of the frame. This means that all decoding spent on the frame upon the deadline miss is wasted. On the other hand, when frames are skipped by *QAFS*, it will adjust the start times and the deadlines of the remaining frames in the GOP, giving them higher probability to meet their deadlines.

Figure 8.3 depicts the results from the second part of the experiment, the amount of successfully decoded frames.

As expected, the best-effort approach performs worse because it makes no distinction between frames: when a P frame is skipped, all its referring frames will also be skipped, and if the I frame is skipped, then no other frame in the GOP can be decoded. Our algorithm will never skip any reference frames unless it is absolutely necessary since it skips frames based on assigned importance values. If a frame need to be skipped skipped, we skip the one with the lowest importance value first, and since the lowest priority frames are not reference frame, no other frames will be influenced.

8.5 Chapter summary

In this chapter we presented a method for quality aware MPEG-2 stream adaptation under limited resources. We use a real-time method to access the amount of available system resources and, based on that information, tailor the video stream by decoding only those frame that are guaranteed to be decoded on time. We use our frame selection algorithm to determine which frames should be skipped if not all can be decoded in time.

We have compared our guarantee algorithm with a naive, best-effort approach that does not provide any guarantee for frame decoding. Our algorithm selects frames based on concrete frame knowledge and ensures that only decoding of frames which can be completed in time is started. It will not start decoding a frame unless we can ensure the that frame will be completely decoded and displayed in time. A naive algorithm will try to decode even those frames that cannot be decoded in time. It will start to decode a frame, and when the decoding deadline miss occurs, it will simply abort it, unnecessarily wasting the CPU time. Furthermore, if no frame distinction between frame types is done, a best-effort algorithm could, in the worst case, ruin an entire GOP by skipping the I frame.

We showed how our stream adaptation algorithm can be used together with the slot shifting method, but since there is a clear separation between the guarantee algorithm and the online resource reservation mechanism, even other real-time methods can easily be used.

Chapter 9

Conclusions

MPEG-2 is widely used as digital video coding standard, used in consumer electronics for DVD players, digital satellite receivers, and TVs today. In order to achieve high video and audio quality, digital media processing is required to provide continuous and synchronized playout without interrupts. At the same time, there are restrictions on the storage media, e.g., limited size of a DVD disc, communication media, e.g., limited bandwidth of the Internet, display devices, e.g., the processing power, memory and battery life of pocket PCs or video mobile phones, and finally the users, i.e., human's ability of perceiving motion. The challenge here is to to keep up the display speed even when resources, such as processing power and network bandwidth, are limited.

Most current software decoders, however, operate under the assumption of sufficient resources, using solutions based on average-case assumptions. The performance of such systems highly relies on the available bandwidth, processing power and the utilization of a large amount of buffers. Furthermore, they do not provide quality of service guarantees on the video. These provide acceptable quality for applications such as video transmissions over the Internet, when decreases in quality, delays, uneven motion or changes in speed are tolerable. In high quality consumer terminals, however, quality losses of such methods are not acceptable.

In this thesis we presented a method to flexibly schedule media processing in resource constrained systems. We proposed real-time methods for resource reservation of MPEG-2 video stream processing and introduced flexible scheduling mechanisms for video decoding. Our scheduling method is a mixed offline and online approach for scheduling of periodic, aperiodic and sporadic tasks, based on slot shifting, where a complete offline schedule can be constructed, transformed into EDF tasks, and scheduled at runtime together with other EDF tasks. The transformation is performed to maximize flexibility of task executions. First, we use the offline part of slot shifting to eliminate all types of complex task constraints before the runtime of the system. During offline analysis our algorithm determines the amount and location of unused resources, which we use to include dynamic activities during the runtime of the system. Then, we propose a new online guarantee algorithm for dealing with dynamically arriving tasks. Aperiodic and sporadic tasks are incorporated into offline schedule by making use of the unused resources and leeways in the schedule. In particular, we presented an efficient method to handle sporadic tasks, providing for $O(N)$ online acceptance test for firm aperiodic tasks. The sporadic tasks are guaranteed during design time, allowing rescheduling or redesign in the case of failure. At runtime, resources reserved for sporadic tasks can be reclaimed and used for efficient aperiodic task handling. Thus, our method combines handling of complex constraints, efficient and flexible runtime scheduling, as well as offline and online scheduling, providing a basis for predictably flexible real-time systems.

We used the scheduling mechanism and resource reservation mechanism from the mixed task scheduling part above to flexibly schedule processing of MPEG-2 video streams. First, we presented results from a study of realistic MPEG-2 video streams and showed a number of misconceptions for software decoding, in particular about relation of frame structures and sizes. We also identified constraints imposed by frame buffer handling and discussed their implications on timing constraints. Using the analysis, we determined realistic flexible timing constraints for MPEG decoding that call for novel scheduling algorithms, as standard ones that assume average values and limited variations, will fail to provide for good video quality. Based on the MPEG-2 analysis and proposed timing constraints, we presented a MPEG-2 video frame selection algorithm, to fully utilize limited resources and, at the same time, with focus on high video quality perceived by the users. The algorithm selects frames providing high video quality if not all frames can be completed in time due to limited resources, such as processing power of the display device or the network bandwidth, if the stream is transmitted. It is based on a priority ordering for frame skipping taking frame importance into account. The algorithm creates ensembles of decoding tasks for the frames in the Groups of Pictures in the stream, each with parameters suited specifically for the particular frame, instead of working with fixed, constant task parameters for periodic tasks. Applying real-time guarantee tests, the algorithm determines the best set of frames while matching the available resources.

The final result of the stream adaptation process is a tailored MPEG stream that is guaranteed to be decoded and displayed in time. The difference between our method and best-effort based algorithms that randomly skip frames if they run out of time, is that we:

- consider useful only what is finished. Partially decoded frames do not contribute to the overall video quality.
- decode only what is guaranteed to finish in time. We will not start decoding a frame unless we can ensure the frame will be completely decoded and displayed in time.
- select the frames that will give best possible video quality. We use a heuristic to determine which frames in a GOP are more important than the others.

While the frame selection algorithm is independent of the actual scheduling algorithm used, we presented an examplatory scheduler for our frame selection method.

Simulation results underline the effectiveness of our approach, even with imprecise execution times for frame decoding. The analysis showed that our method maximally utilize available resources if the exact execution times are known upon guaranteeing. Although there are not yet completely accurate method to predict decoding times online, we have shown the efficiency of our algorithm once a such method is available. In a more realistic case with the average decoding times, our method still performs much better than naive, best-effort algorithms.

It should be noted that media handling presented in this thesis is independent of the scheduling and resource reservation mechanism, i.e., other schedulers and resource management mechanisms can be used. Our approach is applicable on variety of other methods which provide mechanisms for online access of the available system resources. The system resources does not necessarily need to be the available CPU time, it can also be e.g., available network bandwidth: we could apply our method for video streaming through a network, i.e., we take available network bandwidth as input, and create feasible streams which are guaranteed to be transmitted in time. For example, we believe that any type of server-based algorithms that provide bandwidth reservation candidate well as real-time scheduling methods to be applied with our frame selection algorithm.

Appendix A

Simulation Results: Mixed Task Set Handling

We have implemented the algorithms described in chapters 2 and 3 and have run simulations for various scenarios.

In the first set of experiments we simulated the online guarantee algorithm for firm aperiodic tasks, described in chapter 2. We have studied the guarantee ratio for aperiodic tasks for different combinations of total system loads and aperiodic deadlines.

In the second set of experiments we have introduced sporadic tasks, as suggested in chapter 3, and have repeated the simulations for different combinations of periodic, sporadic and aperiodic tasks. We have measured the guarantee ratio of firm aperiodic tasks, depending on different scenarios for sporadic tasks. We also investigated how the variations in minimum inter-arrival times for sporadics influence aperiodic guarantee.

The simulation study underlines the effectiveness of the proposed approach.

A.1 Simulation environment

For the purpose of simulations we have developed a simulator to provide for detailed analysis of slot shifting. We also implemented a debugger, which provides for visual monitoring of the data structures during the simulations. See appendix C for the details on implemented tools.

Simulations were performed in parallel on 5 different PCs with the processor speed between 333 and 1500 MHz. Some 800 000 different interactions of sporadic, periodic and aperiodic tasks were simulated. The total length of simulation for both experiments was about 200 hours.

A.2 Experiment 1: Firm aperiodic guarantee

A.2.1 Experimental setup

For the first experiment series, we have randomly generated offline and aperiodic task loads, so that the combined load of both periodic and aperiodic tasks was set to 10% - 100%. The deadlines for the aperiodic tasks were set to their maximum execution time, MAXT, two times MAXT and three times MAXT. We studied the guarantee ratio for the randomly arriving aperiodic tasks.

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

A.2.2 Results

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

Figure A.4 illustrates the performance of background scheduling for three different deadline settings of aperiodic tasks. Figure A.5 depicts the performance of SSE. In figures A.1,A.2 and A.3 we put both methods together, for aperiodic deadlines equal to MAXT,2*MAXT and 3*MAXT, to see the difference in performance for different deadline settings.

As expected, background scheduling performed poorly in the high load situations, specially with tight aperiodic deadlines. For this reason, background scheduling can be adopted only when the aperiodic activities do not have stringent timing constraints and the periodic load is not high. The graphs show the efficiency of the SSE mechanisms, as guarantee ratios are very high. As expected, the guarantee ratio for aperiodic tasks with larger deadlines is higher than for smaller deadlines. Even under very high load, guarantee ratios stay high.

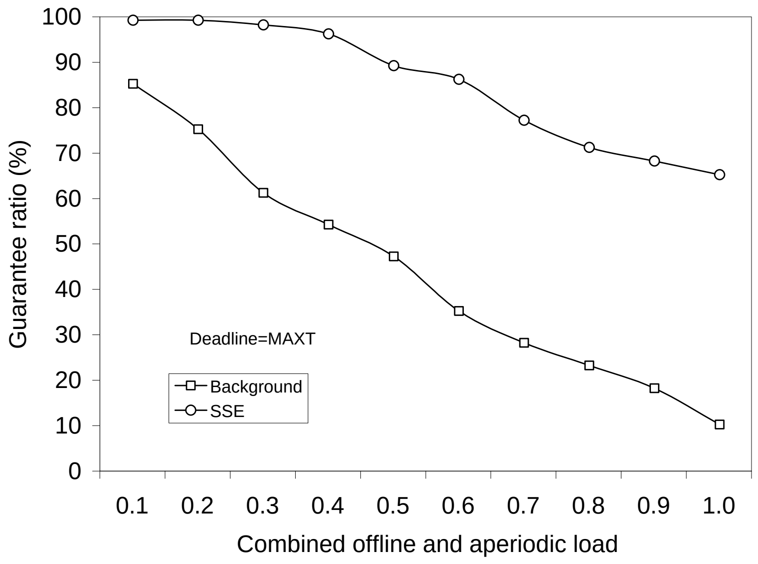


Figure A.1: Guarantee ratio for aperiodic tasks, dl=1*MAXT – SSE vs Bgr

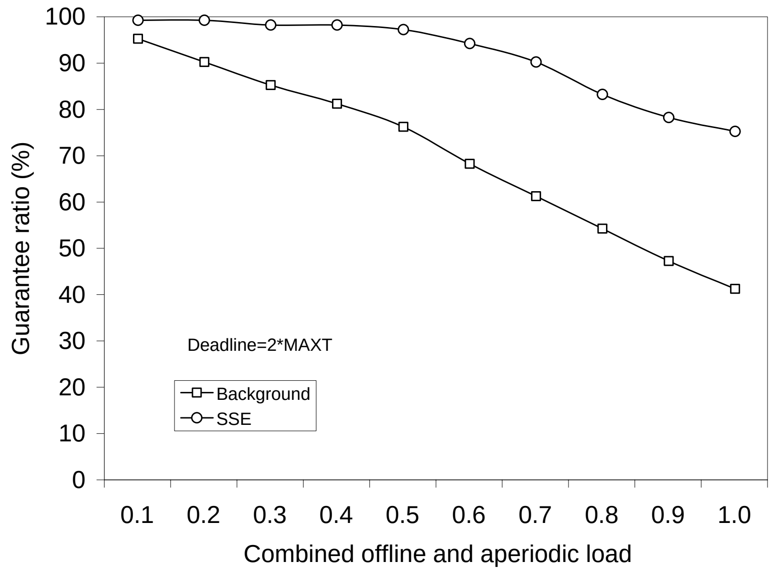


Figure A.2: Guarantee ratio for aperiodic tasks, dl=2*MAXT – SSE vs Bgr

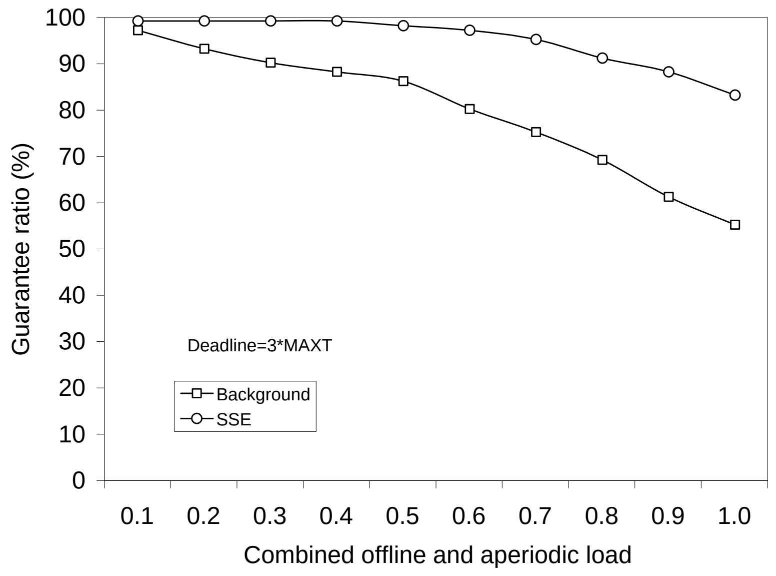


Figure A.3: Guarantee ratio for aperiodic tasks, dl=3*MAXT – SSE vs Bgr

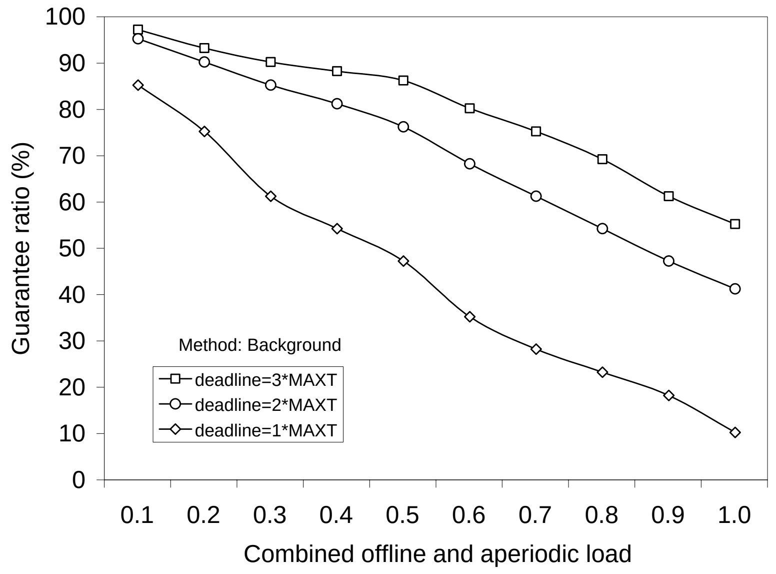


Figure A.4: Guarantee ratio for aperiodic tasks – Background

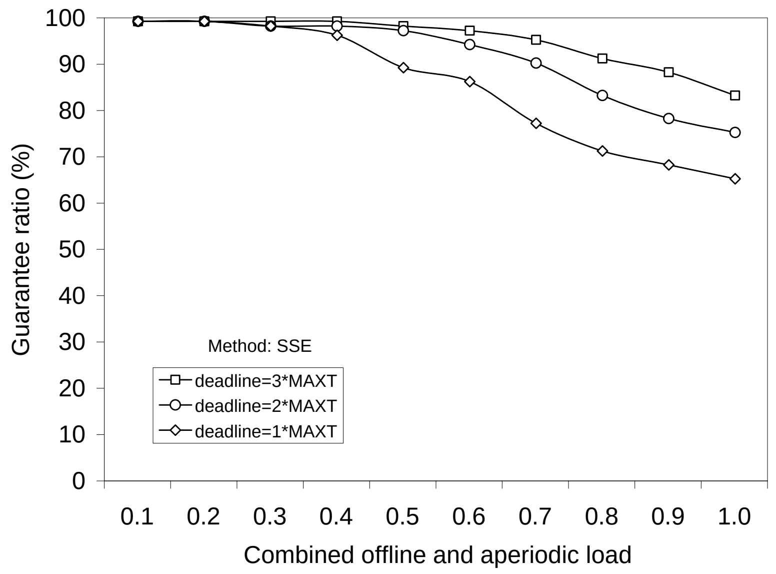


Figure A.5: Guarantee ratio for aperiodic tasks – SSE (our approach)

A.3 Experiment 2: Firm aperiodic guarantee with sporadics

In the second experiment, we have tested the acceptance ratio for firm aperiodic tasks with the methods to handle sporadic tasks: worst case arrivals without knowledge about sporadic invocations (referred as “no info”) and updated worst case with arrival info (“updated”). See chapter 3, section 3.3, case 1 and 2 for details about the different cases.

A.3.1 Experimental setup

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

In the second part of the experiment we varied the arrival frequencies of sporadic tasks according to a factor, f, such that the separation between instances $averageMINT$ is equal to $averageMINT = f * MINT$. This means that if $f = 1$ then the instances are invoked with the maximum frequency, and if $f = 2$, the distance between two consecutive invocations is $2 * MINT$ on average.

A.3.2 Results

The results from the first part of the experiment are summarized in figures A.6 and A.7, while the results from the second one are presented in figures A.8 and A.9.

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

Figure A.10 summarizes the simulation. We can see that guarantee ratio for firm aperiodic tasks is very high, even when we have sporadic tasks in the system. By keeping track off sporadic arrivals, we can accept firm tasks that otherwise would be rejected.

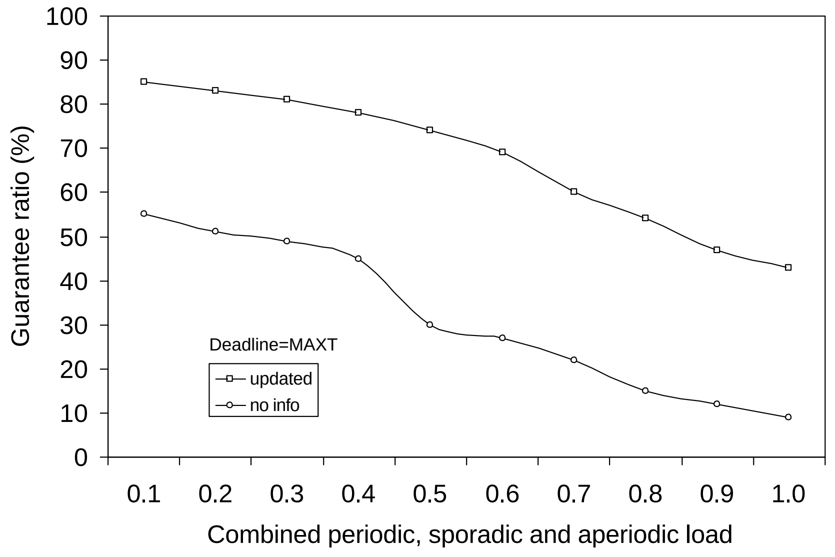


Figure A.6: Guarantee ratio for aperiodic tasks in the presence of sporadics tasks, dl=MAX: load variation

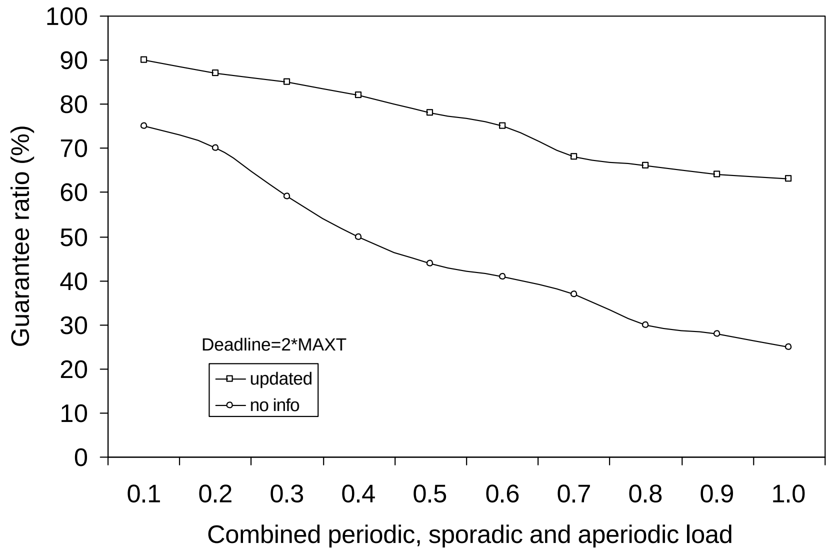


Figure A.7: Guarantee ratio for aperiodic tasks in the presence of sporadics tasks, dl=2*MAXT: load variation

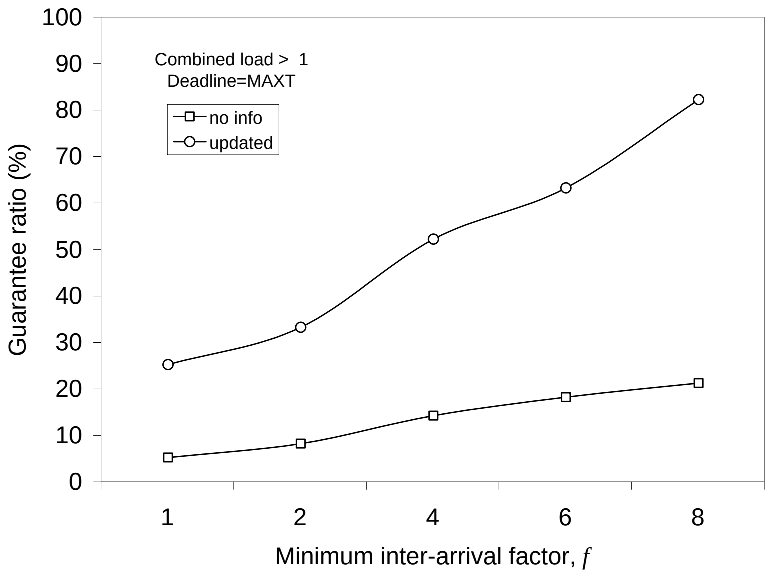


Figure A.8: Guarantee ratio for aperiodic tasks in the presence of sporadics tasks, dl=MAXT: variation of MINT

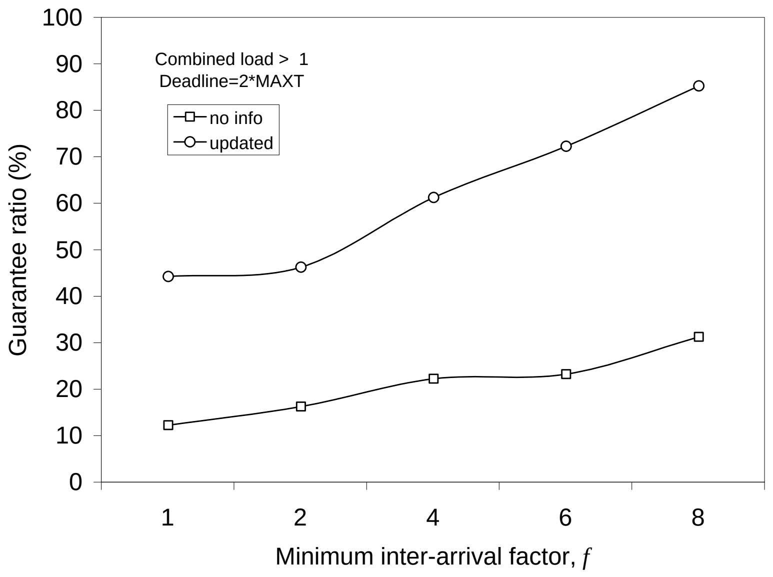


Figure A.9: Guarantee ratio for aperiodic tasks in the presence of sporadics tasks, dl=2*MAXT: variation of MINT

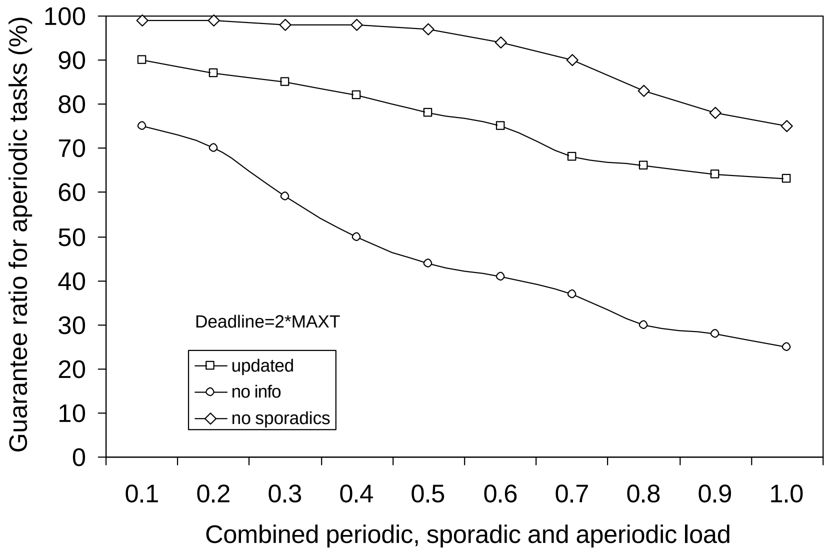


Figure A.10: Guarantee ratio for aperiodic tasks in the presence of sporadics tasks - Final results

Appendix B

Simulation results: Analysis of MPEG-2 Video Streams

We have analyzed a number of realistic MPEG streams to get a clear picture about MPEG video stream structure and processing. We used the analysis result to point out some common misconceptions about MPEG, and identify valid assumptions needed to propose a quality aware frame skipping algorithm based on realistic timing constraints for MPEG-2 processing. Here we present the analysis results.

B.1 Analysis setup

We have analysed the contents of original DVD movies. The movies were not encrypted or copy protected in any sense, which means that we managed to rip their context without breaking the CSS protection code on a DVD. Anyway, we chose not to publish any particular movie titles; instead we refer to them as action movie, drama movie, cartoon, etc.

Ripped MPEG streams were analysed by an in-house written piece of software, see appendix C for details. It took approximately 10 minutes to analyse a 100 minutes long MPEG stream on a PC computer with the processor speed of 1,5 GHz.

B.1.1 Analyzed video streams

An overview of the movies we have analyzed is summarized in table B.1. N and M refer to the GOP length and distance between reference frames respective, e.g. GOP(12,3) means I-to-I distance is 12, while I-to-P and P-to-P distance is 3.

Genre	Length	Fps	Resolution	Mbit/s	GOP
Action	118 min	25	720x576	9800	(12,3)
		30	352x240	1411	(18,3)
Drama	115 min	25	720x576	8700	(12,3)
		30	352x240	1411	(18,3)
Cartoon	104 min	25	720x576	6000	(12,3)
Thriller	106 min	30	720x480	9800	(12,3)
Sci-Fi	122 min	30	720x480	7500	(12,3)
Philharmonic	120 min	25	720x576	7000	(12,3)
Documentary	55 min	30	720x480	6500	(12,3)

Table B.1: Analyzed MPEG streams

The contents of the analysing movies is varying, i.e., we have analysed action movies with a lot of rapid motion, drama movies with slowly changing scenes, cartoons with less complex picture composition, documentaries, music videos, etc. That because some types of videos are more sensitive for frames skipping. For example, skipping 4 frames in an action video reduces half of the original video quality, 50%, while only 10% in a cartoon video [47].

Furthermore, we have looked into statistics for different resolutions for some of the analysed movies, 720x576 and 352x240.

B.1.2 Analyzed properties

In our analysis, we have looked into stream properties, frame sizes, GOP structure and decoding times for the frames. We have also analysed the distribution of the frame sizes. We have divided the range between minimum and maximum frame size for respective frame type into size intervals, and identified the number of frames in respective interval. In that way we can e.g., say that the majority of frames have bit size between some X and Y.

B.2 Analysis results

Action movie – Tables B.2 and B.10 summarize GOP and frame size properties for the movie. Minimum, maximum and average size is given in bits. Size distribution intervals are depicted in figure B.1. We have performed the same analysis for the same movie but with different resolution and frame rate, see tables B.3, B.11 and figure B.2.

Drama – The GOP and frame sizes for the drama movie are presented in table B.4. The GOP properties are described in table B.12 and the size distribution is shown in figure B.3.

Cartoon – The size data and GOP properties for the cartoon are presented in tables B.5 and B.13. The size distribution is shown in figure B.4.

Thriller – The size data and GOP properties for the thriller movie can be found in in tables B.6 and B.14. The size distribution is depicted in figure B.5.

Science Fiction – The GOP and frame sizes for the sci-fi movie are presented in table B.7. The GOP properties are described in table B.15 and the size distribution is shown in figure B.6.

Philharmonic Concert – The size data and GOP properties for the concert are presented in tables B.8 and B.16. The size distribution is shown in figure B.7.

Documentary – The GOP and frame sizes for the documentary movie are summarized in table B.9. The GOP properties are described in table B.17 and the size distribution is shown in figure B.8.

An summary of the analysis on the frame level for all the movies is presented in tables B.18 and B.19.

Sub-frame level We have also looked into stream properties at the sub-frame (macroblock) level for the action and the drama movie. The results are presented in figures B.9, B.10, B.11, B.12 and B.13.

B.3 Comments on analysis results

There is a significant variation in frame sizes arroung the average.

For each of the analysed movies we found that there is a significant variation in frame sizes arround the average value. For example, in the action movie, the size ratio between average values for respective frame type is I:P:B = 4:2:1, which means that on average I frames are twice as big as P frames, and 4 times bigger than B frames. However, this does not hold for a significant number of cases, which is depicted in table B.10. Also, from figure B.1 we can see that 88% of the I frames has bit size between 197737 and 790684 bits ($\approx$ 200 - 800 kB), which is a quite large interval. The assumptions about MPEG based on average frame size will not hold, since the significant number of frames will have twice as large respective twice as small bit size, compared to the average frame size (which is $\approx$ 500 kB).

The average frame ratio also depends on the movie content.

The average size ratio for the action movie is I:P:B = 4:2:1, while the ratio for the concert movie is 6:2:1. This because of a quite static background in the philharmonic concert movie which is not changed often, so the difference between current frame and the next one gets smaller. In other words, we need less bits for predicted frames.

The I frame is not necessarily the largest one in a GOP.

For example, in the thriller movie we have a case with 11% GOPs in which P have the largest size, and 1% of B frames, which corresponds roughly to 14 and 1.5 minutes, resp, in a 90 minute feature film. Furthermore we can see from table B.10 that frames in a GOP are not sorted according to their bit size, e.g., in 81% of the cases, the P frame that is closest to the I frame was not the largest among all P frames in the GOP.

The sequence structure in a GOP is not fixed to a specific I,P,B frame pattern.

In 28% of the GOPs in the drama movie the GOP length was not 12 frames. Not all GOPs consist of the same fixed number of P and B frames following the I frame in a fixed pattern. That is because more advanced encoders will attempt to optimize the placement of the three picture types according to local sequence characteristics in the context of more global characteristics. For instance scene changes or large changes in video content do not occur regularly, and hence the need for I frames in most video sequences is not at regular intervals.

All frames can be coded with different macroblock types.

All macroblocks within an I frame are coded as intra. However, macroblocks within a P frame may either be coded as intra or inter (temporally predicted from a previously reconstructed frame). Macroblocks in a B frame can be independently selected as either intra, forward-predicted, backward-predicted or both forward and backward predicted, see figures B.9 and B.10. We can also se from the figures that in the action movie, there are more intra macroblocks in P and B frames than for the drama movie. The reason is more motion and more frequent scene changes in the action movie; if the encoder cannot find a sufficiently similar block in the reference frames, it simply creates an I block.

We have used the findings above to identify a number of misconceptions about MPEG-2 and propose realistic assumptions, see chapter 6.2.

Item	Count	Minimum	Maximum	Average	Std deviation
I	16873	88	1976584	506114	187598
P	49679	16	1216000	234824	109888
B	112860	32	769048	148200	57616
GOP	16873	88	7541496	2222248	746768

Table B.2: Action movie, 720x576 - Bitsizes for frames and GOPs

Item	Count	Minimum	Maximum	Average	Std deviation
I	12645	24	366544	132232	48055
P	59933	24	261528	61880	22330
B	140041	24	168480	24416	12673
GOP	12645	192	1502856	696056	155401

Table B.3: Action movie, 354x240 - Bitsizes for frames and GOPs

Item	Count	Minimum	Maximum	Average	Std deviation
I	13716	136	1469848	471880	140329
P	52860	32	1009832	231144	76835
B	106478	32	636416	152432	45746
GOP	13716	136	7185768	2543520	627856

Table B.4: Drama movie, 720x576 - Bitsizes for frames and GOPs

Frame type	Nr of frames	Min	Max	Avg	Std dev
I	10139	57424	1121216	674544	216068
P	30406	1272	1097336	255552	133372
B	80861	1264	891240	115184	53982
GOP	10139	69712	4680200	2360792	622665

Table B.5: Cartoon - Bitsizes for frames and GOPs

Item	Count	Minimum	Maximum	Average	Std deviation
I	13404	2856	1282720	514744	174307
P	40088	32	1204808	281864	92779
B	98868	32	762048	129536	48890
GOP	13404	13312	5896728	2324672	583043

Table B.6: Thriller movie - Bitsizes for frames and GOPs

Item	Count	Minimum	Maximum	Average	Std deviation
I	14663	41104	760000	430088	70920
P	43920	1272	809016	249576	65226
B	117090	3184	664968	136720	41336
GOP	14667	76088	4322648	2269352	408220

Table B.7: Sci-Fi - Bitsizes for frames and GOPs

Item	Count	Minimum	Maximum	Average	Std deviation
I	14541	3432	1895088	1019896	363352
P	55248	32	1459952	396576	98782
B	110396	24	1565960	184912	51664
GOP	14541	8912	10635840	4000408	806103

Table B.8: Philharmonic Concert - Bitsizes for frames and GOPs

Item	Count	Minimum	Maximum	Average	Std deviation
I	8036	14392	819736	568824	116259
P	23929	32	764696	414144	48855
B	63747	32	423880	199928	26295
GOP	8036	80672	6412158	3396568	291106

Table B.9: Documentary movie - Bitsizes for frames and GOPs

GOP property	Number of GOPs	Percent
Open GOPs	12900	76%
Closed GOPs	3973	24%
GOPs with normal length (12)	12991	77%
Largest frame I	15061	89%
Largest frame P	1658	10%
Largest frame B	154	1%
GOPs where $P > I$	5256	31%
GOPs where $B > I$	4442	26%
GOPs where $B > P$	6545	39%
$P >$ some previous P in the GOP	13609	81%
$B >$ some previous B in the GOP	16326	97%

Table B.10: Action movie, 720x576 - GOP properties

GOP property	Number of GOPs	Percent
Open GOPs	12093	96%
Closed GOPs	552	4%
GOPs with normal length (18)	9074	72%
Largest frame I	11706	92%
Largest frame P	868	7%
Largest frame B	72	1%
GOPs where $P > I$	905	7%
GOPs where $B > I$	248	2%
GOPs where $B > P$	6029	48%
$P >$ some previous P in the GOP	11775	93%
$B >$ some previous B in the GOP	12464	98%

Table B.11: Action movie, 354x240 - GOP properties

GOP property	Number of GOPs	Percent
Open GOPs	13381	98%
Closed GOPs	335	2%
GOPs with normal length (12)	12573	92%
Largest frame I	12904	94%
Largest frame P	758	6%
Largest frame B	54	0,4%
GOPs where $P > I$	786	6%
GOPs where $B > I$	230	2%
GOPs where $B > P$	5072	37%
$P >$ some previous P in the GOP	11481	84%
$B >$ some previous B in the GOP	13715	100%

Table B.12: Drama movie, 720x576 - GOP properties

GOP property	Number of GOPs	Percent
Open GOPs	10123	100%
Closed GOPs	16	0,2%
GOPs with normal length (12)	10056	99%
Largest frame I	9291	92%
Largest frame P	842	8%
Largest frame B	14	0,1%
GOPs where $P > I$	841	8%
GOPs where $B > I$	79	1%
GOPs where $B > P$	1180	12%
$P >$ some previous P in the GOP	8260	81%
$B >$ some previous B in the GOP	10138	100%

Table B.13: Cartoon - GOP properties

GOP property	Number of GOPs	Percent
Open GOPs	11443	85%
Closed GOPs	1961	15%
GOPs with normal length (12)	11005	82%
Largest frame I	11874	89%
Largest frame P	1477	11%
Largest frame B	53	0%
GOPs where $P > I$	4253	32%
GOPs where $B > I$	2035	15%
GOPs where $B > P$	1112	8%
$P >$ some previous P in the GOP	9587	72%
$B >$ some previous B in the GOP	13264	99%

Table B.14: Thriller - GOP properties

GOP property	Number of GOPs	Percent
Open GOPs	14664	100%
Closed GOPs	23	0%
GOPs with normal length (12)	14595	100%
Largest frame I	13665	93%
Largest frame P	954	7%
Largest frame B	48	0%
GOPs where $P > I$	2453	17%
GOPs where $B > I$	449	3%
GOPs where $B > P$	1491	10%
$P >$ some previous P in the GOP	7424	51%
$B >$ some previous B in the GOP	14662	100%

Table B.15: Sci-Fi - GOP properties

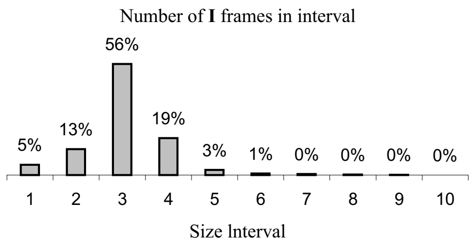
GOP property	Number of GOPs	Percent
Open GOPs	14322	98%
Closed GOPs	219	2%
GOPs with normal length (12)	12292	85%
Largest frame I	13402	92%
Largest frame P	1079	7%
Largest frame B	60	0%
GOPs where $P > I$	3897	27%
GOPs where $B > I$	2999	21%
GOPs where $B > P$	2206	15%
$P >$ some previous P in the GOP	13566	93%
$B >$ some previous B in the GOP	13996	96%

Table B.16: Philharmonic concert - GOP properties

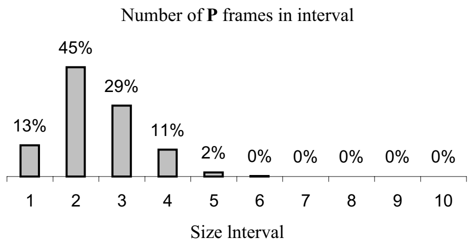
GOP property	Number of GOPs	Percent
Open GOPs	8020	100%
Closed GOPs	16	0%
GOPs with normal length (12)	7674	95%
Largest frame I	7672	95%
Largest frame P	357	4%
Largest frame B	7	0%
GOPs where $P > I$	1317	16%
GOPs where $B > I$	1532	19%
GOPs where $B > P$	333	4%
$P >$ some previous P in the GOP	5872	73%
$B >$ some previous B in the GOP	7997	100%

Table B.17: Documentary - GOP properties

Interval	From	To	Nr of I	Percent
1	88	197737	876	5,2%
2	197737	395386	2190	13,0%
3	395386	593035	9410	55,8%
4	593035	790684	3137	18,6%
5	790684	988333	426	2,5%
6	988333	1185982	129	0,8%
7	1185982	1383631	79	0,5%
8	1383631	1581280	51	0,3%
9	1581280	1778929	23	0,1%
10	1778929	1976584	6	0,0%



Interval	From	To	Nr of P	Percent
1	16	121614	6377	12,8%
2	121614	243212	22355	45,0%
3	243212	364810	14460	29,1%
4	364810	486408	5496	11,1%
5	486408	608006	857	1,7%
6	608006	729604	102	0,2%
7	729604	851202	17	0,0%
8	851202	972800	11	0,0%
9	972800	1094398	3	0,0%
10	1094398	1216000	1	0,0%



Interval	From	To	Nr of B	Percent
1	32	76933	7365	6,5%
2	76933	153834	59938	53,1%
3	153834	230735	35827	31,7%
4	230735	307636	8370	7,4%
5	307636	384537	1195	1,1%
6	384537	461438	113	0,1%
7	461438	538339	32	0,0%
8	538339	615240	13	0,0%
9	615240	692141	3	0,0%
10	692141	769048	2	0,0%

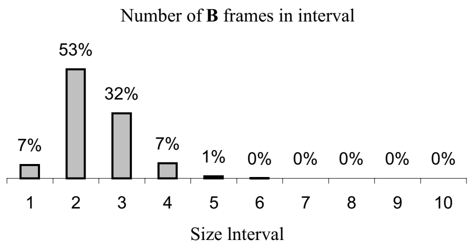
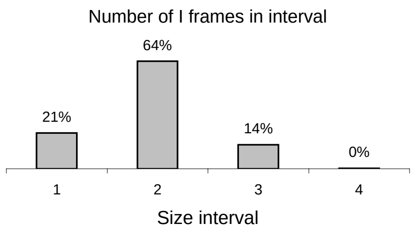
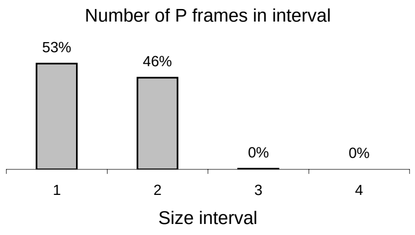


Figure B.1: Action movie, 720x576 - Size distribution for I, P and B frames

Interval	From	To	Nr of I	%
1	0	91636	2692	21%
2	91636	183272	8098	64%
3	183272	274908	1812	14%
4	274908	366544	42	0%



Interval	From	To	Nr of P	%
1	0	65382	31998	53%
2	65382	130764	27689	46%
3	130764	196146	224	0%
4	196146	261528	22	0%



Interval	From	To	Nrof B	%
1	0	42120	125669	90%
2	42120	84240	14314	10%
3	84240	126360	54	0%
4	126360	168480	4	0%

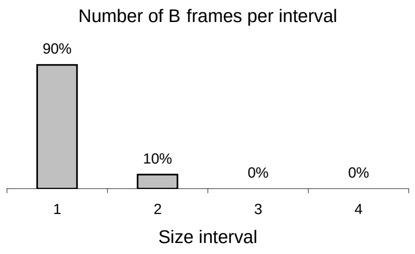
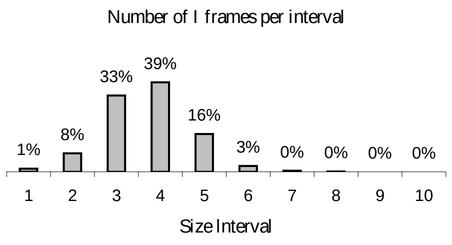
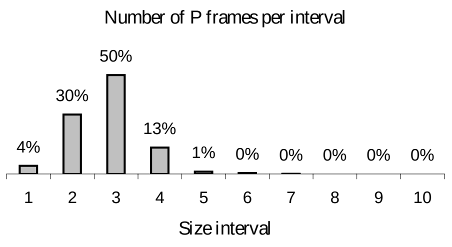


Figure B.2: Action movie, 354x240 - Size distribution for I, P and B frames

Interval	From	To	Nr of I	Percent
1	136	146985	188	1.4%
2	146985	293970	1089	7.9%
3	293970	440955	4503	32.8%
4	440955	587940	5281	38.5%
5	587940	734925	2231	16.3%
6	734925	881910	344	2.5%
7	881910	1028895	63	0.5%
8	1028895	1175880	14	0.1%
9	1175880	1322865	1	0.0%
10	1322865	1469850	2	0.0%



Interval	From	To	Nr of P	Percent
1	32	100984	2206	4.2%
2	100984	201968	15999	30.3%
3	201968	302952	26630	50.4%
4	302952	403936	7098	13.4%
5	403936	504920	618	1.2%
6	504920	605904	252	0.5%
7	605904	706888	35	0.1%
8	706888	807872	3	0.0%
9	807872	908856	0	0.0%
10	908856	1009840	2	0.0%



Interval	From	To	Nr of B	Percent
1	32	63642	1735	1.6%
2	63642	127284	30893	29.0%
3	127284	190926	53692	50.4%
4	190926	254568	18191	17.1%
5	254568	318210	1827	1.7%
6	318210	381852	219	0.2%
7	381852	445494	64	0.1%
8	445494	509136	31	0.0%
9	509136	572778	15	0.0%
10	572778	636420	14	0.0%

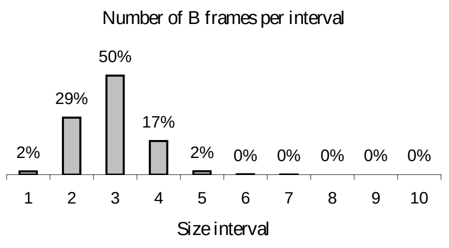
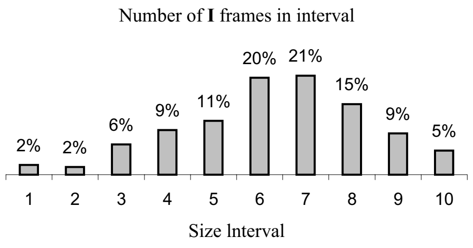
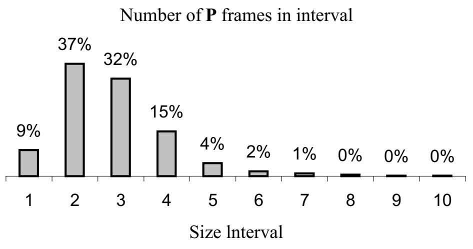


Figure B.3: Drama movie, 720x576 - Size distribution for I, P and B frames

Interval	From	To	Nr of I	Percent
1	57424	163803	207	2,0%
2	163803	270182	165	1,6%
3	270182	376561	643	6,3%
4	376561	482940	948	9,4%
5	482940	589319	1140	11,2%
6	589319	695698	2056	20,3%
7	695698	802077	2098	20,7%
8	802077	908456	1494	14,7%
9	908456	1014835	878	8,7%
10	1014835	1121216	510	5,0%



Interval	From	To	Nr of P	Percent
1	1272	110878	2616	8,6%
2	110878	220484	11245	37,0%
3	220484	330090	9768	32,1%
4	330090	439696	4473	14,7%
5	439696	549302	1298	4,3%
6	549302	658908	478	1,6%
7	658908	768514	279	0,9%
8	768514	878120	141	0,5%
9	878120	987726	63	0,2%
10	987726	1097336	45	0,1%



Interval	From	To	Nr of B	Percent
1	1264	90261	29767	36,8%
2	90261	179258	41449	51,3%
3	179258	268255	8605	10,6%
4	268255	357252	941	1,2%
5	357252	446249	83	0,1%
6	446249	535246	9	0,0%
7	535246	624243	5	0,0%
8	624243	713240	0	0,0%
9	713240	802237	1	0,0%
10	802237	891240	1	0,0%

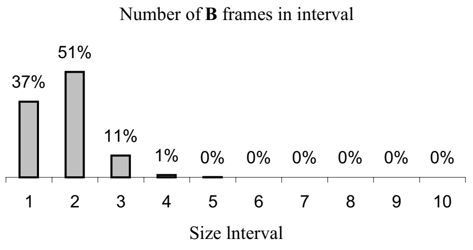
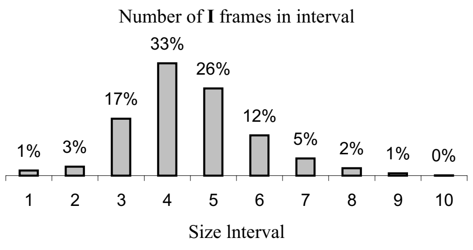
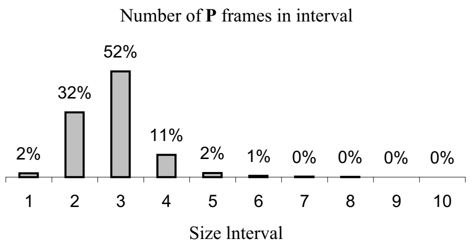


Figure B.4: Cartoon - Size distribution for I, P and B frames

Interval	From	To	Nr of I	Percent
1	2856	130842	198	1,5%
2	130842	258828	356	2,7%
3	258828	386814	2238	16,7%
4	386814	514800	4408	32,9%
5	514800	642786	3422	25,5%
6	642786	770772	1579	11,8%
7	770772	898758	675	5,0%
8	898758	1026744	289	2,2%
9	1026744	1154730	88	0,7%
10	1154730	1282720	12	0,1%



Interval	From	To	Nr of P	Percent
1	32	120509	752	1,9%
2	120509	240986	12809	32,0%
3	240986	361463	20900	52,1%
4	361463	481940	4428	11,0%
5	481940	602417	793	2,0%
6	602417	722894	253	0,6%
7	722894	843371	103	0,3%
8	843371	963848	33	0,1%
9	963848	1084325	13	0,0%
10	1084325	1204808	4	0,0%



Interval	From	To	Nr of B	Percent
1	32	76233	8255	8,3%
2	76233	152434	66744	67,5%
3	152434	228635	21015	21,3%
4	228635	304836	2016	2,0%
5	304836	381037	439	0,4%
6	381037	457238	161	0,2%
7	457238	533439	202	0,2%
8	533439	609640	25	0,0%
9	609640	685841	7	0,0%
10	685841	762048	3	0,0%

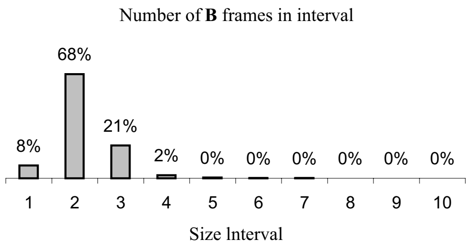
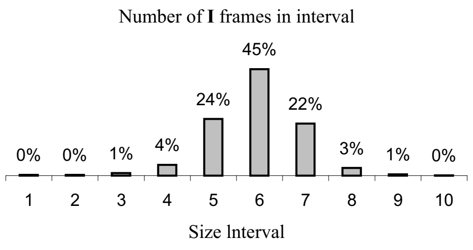
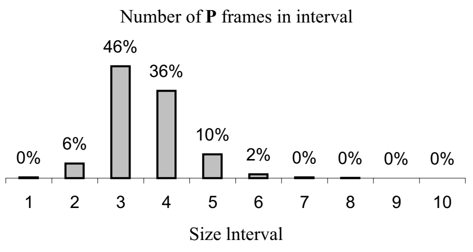


Figure B.5: Thriller movie - Size distribution for I, P and B frames

Interval	From	To	Nr of I	Percent
1	41104	112993	53	0,4%
2	112993	184882	42	0,3%
3	184882	256771	156	1,1%
4	256771	328660	653	4,5%
5	328660	400549	3471	23,7%
6	400549	472438	6529	44,5%
7	472438	544327	3197	21,8%
8	544327	616216	474	3,2%
9	616216	688105	80	0,5%
10	688105	760000	8	0,1%



Interval	From	To	Nr of P	Percent
1	1272	82046	159	0,4%
2	82046	162820	2618	6,0%
3	162820	243594	20183	46,0%
4	243594	324368	15747	35,9%
5	324368	405142	4294	9,8%
6	405142	485916	707	1,6%
7	485916	566690	150	0,3%
8	566690	647464	43	0,1%
9	647464	728238	14	0,0%
10	728238	809016	5	0,0%



Interval	From	To	Nr of B	Percent
1	3184	69362	2967	2,5%
2	69362	135540	65311	55,8%
3	135540	201718	39890	34,1%
4	201718	267896	7782	6,6%
5	267896	334074	966	0,8%
6	334074	400252	127	0,1%
7	400252	466430	28	0,0%
8	466430	532608	9	0,0%
9	532608	598786	5	0,0%
10	598786	664968	3	0,0%

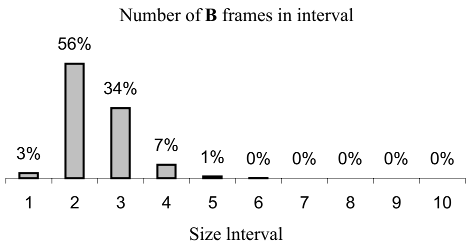
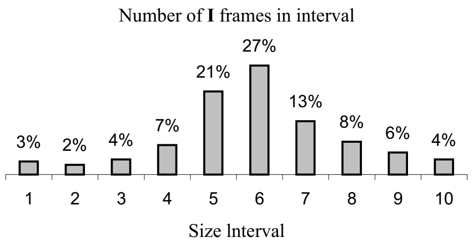
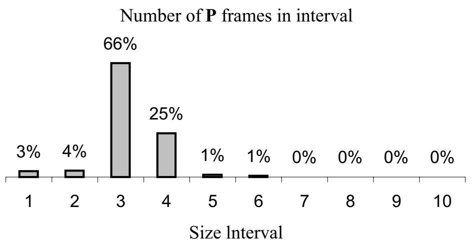


Figure B.6: Sci-Fi movie - Size distribution for I, P and B frames

Interval	From	To	Nr of I	Percent
1	3432	192597	474	3,3%
2	192597	381762	358	2,5%
3	381762	570927	550	3,8%
4	570927	760092	1073	7,4%
5	760092	949257	3036	20,9%
6	949257	1138422	3978	27,4%
7	1138422	1327587	1945	13,4%
8	1327587	1516752	1200	8,3%
9	1516752	1705917	802	5,5%
10	1705917	1895088	551	3,8%



Interval	From	To	Nr of P	Percent
1	32	146024	1866	3,4%
2	146024	292016	2059	3,7%
3	292016	438008	36194	65,5%
4	438008	584000	13921	25,2%
5	584000	729992	762	1,4%
6	729992	875984	415	0,8%
7	875984	1021976	27	0,0%
8	1021976	1167968	2	0,0%
9	1167968	1313960	0	0,0%
10	1313960	1459952	2	0,0%



Interval	From	To	Nr of B	Percent
1	24	156617	24255	22,0%
2	156617	313210	85485	77,4%
3	313210	469803	561	0,5%
4	469803	626396	73	0,1%
5	626396	782989	5	0,0%
6	782989	939582	4	0,0%
7	939582	1096175	3	0,0%
8	1096175	1252768	6	0,0%
9	1252768	1409361	0	0,0%
10	1409361	1565960	3	0,0%

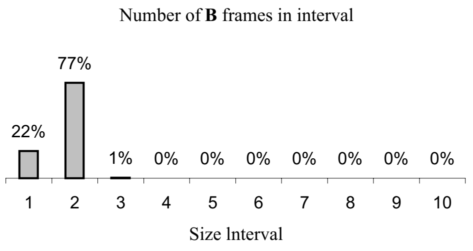
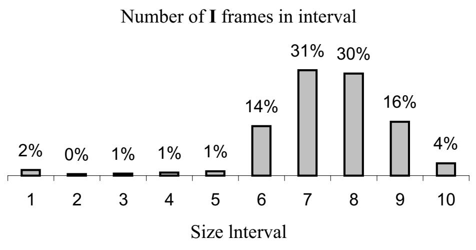
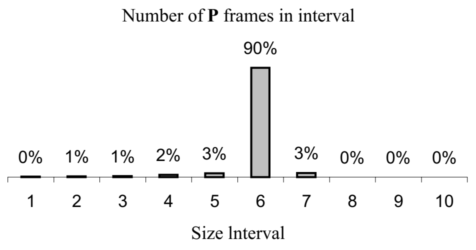


Figure B.7: Philharmonic - Size distribution for I, P and B frames

Interval	From	To	Nr of I	Percent
1	14392	94926	134	1,7%
2	94926	175460	36	0,4%
3	175460	255994	51	0,6%
4	255994	336528	71	0,9%
5	336528	417062	102	1,3%
6	417062	497596	1165	14,5%
7	497596	578130	2480	30,9%
8	578130	658664	2403	29,9%
9	658664	739198	1267	15,8%
10	739198	819736	289	3,6%



Interval	From	To	Nr of P	Percent
1	32	76498	104	0,4%
2	76498	152964	127	0,5%
3	152964	229430	180	0,8%
4	229430	305896	446	1,9%
5	305896	382362	759	3,2%
6	382362	458828	21526	90,0%
7	458828	535294	781	3,3%
8	535294	611760	1	0,0%
9	611760	688226	3	0,0%
10	688226	764696	2	0,0%



Interval	From	To	Nr of B	Percent
1	32	42416	345	0,5%
2	42416	84800	262	0,4%
3	84800	127184	707	1,1%
4	127184	169568	1924	3,0%
5	169568	211952	42148	66,1%
6	211952	254336	18096	28,4%
7	254336	296720	200	0,3%
8	296720	339104	57	0,1%
9	339104	381488	4	0,0%
10	381488	423880	3	0,0%

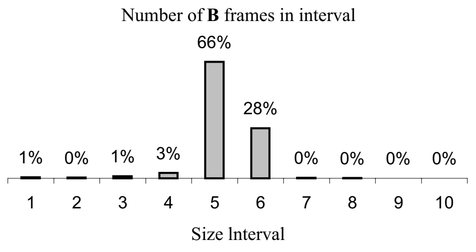


Figure B.8: Documentary - Size distribution for I, P and B frames

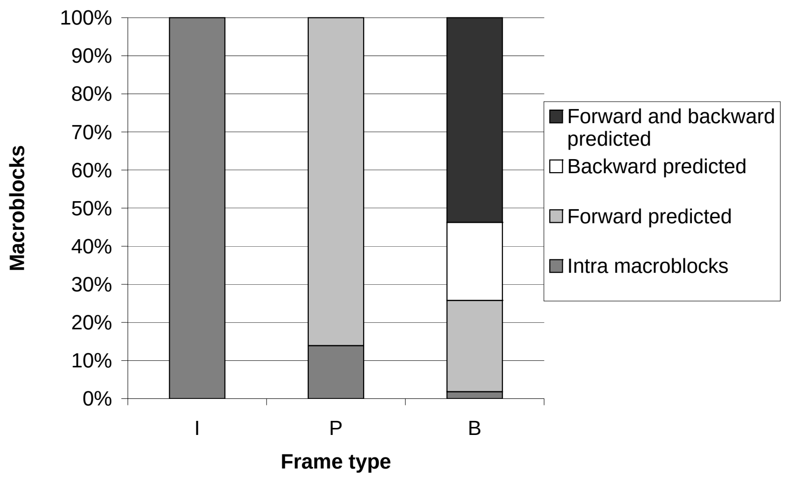


Figure B.9: Action movie - Macroblock types

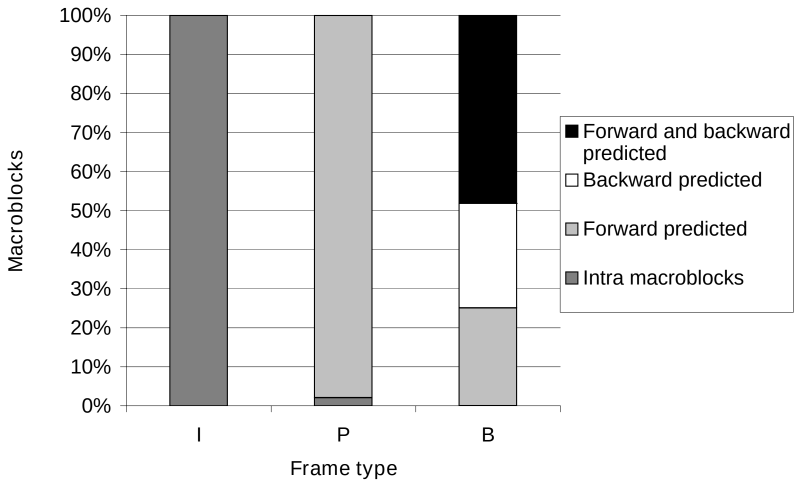
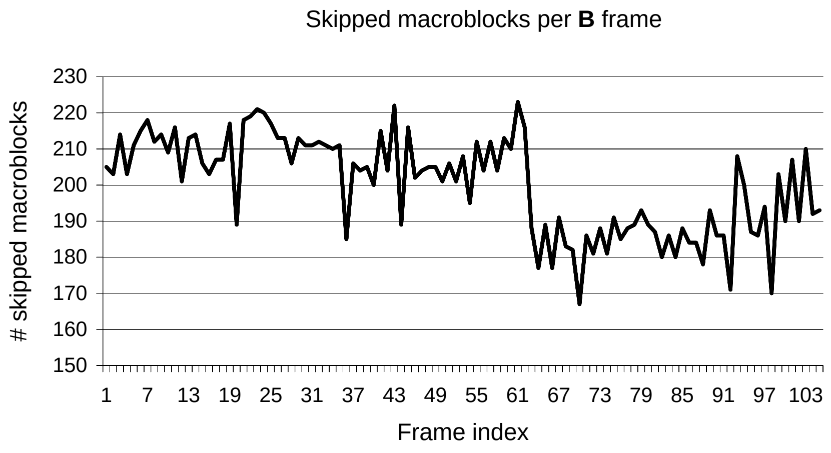


Figure B.10: Drama movie - Macroblock types



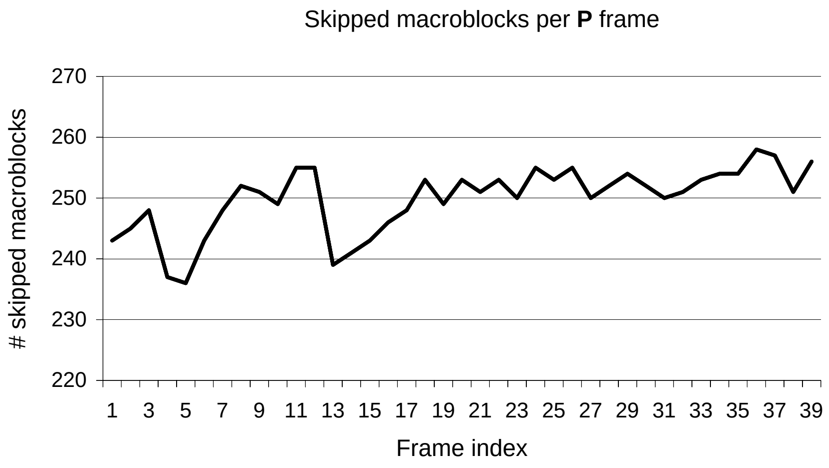


Figure B.11: Action movie - Skipped macroblocks variations

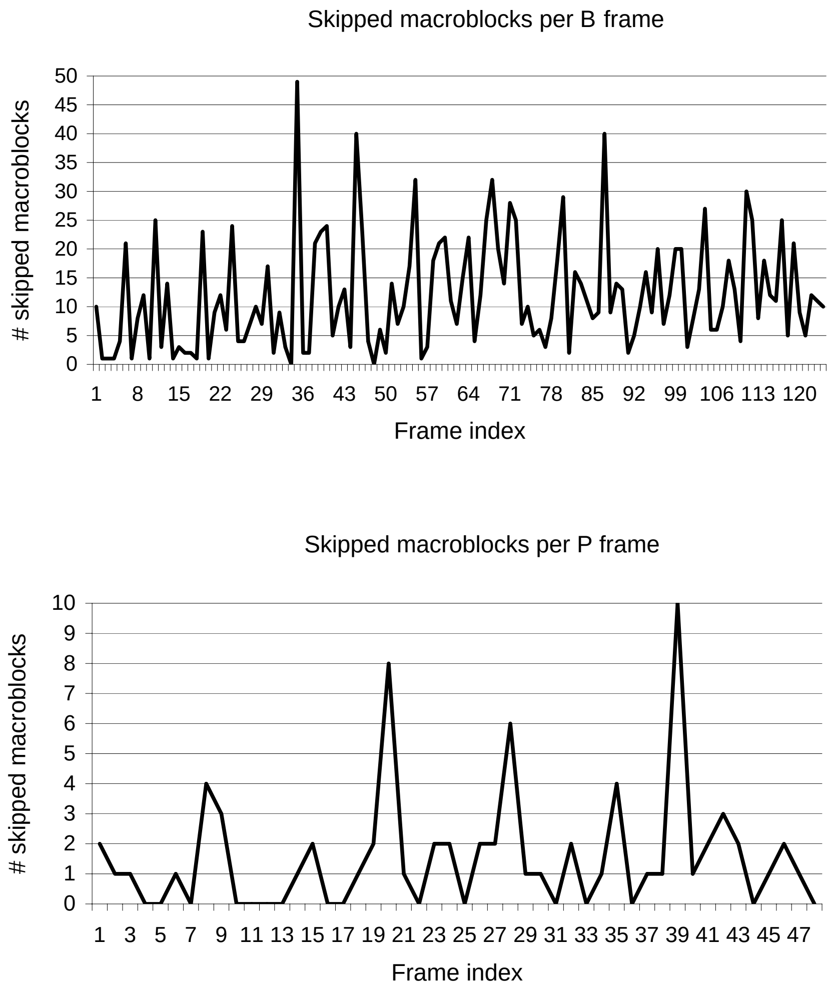
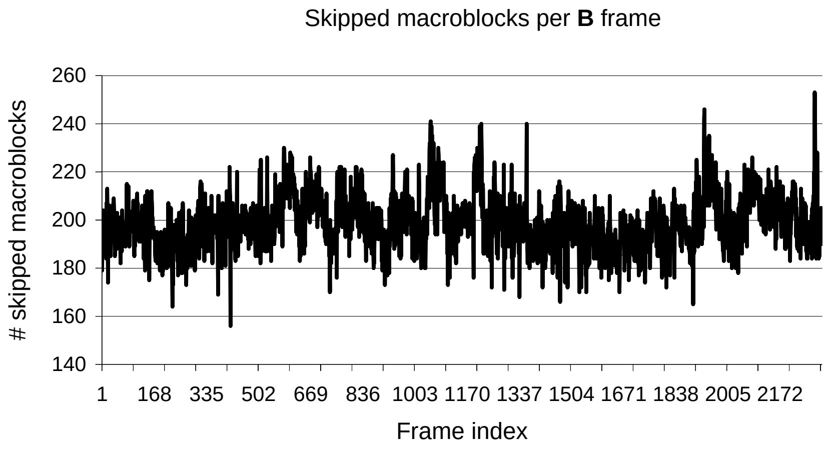


Figure B.12: Drama movie - Skipped macroblocks variations



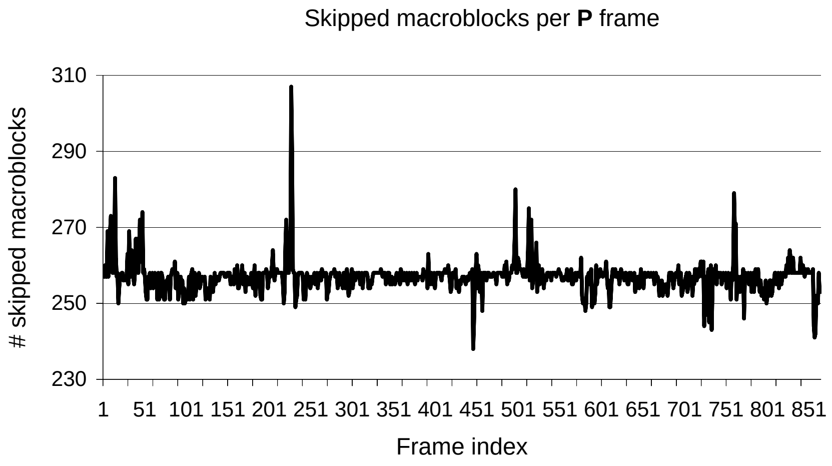


Figure B.13: Sci-Fi movie - Skipped macroblocks variations

Movie	avg	I frames		P frames		B frames	
	I:P:B	avg	std dev	avg	std dev	avg	std dev
Action	4:2:1	506114	187598	234824	109888	148200	57616
Drama	6:3:2	471880	140329	231144	76835	152432	45746
Cartoon	6:2:1	674544	216068	255552	133372	115184	53982
Thriller	4:2:1	514744	174307	281864	92779	129536	48890
Sci-Fi	3:2:1	430088	70920	249576	65226	136720	41336
Philharm.	6:2:1	1019896	363358	396576	98782	184912	51664
Docum.	3:2:1	568824	116259	414144	48855	199928	26295

Table B.18: Comparrison of bitsize properties for the analysed movies

Movie title	Number of GOPs where					
	I largest	P largest	B largest	$P > I$	$B > I$	$B > P$
Action	89%	10%	1%	31%	26%	39%
Drama	94%	5%	1%	6%	2%	37%
Cartoon	91%	8%	1%	8%	1%	12%
Thriller	88%	11%	1%	32%	15%	8%
Sci-Fi	93%	7%	0%	17%	3%	10%
Philharmonic	92%	7%	0%	27%	21%	15%
Documentary	95%	4%	0%	16%	19%	4%

Table B.19: Comparisson of GOP properties for the analysed movies

Appendix C

Implemented Tools

Here we describe all the tools that we have implemented and used for the simulation purposes of the algorithms presented in this thesis. We present both a set of real-time schedule design tools, which we used for the verification and simulation of the algorithm presented in chapters 2 and 3, and a set of MPEG analysis tools, needed for simulations performed in chapter 6.

Some of the tools have been implemented by myself while the others by other members of our research group (SALSART). The tools implemented by other people will be briefly described and properly referred, and all the tools implemented by myself will be described in detail.

C.1 Real-time Schedule Design Tools

SALSART toolset is a web-based cooperative environment for the design of real-time schedules, designed and implemented by our research group. It comprises a set of stand alone tools interacting via an internet based central supervisor. It envisions a set of experts working as a geographically separated team on application specification, scheduling, editing, simulation and analysis of real-time schedules.

The *SALSART* interactive tools are implemented in JAVA for platform independence and using XML for interfacing. Thus, it is able to be applied on a variety of systems and provides for configurable application demands.

The toolset consists of a *precedence graph editor*, a *scheduler*, a *schedule editor*, a *simulator* with random sporadic and aperiodic task generators, and a *supervisor*, a central server that connects all the tools. More detailed description of all *SALSART* tools can be found in [31].

C.1.1 Precedence Graph Editor

Implemented by: Roger Vuolle[1]

The Precedence Graph Editor (PG editor) is an application used to produce and edit precedence graphs to be scheduled with some offline scheduler. It provides for intuitive creation, modification and maintenance of precedence graphs, see figure C.1.

Precedence graphs are created from scratch in a graphical drag-and-drop environment. The output is saved in XML format, providing for easy distribution between different applications.

We have used Precedence Graph Editor to create specification files for the offline slot shifting scheduler that produces static schedules.

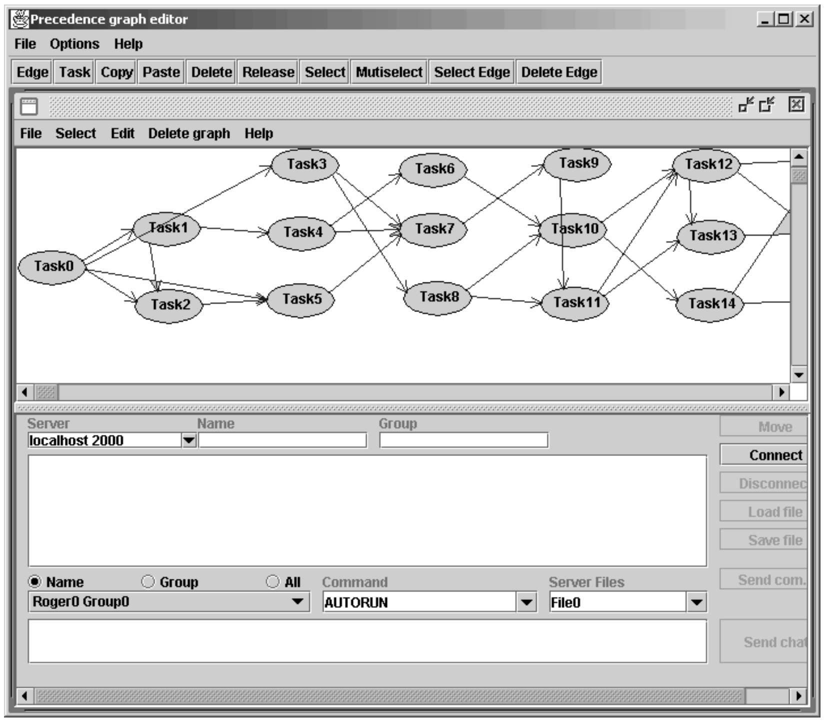
C.1.2 Scheduler

Implemented by: Roger Vuolle and Tomas Lennvall[2]

The offline scheduler reads the precedence graphs created by the PG editor and creates static schedules with intervals and spare capacities, as described in chapter 2.2. It allocates tasks to nodes and constructs tables outlining the temporal execution of tasks in a feasible way (see figure C.2). All complex constraints are resolved offline, providing for very simple runtime mechanisms.

[1]Roger Vuolle is a former student of MdH. PG editor is a part of his MSc thesis which he did under supervison of our research group

[2]Tomas Lennvall is a Ph D student at MdH, a member of our research group



Server: localhost 2000
Name:
Group:

[✓ Name] Group All: Roger0 Group0
Command: AUTORUN
Server Files: File0

Figure C.1: Precedence Graph Editor screenshot

The scheduling algorithm is based on the *simulated annealing* technique. The simulated annealing algorithm is a global optimization technique, which attempts to find the lowest energy point in a world or landscape of energy points. When the algorithm is used to find allocation and execution schedules, every point in the world represents an energy level, an allocation and a schedule for all the instances of tasks in the system, at the same time. This energy level is calculated for every new state the process stops at and the energy level is the sum of all allocation and scheduling constraint violations in the system. The neighbor space of a point is the set of all points that are reachable by moving a single task to another processor node or by moving the task in the time

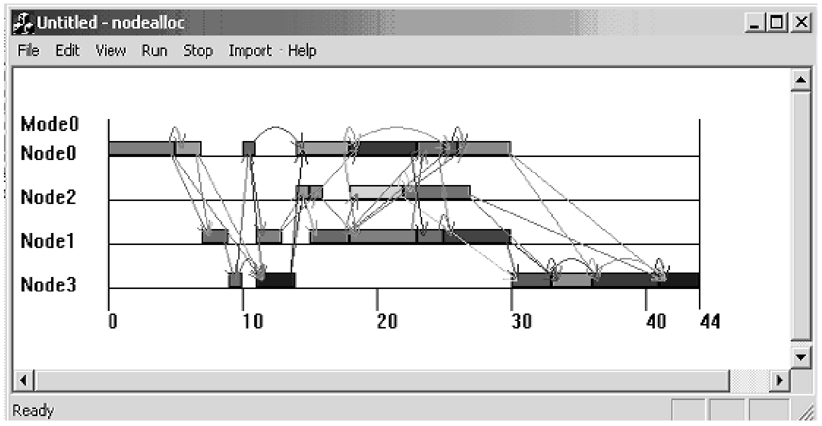


Figure C.2: Scheduler screenshot

space available on some allocated processor. This means that the energy calculated for some point in the world represents the suitability of the allocation and the schedule for the task instances involved at all the processors in the system.

Since the result of the scheduler are scheduling tables or an indication that no schedule was found, it does not provide visualization itself.

C.1.3 Schedule Editor

Implemented by: Damir Isovic

Sometimes we do not want the scheduler to make all allocation decisions. Instead we would like to view the schedule and optimize it by hand, e.g. we want a task to be scheduled on node 2 instead of node 1. The schedule editor (see figure C.3) is an application used for viewing and editing of real-time schedules. The basic editing features, such as moving, deleting, copying and pasting new objects, are extended with additional functionality and constraints that allow advanced but controlled verification and modification of RT schedules. The schedule editor can be executed as a stand-alone application, but it is also able to establish an online cooperation with the other parts of the SALSART tool, making the editing highly distributed.

Interfacing

Input to the Schedule Editor are offline schedule files in XML format, created by some offline scheduler. The editor can easily be configured to get input from any static scheduler, assuming that the attributes to the schedule components are separated from the schedule file. Therefore this additional information, i.e., which tasks belong to which precedence graph, deadlines, colors, descriptions, resources etc. , is kept in separate files rather than in the schedule file. The more information provided the more editor features become available. This way, the schedule editor can utilize the additional information when it is supplied, which eliminates all dependencies upon the file output formats of static schedulers.

The following file formats are recognized by the schedule editor:

- Basic schedule file – contains the minimum amount of information needed for graphical display of a schedule. The outputs from some standard schedulers are supported without any modification. The unsupported ones can easily be converted into this format - the only information needed is start and completion times of the tasks as well as their allocation nodes. This file is necessary for schedule visualization.

- Attribute file – contains the additional information about a schedule, such as intervals, spare capacities, deadlines, WCET, descriptions of the tasks. This file is optional.

- Network file - contains the information about the inter-node messages between tasks. This file is also optional.

The Editor can read three different files describing the same schedule, though it's enough to have only one of them to display the schedule.

However, the more information available, the better the visualization of the system.

Separating the actual schedule file from additional information about the schedule components makes the application configurable for adding new task models and outputs from various schedulers. Modified schedules are saved in the same format they had before editing.

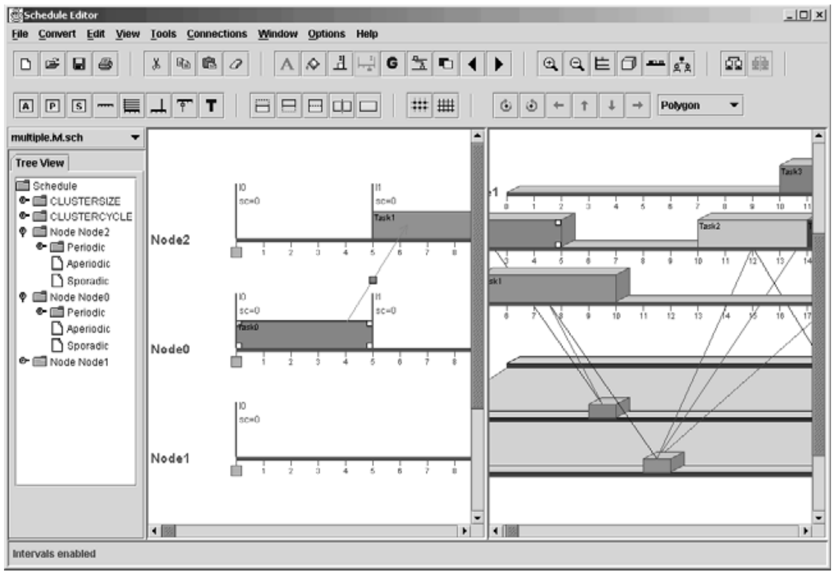


Figure C.3: Schedule Editor screenshot

Visualization

The least amount of information needed for the Editor to work properly is a schedule file. If no additional information is provided, the schedule file is read and displayed both graphically and as a tree structure. Several different views of the schedule are available.

- Tree view – schedule components are viewed as a tree, allowing easy searches.

- Node view – graphical display with nodes and belonging tasks.
- 3D view – sometimes it is difficult to distinguish between many inter-node messages in 2D view, due to the overlaps when drawing those. In 3D views it's much easier to follow a certain message. In this mode schedules can be rotated as well.

Functionality

Schedule editing can be performed with and without constraints on tasks. Editing with constraints supports the slot-shifting algorithm, enabling the warning messages for violated constraints. The schedule can also be edited freely, without any constraints and warning messages, which gives the user total control over the construction. On the other hand, this approach can result in unrealistic schedules due to lack of restrictions in allocation of the resources.

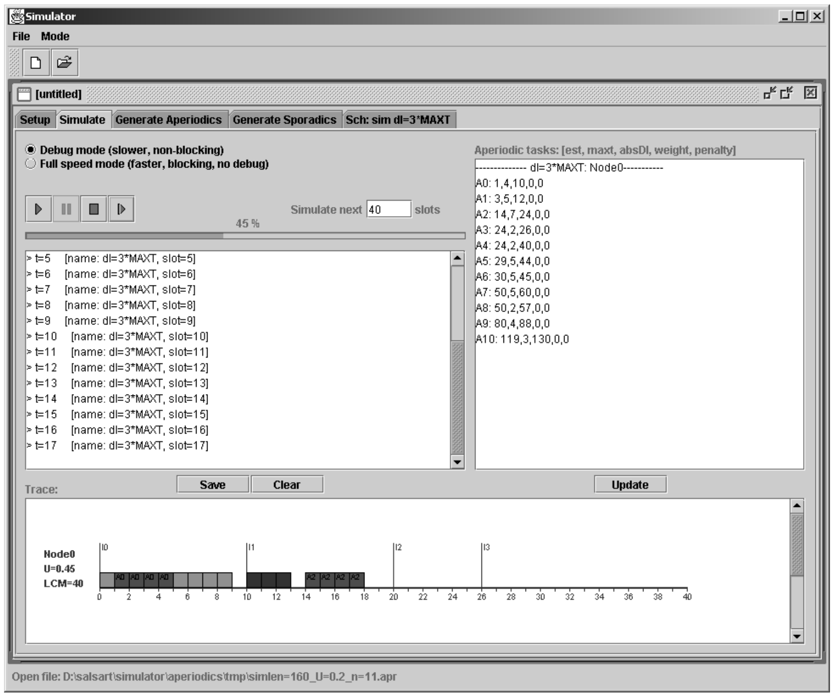
The Schedule Editor can be connected to a real application in order to monitor the run-time behavior of the systems running created schedules, i.e., we can get a signal back to the editor when a task has completed, or a new aperiodic task has arrived. The connection is done via a server, *supervisor* that could be running anywhere in the world. The Schedule Editor is already prepared for the connection to the supervisor, and the monitored application can connect by using a simple monitor client, also available as a part of the SALSART tools. The only restriction on the application is that it has to have support for socket communication.

C.1.4 Schedule Simulator

Implemented by: Damir Isovic

The Simulator (figure C.4) is used for simulation of various guarantee algorithms for firm aperiodic tasks. It can be used to graphically display what happens during the execution of a schedule using a chosen method, displaying the run-time activities and events that occur during execution.

It also logs these events and generates statistics that, for instance, can be used for various kinds of performance analyses. Being part of the SAL-SART suite the simulator supports distributed cooperative team-work, enabling a group of people to work together on a project.



[✓ Debug mode (slower, non-blocking)]
Full speed mode (faster, blocking, no debug)

Simulate next 40 slots

45 %

```
> t=5    [name: dl=3*MAXT, slot=5]
> t=6    [name: dl=3*MAXT, slot=6]
> t=7    [name: dl=3*MAXT, slot=7]
> t=8    [name: dl=3*MAXT, slot=8]
> t=9    [name: dl=3*MAXT, slot=9]
> t=10   [name: dl=3*MAXT, slot=10]
> t=11   [name: dl=3*MAXT, slot=11]
> t=12   [name: dl=3*MAXT, slot=12]
> t=13   [name: dl=3*MAXT, slot=13]
> t=14   [name: dl=3*MAXT, slot=14]
> t=15   [name: dl=3*MAXT, slot=15]
> t=16   [name: dl=3*MAXT, slot=16]
> t=17   [name: dl=3*MAXT, slot=17]
```

Aperiodic tasks: [est, maxt, absDl, weight, penalty]

```
------------- dl=3*MAXT: Node0-----------
A0: 1,4,10,0,0
A1: 3,5,12,0,0
A2: 14,7,24,0,0
A3: 24,2,26,0,0
A4: 24,2,40,0,0
A5: 29,5,44,0,0
A6: 30,5,45,0,0
A7: 50,5,60,0,0
A8: 50,2,57,0,0
A9: 80,4,88,0,0
A10: 119,3,130,0,0
```

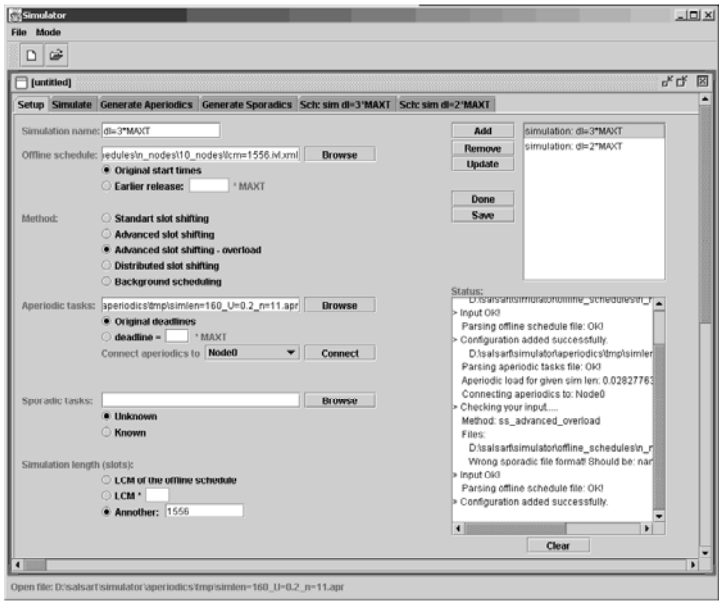
Node0
U=0.45
LCM=40

Open file: D:\salsart\simulator\aperiodics\tmp\simlen=160_U=0.2_n=11.apr

Figure C.4: Simulator - debug mode

Methods

Several guarantee algorithms for firm aperiodic tasks based on slot shifting are currently supported, including the algorithms in chapters 2 and 3, see figure C.5.



Simulation name: dl=3*MAXT

Offline schedule: edules\n_nodes\10_nodes\lcm=1556.lvl.xml

[✓ Original start times] Earlier release: * MAXT

Method: Standart slot shifting / Advanced slot shifting / [✓ Advanced slot shifting - overload] / Distributed slot shifting / Background scheduling

Aperiodic tasks: aperiodics\tmp\simlen=160_U=0.2_n=11.apr

[✓ Original deadlines] deadline = * MAXT

Connect aperiodics to Node0

Sporadic tasks:

[✓ Unknown] Known

Simulation length (slots): LCM of the offline schedule / LCM * / [✓ Annother: 1556]

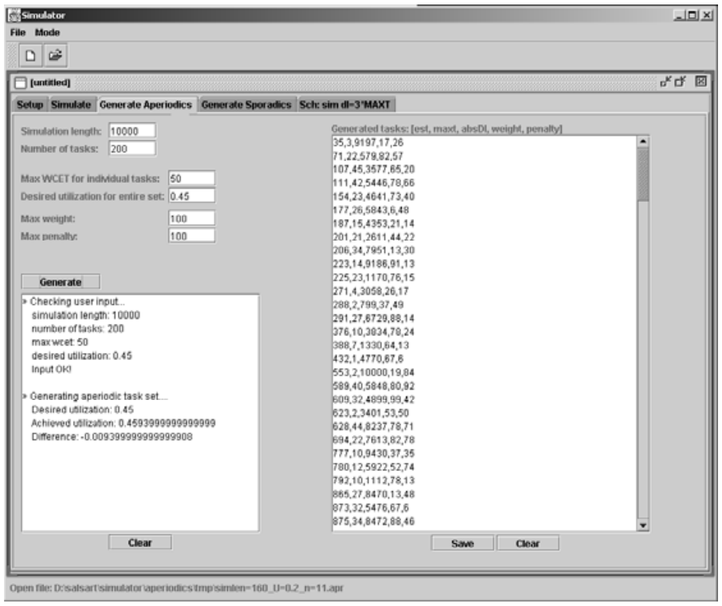
simulation: dl=3*MAXT
simulation: dl=2*MAXT

Status:

```
    D:\salsart\simulator\offline_schedules\n_r
> Input OK!
  Parsing offline schedule file: OK!
> Configuration added successfully.
    D:\salsart\simulator\aperiodics\tmp\simlen
  Parsing aperiodic tasks file: OK!
  Aperiodic load for given sim len: 0.02827763
  Connecting aperiodics to: Node0
> Checking your input....
  Method: ss_advanced_overload
  Files:
    D:\salsart\simulator\offline_schedules\n_r
    Wrong sporadic file format! Should be: nar
> Input OK!
  Parsing offline schedule file: OK!
> Configuration added successfully.
```

Open file: D:\salsart\simulator\aperiodics\tmp\simlen=160_U=0.2_n=11.apr

Figure C.5: Simulator - config mode, supported algorithms



Simulation length: 10000
Number of tasks: 200

Max WCET for individual tasks: 50
Desired utilization for entire set: 0.45

Max weight: 100
Max penalty: 100

```
> Checking user input...
   simulation length: 10000
   number of tasks: 200
   max wcet: 50
   desired utilization: 0.45
   Input OK!

> Generating aperiodic task set...
   Desired utilization: 0.45
   Achieved utilization: 0.4593999999999999
   Difference: -0.009399999999999908
```

Generated tasks: [est, maxt, absDl, weight, penalty]

```
35,3,9197,17,26
71,22,579,82,57
107,45,3577,65,20
111,42,5446,78,66
154,23,4641,73,40
177,26,5843,6,48
187,15,4353,21,14
201,21,2611,44,22
206,34,7951,13,30
223,14,9186,91,13
225,23,1170,76,15
271,4,3058,26,17
288,2,799,37,49
291,27,6729,88,14
376,10,3834,78,24
388,7,1330,64,13
432,1,4770,67,6
553,2,10000,19,84
589,40,5848,80,92
609,32,4899,99,42
623,2,3401,53,50
628,44,8237,78,71
694,22,7613,82,78
777,10,9430,37,35
780,12,5922,52,74
792,10,1112,78,13
865,27,8470,13,48
873,32,5476,67,6
875,34,8472,88,46
```

Open file: D:\salsart\simulator\aperiodics\tmp\simlen=160_U=0.2_n=11.apr

Figure C.6: Simulator - random task generator

Visualization

The simulator supports a number of different views; each view showing distinct kinds of information and data to graphically display the inner workings and stages of a simulated algorithm. These views display the following:

- The actual schedule with tasks, inter-node messages, intervals, spare capacities, etc.
- The set of tasks that are ready to execute.
- Aperiodic task queues; both soft and firm.
- Sporadic tasks queues

Statistics and information

As well as performing the actual simulation the simulator also logs the simulation, gathering event data and generating statistics. This information can be used to analyse different aspects of a schedule. Some examples of such data are the aperiodic task load, aperiodic response times, guarantee ratios, and deadline violations.

Random task generators

The simulator includes random task generators for aperiodic and sporadic task arrivals, see figure C.6. The generated task sets can be tuned by specifying attributes for the tasks, such as earliest start time, deadline and WCET intervals.

C.1.5 Example: operational scenario

We envision the following setup for the use of presented scheduling design tools. A schedule could be designed as follows:

- PG Editor: application designer specifies the relations between tasks and their temporal constraints, including precedence, mutual exclusion, end-to-end deadlines in distributed systems, preallocation of some tasks to node in the systems.

- Scheduler: takes the application specification and applies a scheduling tool to construct an offline schedule. In the failure case new attempts with different settings, e.g., more comprehensive search, different heuristics are initiated. If these fail, the application designer is asked to change the designer.

- Scheduling Editor: product engineer takes the constructed schedule and is responsible for installing it on the system infield. The schedule will be analyzed and tested for engineering constraints, e.g., separation of activities, and potentially edited and modified to meet these demands, while pertaining feasibility.

- Simulator: analysts perform extra analysis on the constructed schedule, e.g., simulation of run-time activities by "stress testing" via overload scenarios, reliability analysis, etc. The analysts may not be tightly involved, but provide analysis results only.

C.2 MPEG Analysis Tools

We presented in chapter 6 the analysis results for a number of realistic MPEG-2 video stream. In order to perform the analysis we have both implemented and used/modified existing MPEG tools. Here we describe both categories.

C.2.1 MPEG Stream Analyzer

Implemented by: Damir Isovic

This is a command line based tool implemented in C. It parses a MPEG-2 video stream and collects relevant information. It reads the input stream byte by byte and as soon a relevant header is found, corresponding function is called to extract desired information from it.

Here is an implementation example of a function that extracts some stream info from the sequence header, depicted in figure C.7.

byte 4								byte 5								byte 6								byte 7							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
horizontal size												**vertical size**												**aspect**				**frame**			

byte 8								byte 9								byte 10								byte 11							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
bit rate																		**1**	**VBV buffer size**												

Figure C.7: MPEG-2 Sequence Header

As we can see from the figure e.g., horizontal size information is contained in two bytes: all bits in byte 4[3] plus the first four bytes of byte 5. This requires some bit-shifting to extract relevant information.

```
/****************************************************
 NAME: getStreamInfo
 IN  : MPEG-2 video stream pointer
 OUT : -

 DESC: Get stream info from sequence header (SH)

      width   = byte4 and the first half of byte5
      height  = the second half of byte5 and byte6
      aspect  = the first half of byte7
      fps     = the second half of byte7
      bitrate = bytes8,9 and two first bits of byte10
```

[3]Bytes 0-3 in the each header are reserved for start code prefix (byte 0-2) and header start code or stream ID (byte 3)

```
      The extracted information is copied into the
      global variables:

      width, height, fps, aspect_ratio_str

 ***************************************************/

void getStreamInfo(FILE *MPEG_video_stream){

    long SH_byte4, SH_byte5, SH_byte6, SH_byte7;
    long SH_byte8, SH_byte9, SH_byte10;

    int aspect_ratio_code, frame_rate_code;


    // get relevant bytes from the sequence header
    SH_byte4 = getc(MPEG_video_stream);
    SH_byte5 = getc(MPEG_video_stream);
    SH_byte6 = getc(MPEG_video_stream);
    SH_byte7 = getc(MPEG_video_stream);
    SH_byte8 = getc(MPEG_video_stream);
    SH_byte9 = getc(MPEG_video_stream);
    SH_byte10 = getc(MPEG_video_stream);


    // extract width and height from first 3 bytes
    width = (SH_byte4 << 4) | (SH_byte5 >> 4);
    height = ((SH_byte5 & 0x0f) << 8) | SH_byte6;


    // extract aspect ratio anf fps codes from byte 7
    aspect_ratio_code = (SH_byte7 >> 4) & 0x0f;
    frame_rate_code = SH_byte7 & 0x0f;


    // translate the codes for aspect ratio
    switch(aspect_ratio_code){
        case 0x01: aspect_ratio_str = "1:1"; break;
        case 0x02: aspect_ratio_str = "3:4"; break;
        case 0x03: aspect_ratio_str = "9:16"; break;
        case 0x04: aspect_ratio_str = "1:2.21"; break;
        default: aspect_ratio_str = "unknown"; break;
    }


    // translate the codes for fps value
    switch(frame_rate_code){
        case 0x01: fps = 23.976; break;
        case 0x02: fps = 24; break;
        case 0x03: fps = 25; break;
        case 0x04: fps = 29.970; break;
        case 0x05: fps = 30; break;
        case 0x06: fps = 50; break;
        case 0x07: fps = 59.940; break;
        case 0x08: fps = 60; break;
        default: fps = 0; break;
    }


    // extract bitrate
    bitrate = (((SH_byte8 << 8) | SH_byte9) << 2 )
              | (SH_byte10 >> 6);


    // bitrate is measured in units of 400bits/sec
    bitrate *=400;
}
```

The extracted information is collected in a text file, on per GOP basis. For each GOP, frame information (type and size) is printed followed by some GOP statistics such as GOP size, GOP length, GOP type (open or closed), the smallest and the largest frame in GOP, and some unusual cases, e.g., some P or B frame is larger than the I frame.

Here is an example of an output file:

```
Stream       :  action_movie.m2v
```

```
aspect ratio:  9:16
fps         :  29.970000
resolution  :  720 x 480
bitrate     :  7500000 bits/sec
======================================================
GOP 1        Type Size GOP summary
======================================================
f [1]         I      429568      sum  = 3300816 bits
f [2]         B      214752      len  = 13 frames
f [3]         B      206208      type = open GOP
f [4]         P      355176      min  = 214752 (B)
f [5]         B      240512      max  = 429568 (I)
f [6]         B      243600
f [7]         P      335096      P>some_previous_P=true
f [8]         B      238144      B>some_previous_B=true
f [9]         B      233128
f [10]        P      336168
f [11]        B      233968
f [12]        B      234496
======================================================
GOP 2

f [13]        I      414336      sum  = 1995408 bits
f [14]        B      534944      len  = 12 frames
f [15]        B      83856       type = open GOP
f [16]        P      214568      min  = 56328 (B)
f [17]        B      66240       max  = 534944 (B)
f [18]        B      61712
f [19]        P      187696      # B>I = 1
f [20]        B      59928       # B>P = 3
f [21]        B      56328       P>some_previous_P=true
f [22]        P      188888      B>some_previous_B=true
f [23]        B      62112
f [24]        B      64800
======================================================
GOP 3
...
```

C.2.2 MPEG Transcoder

Implemented by: Damir Isovic

MPEG Transcoder is a command line based tool written in C that applies our frame selection selection algorithm presented in chapter 7. It takes a MPEG-2 video stream and the amount of available resources as input and produces a tailored video stream, i.e., the one that is guaranteed to be timely processed with respect to the available bandwidth. The produced MPEG-2 video streams are fully compatible with the standard and can be played out in third-party decoders, e.g., Windows Media Player by Microsoft.

MPEG Transcoder also works as a simulator for different frame selection algorithms. Due to the modular architecture of the transcoder, other frame selection algorithms can easily be added. Current implementation includes a naive, best-effort frame selection algorithm, needed for the comparison to our method, see chapter 8.4 for comparison details.

Furthermore, the tool provides a random system load generator for simulation purposes, but the load can also be specified by the user and given as input to the tool.

This tools can be used both offline, transcoding a stored MPEG stream, and online, responding to an incoming MPEG stream, e.g., via the network.

Here is an example of an output file with statistics on useful resource consumption i.e, fully decoded frames that contribute to the overall picture quality, wasted resources and the number of decoded frames per GOP:

```
----------------------------------------------------

SIMULATION        : QAFS (uneven load, avg dec times)
GOP Satisfaction : 0.7
INPUT stream      : test_movie.m2v
OUTPUT stream     : test_movie_[GOP_sat=0.7].m2v
```

```
====================================================
Useful (%)   Wasted (%)   Decoded/GOP (%)
====================================================
85           15              83
81           19              63
62           28              50
...
====================================================
Total frames    :  156322
Decoded frames  :  118804  (76%)
Skipped frames  :  37518   (24%)
----------------------------------------------------
```

C.2.3 Peggy Tracer

Implemented by: Christian Hultman and Patrik Samuelsson[4]

Peggy Tracer (see figure C.8) is an application for stepping through an arbitrary MPEG video stream frame by frame. It is written in C and it is based on the *libmpeg2* library [33].

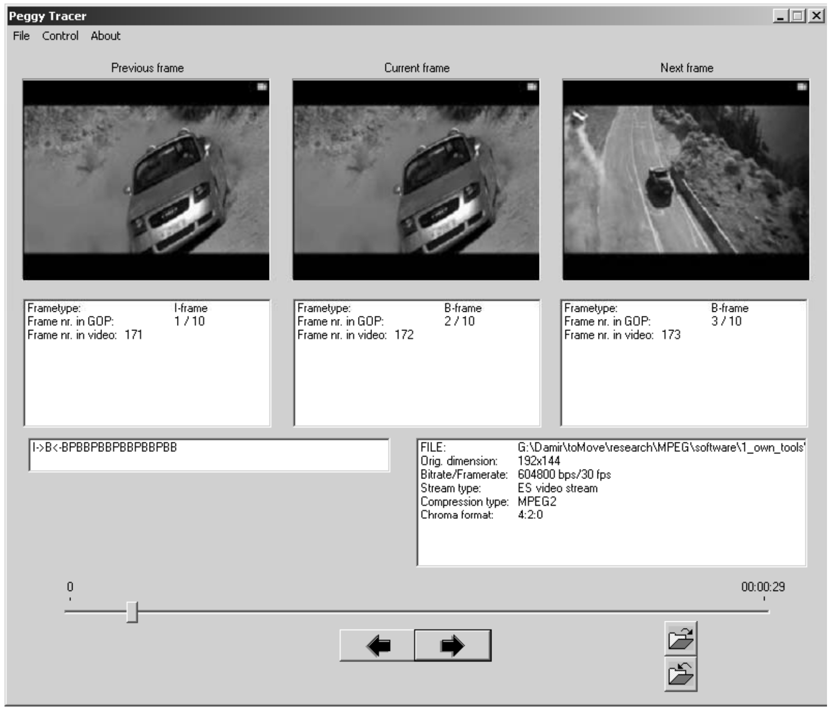
The tool displays general stream information, e.g., bit rate, frame rate, compression type, resolution etc and frame specific information, e.g., frame type, frame number in GOP etc.

The user may step through the stream frame-by-frame, forward or backward , or make fast jumps to an arbitrary point in the stream. At any step, the tools shows the current frame, the previous frame and the next frame allowing easy comparison between them. We have used this feature to study differences between frames.

C.2.4 Other MPEG tools

Here is the list of MPEG tools written by people other than myself, my co-researchers or master students within our research group, which we used or modified to fit our needs:

[4]Christian and Patrik are former master students supervised by our research group



```
Frametype:            I-frame
Frame nr. in GOP:     1 / 10
Frame nr. in video:  171
```

```
Frametype:            B-frame
Frame nr. in GOP:     2 / 10
Frame nr. in video:  172
```

```
Frametype:            B-frame
Frame nr. in GOP:     3 / 10
Frame nr. in video:  173
```

```
I->B<-BPBBPBBPBBPBBPBB
```

```
FILE:               G:\Damir\toMove\research\MPEG\software\1_own_tools'
Orig. dimension:    192x144
Bitrate/Framerate:  604800 bps/30 fps
Stream type:        ES video stream
Compression type:   MPEG2
Chroma format:      4:2:0
```

Figure C.8: Peggy Tracer screenshot

mpeg2dec/libmpeg2

Authors: Aaron Holtzman, Michel Lespinasse et al

libmpeg2 [33] is a free library written in C for decoding MPEG-2 and MPEG-1 video streams. It is released under the terms of the GPL license. mpeg2dec is a test program for libmpeg2. It decodes MPEG-1 and MPEG-2 video streams, and also includes a demultiplexer for MPEG program streams. We have modified this software to measure decoding times for frames.

Berkeley MPEG player

Author: Berkeley Multimedia Research Center

The Berkeley MPEG player [18] is an open source MPEG player written in C. We have extended it to support different frame selection strategies and to measure decoding times for the frames.

MPEG2event

Author: Ketan Mayer-Patel[5]

MPEG2Event [43] is a library written in C# (.NET) intended to facilitate rapid prototyping of MPEG-2 analysis tools. It provides an event-based architecture: as the video stream is parsed, the library constructs and publishes an event for each coding element encounter. We have implemented C# programs that use MPEG2event library to analyze frames on sub-frame level, i.e., slices, macroblocks and blocks.

[5]Ketan is an Assistant Professor at the University of North Carolina at Chapel Hill, USA. I have interacted with Ketan and used his library MPEG2event during my visit of UNC in Spring 2004.

Bibliography

[1] ISO/IEC 13818-2: Information technology - generic coding of moving pictures and associated audio information, part2: Video. 1996.

[2] Luca Abeni and Giorgio C. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium, RTSS'98*, Madrid, Spain, December 1998.

[3] S. A. Aldarmi and Alan Burns. Dynamic value-density for scheduling real-time systems. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems (ECRTS'99)*, December 1999.

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

[5] Veronica Baiceanu, Crispin Cowan, Dylan McNamee, Calton Pu, and Jonathan Walpole. Multimedia applications require adaptive cpu scheduling. In *Proceedings of the Workshop on Resource Allocation Problems in Multimedia Systems*, Washington, DC, USA, 1996.

[6] S. Baruah, G. Buttazzo, S. Gorinsky, and G. Lipari. Scheduling periodic task systems to minimize output jitter. In *Proceedings of*

the 6th International Conference on Real-Time Computing Systems and Applications, Dec. 1999.

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

[8] S. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard real-time sporadic tasks on one processor. December 1990.

[9] A.C. Bavier, A.B. Montz, and L.L. Peterson. Predicting mpeg execution times. In *Proceedings of ACM International Conference on Surement and Modeling of Computer Systems (SIGMETRICS 98)*, Madison, Wisconsin, USA, June 1998.

[10] Reinder J. Bril, Maria Gabrani, Christian Hentschel, G. C. van Loo, and E. F. M. Steffens. Qos for consumer terminals and its support for product families. In *Proceedings of the International Conference on Media Futures*, Florence, Italy, May 2001.

[11] Reinder J. Bril and Liesbeth F.M. Steffens. User focus in consumer terminals and conditionally guaranteed budgets. In *Proceedings of 9th International Workshop on Quality of Service - IWQoS 2001*, June 2001.

[12] Lars O. Burchard and Peter Altenbernd. Estimating decoding times of mpeg-2 video streams. In *Proceedings of International Conference on Image Processing (ICIP 00)*, Vancouver, Canada, September 2000.

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

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

[15] G.C. Buttazzo, G. Lipari, and L. Abeni. A bandwidth reservation algorithm for multi-application systems. *Proceedings of the International Conference on Real-time Computing Systems and Applications, Japan*, 1998.

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

[17] Jan Carlson, Tomas Lennvall, and Gerhard Fohler. Enhancing time triggered scheduling with value based overload handling and task migration. In *Proceedings of 6th IEEE International Symposium on Object-oriented Real-time distributed Computing ISORC'2003*, May 2003.

[18] Berkeley Multimedia Research Center. Berkeley MPEG player. URL: http://bmrc.berkeley.edu/frame/research/mpeg/.

[19] 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.

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

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

[22] G. de Haan. IC for motion compensated deinterlacing, noise reduction and picture rate conversion. *IEEE Transactions on Consumer Electronics*, August 1999.

[23] M.L Dertouzos. Control robotics: the procedural control of physical processes. *Information Processing*, Fall 1974.

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

[25] Michael Ditze and Peter Altenbernd. Method for real-time scheduling and admission control of MPEG-2 streams. In *Proceedings of the 7th Australasian Conference on Parallel and Real-Time Systems (PART2000)*, Sydney, Australia, November 2000.

[26] J.A. Stankovic et. al. Implications of classical scheduling results for real-time systems. *IEEE Computer, Volume 28, Number 6, pp. 16-25*, June 1995.

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

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

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

[30] Gerhard Fohler. Dynamic timing constraints — relaxing overconstraining specifications of real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium – Work-in-Progress Session*, December 1997.

[31] Gerhard Fohler, Damir Isović, Tomas Lennvall, and Roger Vuolle. SALSART - a web based cooperative environment for offline real-time schedule design. In *10th Euromicro Workshop on Parallel, Distributed and Network-based Processing (PDP '02)*, Gran Canaria, Spain, January 2002.

[32] Christian Hentschel, Ralph Braspenning, and Maria Gabrani. Scalable algorithms for media processing. In *Proceedings of the IEEE*

International Conference on Image Processing (ICIP), Thessaloniki, Greece, pp. 342-345, October 2001.

[33] Aaron Holtzman and Michel Lespinasse. libmpeg2 library. URL: http://libmpeg2.sourceforge.net/.

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

[35] D. Isovic, G. Fohler, and Liesbeth F. Steffens. Timing constraints of MPEG-2 decoding for high quality video: misconceptions and realistic assumptions. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, Porto, Portugal, June 2003.

[36] Damir Isovic, Gerhard Fohler, and Liesbeth F. Steffens. Some misconceptions about temporal constraints of MPEG-2 video decoding. Austin, Texas, USA, December 2002.

[37] Damir Isovic, Gerhard Fohler, and Liesbeth F. Steffens. Real-time issues of MPEG-2 playout in resource constrained systems. June 2004.

[38] K. Jeffay. Scheduling sporadic tasks with shared resources in hard real-time systems. *Deptartment of Computer Science, University of North Carolina at Chapel Hill*, 1992.

[39] Yingwei Chen John Tse-Hua Lan and Zhun Zhong. MPEG-2 decoding complexity regulation for a media processor. In *Proceedings of the 4th IEEE Workshop on Multimedia Signal Processing (MMSP)*, Cannes, France, pp. 193 - 198, October 2001.

[40] H. Kopetz. Sparse time versus dense time in distributed real time systems. *Proceedings of the Second International. Workshop on Responsice Computer Systems, Saitama, Japan*, Oct. 1992.

[41] G. Koren and D. Shasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In *Proceedings of the Real-Time Systems Symposium (RTSS'02)*, Dec. 1992.

[42] J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanched aperiodic responsiveness in hard real-time environment. In *Proceedings of the IEEE Real-Time Symposium, RTSS 1987*, San Jose, California, USA, December 1987.

[43] Ketan Mayer-Patel. Mpeg2event library. URL: http://kmp-cs.cs.unc.edu:8080/mpeg2event/overview.html.

[44] Ketan Mayer-Patel. Performance of a software MPEG video decoder. In *ACM Multimedia Conference*, 1993.

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

[46] J. Kee-Yin Ng, C. Kin-Cheung Hui, and Wai Wong. A multi-server design for a distributed MPEG video system with streaming support and QoS control. In *Proceedings of the 7th International Conference on Real-Time Systems and Applications (RTCSA'00)*, Cheju Island, South Korea, December 2000.

[47] Joseph K.Y. Ng, Karl R.P.H. Leung, W. Wong, Victor C.S. Lee, and Calvin K.C. Hui. Quality of service for MPEG video in human perspective. In *Proceedings of the 8th Conference on Real-Time Computing Systems and Applications (RTCSA 2002)*, Tokyo, Japan, March 2002.

[48] Sharon Peng. Complexity scalable video decoding via idct data pruning. In *Digest of Technical Papers IEEE International Conference on Consumer Electronics (ICCE)*, pp. 74-75, June 2001.

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

[50] 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.

[51] M. Spuri, Giorgio C. Buttazzo, and F. Sensini. Robust aperiodic scheduling under dynamic priority systems. *Proceedings of the Real-time Systems Symposium IEEE RTSS'95*, Dec. 1995.

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

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

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

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

[56] Liesbeth Steffens. Software mpeg decoding. *Koninklijke Philips Electronics N.V. 2001, Company restricted paper*, April 2001.

[57] L. Teixera and M. Martins. Video compression: The MPEG standards. In *Proceedings of the 1st European Conference on Multimedia Applications Services and Techniques (ECMAST 1996)*, Louvian-la-Neuve, Belgium, May 1996.

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

[59] T. Tia, W.S. Liu, J. Sun, and R. Ha. A linear-time optimal acceptance test for scheduling of hard real-time tasks. *Deptartment of*

Computer Science, University of Illinois at Urbana-Champaign, 1994.

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

[61] M. Torngren. Fundamentals of implementing real-time control applications in distributed computer systems. *Real-Time Systems Journal, Volume 14, Number 3,* May 1998.

[62] Clemens C. W&#252;st and Wim F. J. Verhaegh. Quality control for scalable media processing applications. *Journal of Scheduling*, 7(2):105–117, 2004.

[63] Weirong Wang, Al Mok, and Gerhard Fohler. A hybrid proactive approach for integrating off-line and on-line real-time schedulers. In *Proceedings of Third International Conference on Embedded Software (EMSOFT 03)*, October 2003.

[64] Weirong Wang, Aloysius K. Mok, , and Gerhard Fohler. Generalized pre-scheduler. In *Proceedings of 16th Euromicro Conference on Real-time Systems (ECRTS 04)*.

[65] Weirong Wang, Aloysius K. Mok, and Gerhard Fohler. Pre-scheduling on the domain of integers. In *Proceedings of IEEE Real-Time Systems Symposium (RTSS 04)*, December 2004.

[66] John Watkinson. *The MPEG handbook*. ISBN 0 240 51656 7, Focal Press, 2001.

[67] Clemens Wüst, Liesbeth Steffens, Reinder J. Bril, and Wim F.J. Verhaegh. Qos control strategies for high-quality video processing. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, Catania, Sicily, Italy, June 2004.

[68] Jia Xu and David Lorge Parnas. On satisfying timing constraints in hard-real-time systems. In *Proceedings of the conference on Software for citical systems*, 1991.

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

[70] Zhun Zhong and Yingwei Chen. Scaling in MPEG-2 decoding loop with mixed processing. In *Digest of Technical Papers IEEE International Conference on Consumer Electronics (ICCE)*, pp. 76-77, June 2001.

Glossary

AAC

Advanced Audio Coding. An audio compression format defined by the MPEG standard. The audio part of MPEG-2 audio/video standard that has been further improved in MPEG-4 , 21

Aperiodic task

A type of dynamically arriving task that consist of a sequence of identical instances activated at *irregular* intervals. Events that triggers an aperiodic task may occur at any time, e.g., a device generates interrupts, an operator presses the emergency button, alarms, etc, 10

Bit rate, BR

The rate at which the coded bit stream is delivered from the storage medium to the input of a decoder, 91

CBR

Constant bit rate means that the rate at which the video data in a stream should be consumed is constant. It varies the quality level of the video frames in order to ensure a consistent bit rate throughout an encoded file, 23

Codec

Short for compressor/decompressor, a codec is any technology for compressing and decompressing data, 22

DTV
Digital Television refers to the standard of transmitting and receiving television signals using purely digital transmission. See High Definition Television for more info , 104

DVB
Digital Video Bradcasting is a set of standards that define digital broadcasting using existing satellite, cable, and terrestrial infrastructures, 20

DVD
Digital Versatile Disc is an optical disc storage media format that is used for playback of movies with high video and sound quality and for storing data, 20

Deadline
A task deadline is the time within which a real-time task should complete its execution. A task deadline can be hard, soft and firm. A *hard* task deadline must never be missed. Missing a hard deadline may cause catastrophic consequences on the environment being controlled. A *soft* task deadline can be missed. Meeting soft deadline is desirable for performance reasons. A *firm* deadline must be met once the task is guaranteed to complete in time, 6

Decoding order
The order in which frames are transmitted and decoded. This order is not necessarily the same as the display order, 90

Display order
The order in which the decoded frames are displayed. Normally this is the same order in which they were presented at the input of the encoder, 90

Elementary Stream
A general term for a coded bit stream such as audio or video. An elementary stream contain a single type of (usually compressed)

signal, e.g., digital control data, digital audio or digital video. Elementary streams are made up of packs of packets, 23

Event-triggered approach

In event-trigged systems all activities are carried out in response to relevant events external to the system, e.g., a sensor generates an interrupt which triggers a certain task. Temporal control is enforced from the environment onto the system in an unpredictable manner (interrupts)., 7

Frame rate, FR

The number of frame frames processed by the decoder per unit of time. E.g., a frame rate of 30 fps means: thirty frames are displayed per second, 92

HDTV

High Definition Television means broadcast of television signals with a higher resolution than traditional formats (NTSC, SECAM, PAL) allow. The high resolution images (1920 pixels 1080 lines or 1280 pixels 720 lines) allow much more detail to be shown compared to analog television or regular DVDs. MPEG-2 is used as the compression codec, 21

Inter-coding

A coding technique for motion video that expolits spatial redundancy within a single picture, 23

Intra-coding

A coding technique for motion video that exploits temporal redundancy between successive frames, i.e., instead of sending entire picture information, only the difference from the previous picture is sent, 22

MP3

MPEG-1 Layer 3. A compression standard for audio. The audio part of the MPEG-1 audio/video compression standard, 21

MPEG-1
An MPEG video/audio standard optimized for CD-ROM. Used mostly in VCDs , 20

MPEG-2
An MPEG video/audio standard optimized for broadcast quality video. In particular, MPEG-2 has become the coding standard for digital video streams in consumer content and devices, such as DVD movies and digital television set top boxes for Digital Video Broadcasting (DVB) , 20

MPEG-3
A proposed standard of the MPEG group that never has been implemented. MPEG-3 was intended as an extension of MPEG-2 for HDTV but was eventually merged into MPEG-2 , 20

MPEG-4
An MPEG video/audio standard primarily designed to handle low bit rate content. MPEG-4 contains many of the features of MPEG-1 and MPEG-2, adding new features such as object-oriented composite files and various types of interactivity. Application areas include picture phones, streaming media, Internet, etc , 20

MPEG
Moving Picture Experts Group (MPEG), standard for coded representation of digital audio and video, 20

Offline (pre-run-time) scheduling
A scheduling method in which all scheduling decisions are pre-computed offline, before we start the system. Task are executed in predetermined fashion, according to a time-triggered approach, 6

Online (run-time) scheduling
A scheduling method in which all active tasks are reordered every time a new task enters the system or a new event occurs. Online

scheduling algorithms make their scheduling decisions at runtime, 7

Periodic task
A type of task that consist of a sequence of identical instances, activated at *regular* intervals. Examples include audio and video sampling, speed regulation, monitoring of temperature, etc., 9

Precedence order
A partial ordering between individual task executions. If task A precedes B then both tasks runs with the same period, but A must complete before B starts to execute, 11

Preemption
An operation of the kernel that interrupts the currently executing task and assigns the processor to a more urgent task ready to execute., 8

Program Stream
MPEG system stream that consists of one or several Elementary Streams with the same time basis. This form of multiplexing is used transmission in a relatively error-free environment, 24

Scheduling
Real-time scheduling is an activity that determines the order in which concurrent tasks are executed on a processor. If several processors are used in the system, then a scheduling policy determines both when (the order in time) and where (which processor) tasks are executed on, 6

Skipped macroblock
A macroblock for which no data is encoded. , 122

Slot Shifting
A metod to combine offline and online scheduling. Dynamic activities are incorporated into static schedules by making use of the unused resources and leeways in the schedule, 39

Spatial redundancy
Redundant (nearly identical) information within the same picture. It occurs because pats withing a single picture are often replicated (with minor changes), 23

Sporadic task
A type of dynamically arriving task that consist of a sequence of identical instances activated at irregular intervals, but with known *minimum interarrival time* between consecutive instances. After the minimum interarrival time has elapsed, the next instance can arrive at any time, 10

Task constraints
Task constraints can be simple and complex. *Simple* task constraints are task attributes such as period, start-time and the deadline. *Complex* task constraints are such relations or attributes which cannot be expressed directly using simple task constraints, e.g., synchronization, precedence, end-to-end deadlines etc. In the most cases, offline transformations are needed to schedule these at run-time, 11

Task
A sequential program that performs a specific activity and that possibly communicates with other tasks in the system. A task often has a priority relative to other tasks in the system, 6, 9

Temporal redundancy
Redundant (nearly identical) information between adjacent video frames. It arises when successive pictures of video display images of the same scene, 22

Time-triggered approach
Time-triggered systems are those that react to passage of time, i.e., all activities are initiated at predetermined points in time. Real-time systems of this kind are time triggered in the sense that their overall behaviour is globally controlled by a recurring clock tick, 7

Transport Stream
MPEG system stream that consists of one or several Elementary Streams with the different time basis. The transport stream is intended for broadcast systems where error resilience is one of the most important properties , 24

VBR
Variable bit rate, varies the amount of output data in each time segment based on the complexity of the input data in that segment. The goal is to maintain constant quality instead of maintaining a constant data rate by making intelligent bit-allocation decisions during the encoding process, 23

VBV
Video Buffering Verifier, a hypothetical decoder that is conceptually connected to the output of the encoder. Its purpose is to provide a constraint on the variability of the data rate that an encoder or editing process may produce. , 94

VCD
Video CD is the technology that allows around 70 minutes of compressed MPEG-1 video/audio to be stored on a CD, 21

Index

Populärvetenskaplig svensk sammanfattning

”Flexibelt multimedia för resursbegränsade system”

Inom några år kommer de flesta underhållningsprodukter i hemmet, som till exempel TV och videobandspelare, att ersättas av motsvarande digitala produkter. Digitala sändningar lämnar utrymme åt ett bredare utbud av kanaler och helt nya tjänster med hög ljud- och bildkvalitet. Dessa sändningar ställer andra krav än nuvarande analoga tekniker, samtidigt som det finns begränsningar i både uppspelningsenheterna i sig, men även hos omgivningen och användaren. En begränsning är t ex att det mänskliga ögat inte kan arbeta hur fort som helst samtidigt som det ställs höga krav på att ljud och bild upplevs som avbrottsfritt och synkroniserat. Ett annat krav är att uppspelningsenheterna - videomobiltelefonen, handdatorn - skall vara lättare, mindre och energisnålare än idag, och i detta ligger en teknisk begränsning avseende beräkningskapacitet, minne och batteritid. Ett krav är att överföringen måste ske snabbt och med hög och bibehållen kvalitet, samtidigt som nätverken – som t ex Internet – har en begränsad kapacitet.

Vår forskning går ut på att uppfylla krav som ställs av multimedia med tanke på de begränsningar som finns, på ett högkvalitativt sätt. Om det visar sig att uppspelningsenheten inte klarar av att visa en fullständig videofilm, anpassar vår metod filmen till enheten genom att identifiera och spela upp de delar av filmen som ger den bästa möjliga kvaliteten på resulterande bild och ljud. Den digitala filmen anpassas antingen innan man skickar iväg den över nätverket, t ex det mobila nätet, eller när man har tagit emot den i sin enhet, t ex videomobiltelefonen. Sättet filmen anpassas på är beroende av vilka begränsningar och krav som ställs i det specifika tillfället.

Vi använder realtidssystem för att matcha krav som ställs av multimedia med begränsningarna hos uppspelningsenheter och nätverket för videoöverföring. Realtidssystem är en typ av datorsystem som garanterar att resultat av en viss beräkning levereras vid rätt tidpunkt. Tack vare realtid kan de olika medierna för ljud och bild synkroniseras med varandra, så att ljudet följer noggrant video som visas.

Användningsområdena för vår forskning är stora inom multimedia branschen. Om det skall vara möjligt att titta på filmtrailers på en mobiltelefon måste man anpassa filmen till den kapacitet som mobiltelefonen klarar av att spela upp. Andra områden som kan behöva anpassade videoströmmar är handhållna datorer, kameraövervakningssystem, videokonferenssystem, distansundervisning och så vidare. I alla dessa tillämpningsområden möjliggör vår metod en kvalitetsmedveten anpassning av den digitala filmen som resulterar i den bästa möjliga bilden med avseende på tillgängliga resurser för dataöverföring och uppspelning.