Dynamic resource allocation in multimedia applications

van den Heuvel, M.M.H.P.

Award date:
2009

Dynamic Resource Allocation in Multimedia Applications

2IM91 - Master Thesis

Version: 1.0
Eindhoven University of Technology
Department of Mathematics and Computer Science
Chair of System Architecture and Networking

In cooperation with:
Brandenburgische Technische Universität Cottbus
Department of Mechanical, Electrical and Industrial Engineering
Chair of Media Technology

July 24, 2009

Martijn M.H.P. van den Heuvel
Student number: 0547028
m.m.h.p.v.d.heuvel@student.tue.nl

Supervisor:
Dr. ir. R.J. Bril
Assistant Professor
r.j.bril@tue.nl

Internship Supervisor:
Dr.-Ing. habil. C. Hentschel
Professor
christian.hentschel@tu-cottbus.de

Tutor:
ir. M.J. Holenderski
PhD. Candidate
m.holenderski@tue.nl

Internship Tutor:
Dipl.-Ing. S. Schiemenz
Researcher
stefan.schiemenz@tu-cottbus.de

PREFACE AND ACKNOWLEDGEMENT

This master thesis describes the work of a six month project to conclude the Embedded Systems Master study at Eindhoven University of Technology. The project is a collaboration between two universities, Eindhoven University of Technology (TU/e) and Brandenburg University of Technology Cottbus (BTU).

The work for this project is partially (the first 1.5 month) done at the System Architectures and Networking (SAN) department at the Eindhoven University of Technology. The main research field of the SAN department is on the architecture of networked embedded systems, including hardware and software aspects. Another part of this project (4 months) is done at the Multimedia Technology (MT) department at the BTU, where extensive research is done in the field of scalable multimedia processing algorithms and technologies.

During the project, the emphasis of the work in this project has moved from research to mechanisms to allow preliminary termination of jobs, towards resource allocation within priority processing applications. The primary description of the project concerns multimedia applications allowing different modes of operation, as developed within the context of the project ITEA/CANTATA. The main goal originally is to investigate, prototype, design, implement and evaluate a mechanism, within a real-time operating system, in order to reduce the required time to perform a mode change.

The project description, as provided by the Multimedia Technology department of the BTU, concerns optimizing the control of priority processing applications. The concept of priority processing, as a paradigm for real-time scalable video algorithms, relies on the availability of mechanisms to preliminary terminate jobs (processing a frame). Although both project descriptions demand preliminary termination of jobs, the project as provided by the SAN department (TU/e) is logically stated from a system's perspective, whereas the project description at the Multimedia department points more to the application domain.

Initial research done at the SAN department at the TU/e (during the first 1.5 month before my internship period at BTU Cottbus), was targeted at preliminary termination in multimedia applications which support multiple modes of operation. The priority processing application can be seen as a more fine-grained approach of scalability which are applicable for special cases of signal processing algorithms. The goal was to implement and compare performance of different mechanisms allowing preliminary termination within the field of priority processing. This would give additional insights in difficulties from the application perspective, before actual implementation and evaluation of the mechanisms within a real-tme operating system. The implementation and evaluation of mechanisms for preliminary termination within a real-time environment is left as future work and replaced with additional work in optimizing the control of the priority processing applications by means of additionally required mechanisms.

On one hand, changing the emphasis in the project makes parts of the initial research less relevant, especially research towards systems allowing mode changes. On the other hand, new research objectives showed up by means of dynamically allocating processor resources among competing scalable video processing algorithms within the priority processing application. In this report it is briefly pointed out how mechanisms for preliminary termination can be used to speed up mode changes. The new direction taken within the project is based upon the diagnoses of the provided priority processing application, in which it became clear that there was lack of a correct real-time task model. This elementary missing model caused a wrong implementation of task priorities and incorrect assumptions on the behavior of the application. The need for correcting the application's misbehavior was a reason to put the emphasis of the work to diagnosing the existing mechanisms used for allocating processor time within the priority processing application, and to compare performance attributes between alternative implementations.

Summarizing the research directives taken during this project, my project is especially targeted at the domain of scalable video algorithms. At the BTU Cottbus, a more fine-grained approach by means of priority processing is developed, than the classical approach of discrete quality modes as a paradigm for scalable video algorithms. In this project, we investigate different mechanisms to efficiently manage resource allocation for priority processing applications. Priority processing applications are composed from one or more scalable video algorithms, requiring controlled preliminary termination of jobs and distribution of resources over its components, hence the name of this report. The goal of this project is to:

- evaluate different mechanisms to allow preliminary termination of scalable video algorithm jobs;
- investigate mechanisms to efficiently divide resources among competing video algorithms;
- reduce control overhead of priority processing applications.

ACKNOWLEDGEMENT

I am grateful to my supervisors, Reinder J. Bril and Christian Hentschel, for introducing me to this project and providing me the opportunity to work in a new, challenging environment at BTU. I experienced a very pleasant internship period in Cottbus, which enriched me as well educational as personal. Furthermore, I want to thank them for their suggestions, contributions and feedback during this project.

I also want to thank my tutors, Mike J. Holenderski and Stefan Schiemenz, for their time, encouraging support and useful (sometimes heated) discussions during my project. Next to expressing my gratitude to them, I want to wish them the best with completing their doctorate track.

Furthermore, I want to thank all students and employees at the SAN department (TU/e) as well as the Multimedia Technology department (BTU) for creating a great working atmosphere and an enjoyable time. Additionally, I want to thank the students I successfully collaborated with during the computer science bachelor and embedded systems master tracks at the TU/e.

I want to thank my friends for supporting me; sharing their ideas and showing interest (also during my internship period). Especially, I want to mention Sander J. Floris and Erik Strijbos for their technical support and friendship.

During my internship in Cottbus, I contacted a lot of fellow students with diverse backgrounds and interests. I want to thank them for spending our spare time in an enjoyable way; sharing insights and relaxing moments.

Finally, my special thanks go to my family for their continuous support during my entire study period.

ABSTRACT

Scalable video algorithms using novel priority processing can guarantee real-time performance on programmable platforms even with limited resources. According to the priority processing principle, scalable video algorithms follows a priority order. Hence, important image parts are processed first and less important parts are subsequently processed in a decreasing order of importance. After creation of an initial output by a basic function, processing can be terminated at an arbitrary moment in time, yielding the best output for given resources. This characteristic of priority processing allowing preliminary termination, requires appropriate mechanisms.

Dynamic resource allocation is required to maximize the overall output quality of independent, competing priority processing algorithms that are executed on a shared platform. To distribute the available resources, i.e. CPU-time, among competing, independent priority processing algorithms, a scheduling policy has been developed, which aims at maximizing the total relative progress of the algorithms on a frame-basis. Allocation of processor resources to priority processing algorithms can be established by means of different mechanisms.

The allocation of resources requires monitoring mechanisms to provide the control component with information about the progress of the individual algorithms relative to the amount of consumed time.

In this report, we describe basic mechanisms for dynamic resource allocation:

1. **Preliminary Termination:** Upon expiration of a predefined, periodic deadline, all scalable priority processing algorithms must be terminated and computations for a next frame must be started. Although it is conceivable to let the algorithms check whether or not it should preliminary terminate at regular intervals (e.g. at a finer granularity level than entire frames), this would give rise to additional overhead. Alternatively, it is conceivable to provide an a-synchronous signalling mechanism allowing preliminary termination of activities.
2. **Resource Allocation:** The decision scheduler divides the available resources within a period into fixed-sized quanta, termed time-slots, and dynamically allocates these time-slots to the algorithms based upon the scheduling policy. To allocate processor time, a task implementing a scalable priority processing algorithm is assigned the processor by means of either (i) suspending and resuming the task or (ii) manipulating the task priority such that the native fixed priority scheduler (FPS) of the platform can be used. The latter option allows the consumption of gain-time.
3. **Monitoring:** The scheduling policy decides each time-slot which algorithm to assign the processor resources. This decision is based upon progress values of the algorithms relative to the amount of time consumed by these algorithms. Monitoring requires the decision scheduler to be activated after each time-unit. Each period consists of a fixed amount of time-slots. Both mechanisms for preliminary termination and resource allocation rely on correct accounting of these time-slots.

The priority processing applications are prototyped in a Matlab/Simulink environment and executed on a multi-core platform under Microsoft Windows XP. This platform is used to compare the performance of different implementations of the above described mechanisms.

For preliminary termination, we want to minimize the following two measurable performance attributes:

1. **Termination latency:** Based on extensive tests, there is no measurable difference in polling each block- or pixel-computation. Signalling is less reliable, causes relative high latencies and is not available on most platforms (including the Windows platform).
2. **Computational overhead:** Polling involves computational overhead and disturbs signal processing. In our simulation environment there is just a small computational overhead measurable.

Resource allocation mechanisms are optimized by reducing context-switching overhead. Furthermore, priority manipulation mechanisms has the additional advantage over suspending-resuming of tasks that it allows priority processing to consume gain-time.

Monitoring shows difficulties in enforcing time-slots on the Windows platform, since accounting of time-slots relies on the availability of high-resolution timers. Therefore, the decision scheduler implements a busy-waiting loop and must be mapped on a separate core.

Finally, it is shown that the control overhead of the priority processing application is significantly improved by combining block-based polling as a mechanism for preliminary termination; priority manipulation using the Windows fixed priority scheduler as resource allocation mechanism; reducing context-switching overhead and allowing gain-time consumption.

TABLE OF CONTENTS

1. INTRODUCTION

In this report, we look at mechanisms to support the control of priority processing applications. Priority processing defines a novel approach towards scalable video algorithms. Dynamic resource allocation is required to maximize the overall output quality of multiple video algorithms that are executed on a shared platform.

First, Section 1.1 motivates the development of scalable video algorithms. Next, in Section 1.2 the problem description is stated more carefully, followed by Section 1.3 which defines the goals of this report. In Section 1.4 the used approach of this research project is summarized, followed by a description of the contributions of this project compared to current research results. Section 1.6 presents a global report outline. Finally, the glossary at the end of this section presents a list with abbreviations used through this report.

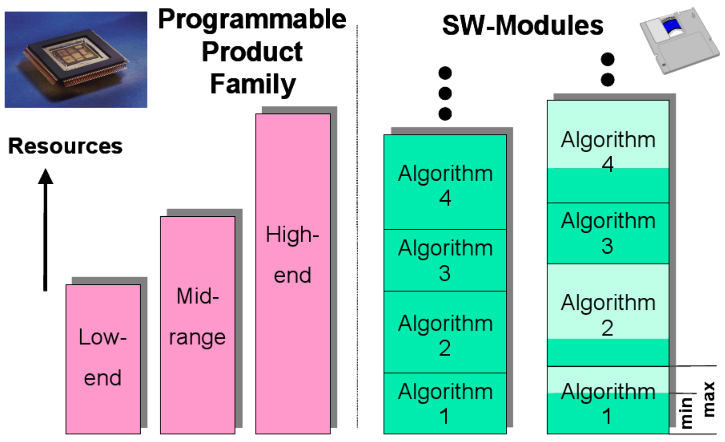
1.1 CONTEXT AND BACKGROUND

In current multimedia systems, signal processing functionality are moved from dedicated hardware to software implementations [18]. A problem is to guarantee real-time constraints on programmable platforms with data dependent load conditions. Most algorithms for video processing have a fixed functionality and cannot be adapted to resources. First generation of scalable video algorithms can trade-off picture quality against resource usage at the level of individual frames, by means of providing a limited number of quality levels [18, 39]. These first developed scalable video algorithms typically have a kernel which is not scalable and a scalable part to increase quality.

High quality video processing has real-time constraints. Hard real-time systems require analysis based upon Worst-Case Execution Time (WCET). Due to the highly fluctuating, data dependent resource demands of video processing algorithms, it is not very cost effective to deploy these algorithms on hardware platforms based upon worst-case resource demands. On one hand, implementing video algorithms using the classical real-time approach based upon WCET analysis would lead to:

1. high development cost to extract the WCET from an algorithm. Due to complex algorithms and the data dependent characteristics it is very difficult, if not impossible, to obtain correct WCET estimates.
2. low processor utilization. Due to high load fluctuations within video processing applications, the average case computation times for a frame are not near to the worst case computation times. By requiring resource availability based upon worst case demands, a lot of resources are wasted.

On the other hand, due to the characteristic that worst-case and average case execution times differ a lot, also overload situations can occur. Overload situations cause deadline misses and therefore a reduced output quality.



• *Figure 1: Scalable video algorithms allow a trade-off in resource requirements and output quality and provide cost-effective development. Products in different price-ranges and product families can be supplied with the same software by adapting the resource demands of the algorithms to the available resources on the platform. Source: [18].*

Scalable video algorithms allow to compromise quality for lower resource demands, as indicated in Figure 1. The trade-off in resource requirements allows deployment of the same software components in consumer electronic products of different price-ranges as well as different product families. A typical use-case, as shown in Figure 1, shows that full software functionality is available in the highest price-range consumer product, which is supplied with the most powerfull hardware. Lower price-ranges are supplied with cheaper hardware and must deal with reduced functionality and/or quality.

The possibility to reuse software components makes scalable video algorithms attractive in a cost-effective sense and decreases the time to market for new products.

1.2 PROBLEM DESCRIPTION

Current trends in consumer electronic products move towards more software oriented approaches, with an increased amount of functionality within the same device. This trend causes that multiple (scalable) software components have to share the same hardware. In this report we consider a special variant of scalable video algorithms, named priority processing. It is investigated which mechanisms are required to efficiently share (processor) resources among competing priority processing video algorithms, which have to share the same resources.

1.3 PROJECT GOALS

Mechanisms are investigated to allow efficient allocation of resources over competing, independent scalable video components. These mechanisms are applied in the priority processing applications to compare efficiency of resource usage. The goal is to allow an abstraction layer, which provide appropriate mechanisms allowing the priority processing components to share processor resources as efficient as possible, and therefore reduce control overhead.

Part of the resource management scheme is to support fast termination of jobs in scalable video applications, preventing overload situations. Using scalable priority processed multimedia applications, a sub-goal is to get rid of the additional overhead caused by polling of the available budget at regular intervals. Providing signalling mechanisms to preliminary terminate tasks in a predictive manner makes the polling mechanism superfluous. Depending on the granularity of the scalable video algorithms, the polling versus the signalling mechanism can be considered trade-offs in overhead versus latency.

Due to the fluctuating load characteristic of video processing algorithms, it is conceivable that processing of a frame consumes less time than expected. This leaves space for other components to use the otherwise wasted processor resources. An additional challenge is to effectively use this so called gain-time.

1.4 APPROACH

The approach of this project is to first investigate mechanisms to support preliminary termination of jobs and distribute resources over competing, independent components. An important issue is to guarantee state and data consistency for an application.

The next step is to evaluate how the investigated mechanisms can be applied in the priority processing application. These priority processing applications are prototyped in a Matlab/Simulink environment which run on standards Microsoft Windows or GNU/Linux machines.

Subsequently, the investigated mechanisms are compared amongst each other in terms of efficiency, by implementing them in scalable priority processing applications. We consider different mechanisms to allow local scheduling of tasks, as well as mechanisms to account the amount of time used by an task. Mechanisms for preliminary termination of jobs are the first objective to optimize.

Finally, the observations extracted from the experiments are used to indicate further directions towards successful implementation of priority processing applications on a real-time platform.

1.5 CONTRIBUTIONS

This report contributes a dynamic resource allocation scheme applied within the priority processing application. Three basic mechanisms for dynamic resource allocation are identified, for which different implementations are compared to each other:

1. **Preliminary Termination:** This report compares different mechanisms to allow preliminary termination of jobs in multimedia applications. For example: a signalling based approach for preliminary terminating jobs is implemented in a priority processing application, in which the performance, by means of control

overhead and termination latency, can be compared to polling based control mechanisms.

2. **Resource Allocation:** It is investigated which mechanisms are needed to supply the priority processing application with a resource management scheme, which dynamically allocates the available resources over competing, independent components.
3. **Monitoring:** The monitoring mechanism extracts state information from the application on which the scheduling policy relies to make appropriate decisions. Furthermore, the resource allocation and preliminary termination mechanisms rely on information concerning the consumed time by a scalable video component.

Furthermore, the priority processing application has been ported to the GNU/Linux platform, which helped to identify incorrect behavior of the application. A main issue corrected in the models is remapping the priority processing tasks on threads, resulting in reduced latencies upon preliminary termination of jobs.

Finally, a proposal is done to deploy the priority processing applications in a real-time environment, based on gathered simulation results.

1.6 OVERVIEW

During this section, the problem to be solved has been briefly defined and the context of this project is sketched.

Section 2 summarizes related literature, containing useful background information for a general system model. During this section, more elaborate information about scalable video techniques and their control are described. After this section, issues to be solved in project should be clarified in more detail.

Section 3 presents a general architecture for the control of scalable multimedia applications with fluctuating quality levels. After presenting this general architecture, the interaction between the controlling components, system level components and application components will be described in more detail. The main contribution of Section 3 is to outline different implementations for the preliminary termination, resource allocation and monitoring mechanisms.

Section 4 introduces the priority processed video application and shows how the architecture and mechanisms introduced in Section 3 can be mapped onto the priority processing applications. The simulation environment, in which the priority processing applications are prototyped, is also explained in detail. The prototyped models of the priority processing applications, as initially implemented, are carefully diagnosed. Finally, questions and hypotheses are stated in Section 4 which are to be answered and tested.

Section 5 describes the experiments performed with the priority processing applications and presents the corresponding measurement results.

Finally, Section 6 summarized conclusions extracted from the priority processing application; proposes directions of further research and states some overall concluding remarks.

1.7 GLOSSARY

Notation	Description	Page List
APC	A-synchronous Procedure Call	40
API	Application Programming Interface	39
ARM	Advanced RISC Machine	61
ASP	Application Specific Processor	14
CPU	Central Processing Unit	3
DSP	Digital Signal Processor	14
FPDS	Fixed Priority Deferred Scheduling	69
FPPS	Fixed Priority Preemptive Scheduling	69
FPS	Fixed Priority Scheduling	17
GCC	GNU Compiler Collection	40
GOPS	Giga Operations Per Second	63
LCD	Liquid Crystal Display	11
MIPS	Microprocessor without Interlocked Pipeline Stages	61
MPSoC	Multi-Processors System on Chip	61
POSIX	Portable Operating System Interface for Unix	78
QMC	Quality Manager Component	69
QoS	Quality of Service	10
RDTSC	Read Time Stamp Counter	76
RL	Reinforcement Learning	54
RR	Round Robin	54
SMP	Symmetric Multi-core Processing	76
TSC	Time Stamp Counter	58
VLIW	Very Long Instruction Word	35
VQEG	Video Quality Experts Group	32
WCET	Worst-Case Execution Time	6

2. RELATED WORK

This section presents related literature to topics relevant to this report. First a more elaborate description of scalable video algorithms will be given.

Scalable video applications which can run in several modes of operation must ensure smooth transition between quality levels. During mode switches overload situations must be prevented. Similarly, frame-computations must reside within assigned (fixed-sized) budgets. A trend is observed towards more fine-grained scalable video applications. For example: set a quality mode for each frame. Section 2.1 presents most common techniques in the field of scalable video algorithms and relates it to the priority processing concept. Priority processing is introduced in Section 2.2, as a way of self-adapting scalable video algorithms, is capable of handling fluctuating resource availability.

In Section 2.3 different control strategies are compared. First reservation-based strategy for a single algorithm is explained. Thereafter, it is compared with the control strategy developed for competing algorithms within the priority processing application.

Scalable video algorithms are developed to exploit their advantageous characteristic of adaptable resource requirements. This means that salable video algorithms can be ported to different hardware platforms, which provide different amount of resources. However, each platform (software and hardware) provides its own set of primitives to do resource management. In Section 2.4 a overview of trends in platform design is given as basic background information.

In the application domain of scalable video algorithms, preliminary termination of pending work is a desired mechanism to guarantee progress at a certain quality level within resource constraints. Mechanisms for resource management within scalable video applications typically assume a semi-static resource availability. Mechanisms for resource management while guaranteeing availability of resources, known as resource reservation mechanisms, are described in Section 2.5.

A summary of the related work is given in Section 2.6. For a brief description of related research topics which are applicable in this project, it is sufficient to read Section 2.6.

2.1 SCALABLE VIDEO ALGORITHMS: FROM MODES TO PRIORITY PROCESSING

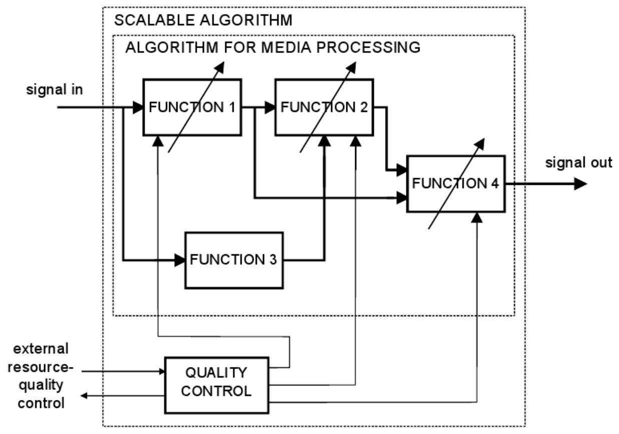
A classical approach to achieve scalability in multimedia applications is to introduce multiple modes of operation, which trade-off resource requirements versus *Quality of Service (QoS)*. An application running on a particular platform is assumed to be able to run in a number of different application-modes. Each mode may for example represent a specific quality level of that application. Example applications include a video decoder that can process an MPEG video-stream at different quality levels or an algorithm improving the quality of an individual frame. Whenever such an application is requested to change its quality level, it typically defers the actual change to a well-defined moment in its code, for example upon completion of an entire frame or completion of an enhancement layer of that frame.

Figure 2 shows an example of a scalable video algorithm which is split into a set of specific functions. Some of these functions are scalable to provide different quality levels. The quality control block contains the information to set appropriate combinations of quality levels of the functions in order to provide an overall acceptable output quality. These quality levels are modes in which the multimedia application can act. A mode change request is initiated when the control triggers an application to change its quality level.

Hentschel et al. [18] define scalable video algorithms as an algorithm that:

1. allows dynamic adaption of output quality versus resource usage on a given platform;
2. support different platforms for media processing;
3. is easily controllable by a control device for several predefined settings.

As an alternative to modes of operations in scalable real-time multimedia applications, the scalable priority processing principle is developed [17]. Priority processing offers a way of deploying scalable video algorithms, which can be interrupted during processing and still deliver best quality for the resources used, instead of skipping a complete output image.



• *Figure 2: Basic structure of a scalable component, in which all functions have common interfaces to communicate with the quality control block in order to adapt quality levels to available resources; source: [18]*

2.2 PRIORITY PROCESSING

With the principle of priority processing [17], an efficient way for self-adapting video algorithms at not guaranteed system resources has been proposed. According to this principle, signal processing follows a priority order. Important image parts are processed first, less important parts in a decreasing order. After creation of an initial output by a basic function, signal processing can be preliminary terminated at arbitrary moment in time.

Hentschel and Schiemenz [17] introduce a scalable priority processing algorithm for sharpness enhancement. After creating a basic output, the image content is analyzed and finally the scalable functions processes the image blocks in which the blocks with high frequency content get preference over other blocks. An example of this algorithm is shown in Figure 3. Traditional image enhancement algorithms process the content from left to right and in a top down fashion. By only enhancing 25% of the image content, this would result in enhancing background content. By giving preference to high frequent blocks, important parts of the foreground are processed first, resulting in an immediately visible result.

Priority processing is characterized by an algorithm dependent quality function with three stages: apply a basic function, analyze the image content and finally apply the scalable algorithm part in order of importance. The behavior of priority processed scalable video algorithms is shown in Figure 4, by a characteristic function in time versus the output quality:

1. Initially, time is spent to produce a basic output at the lowest quality level.
2. Thereafter, time is spent to identify the most important parts of the image content.
3. Finally, the quality of the picture is increased by processing the most important picture parts first.

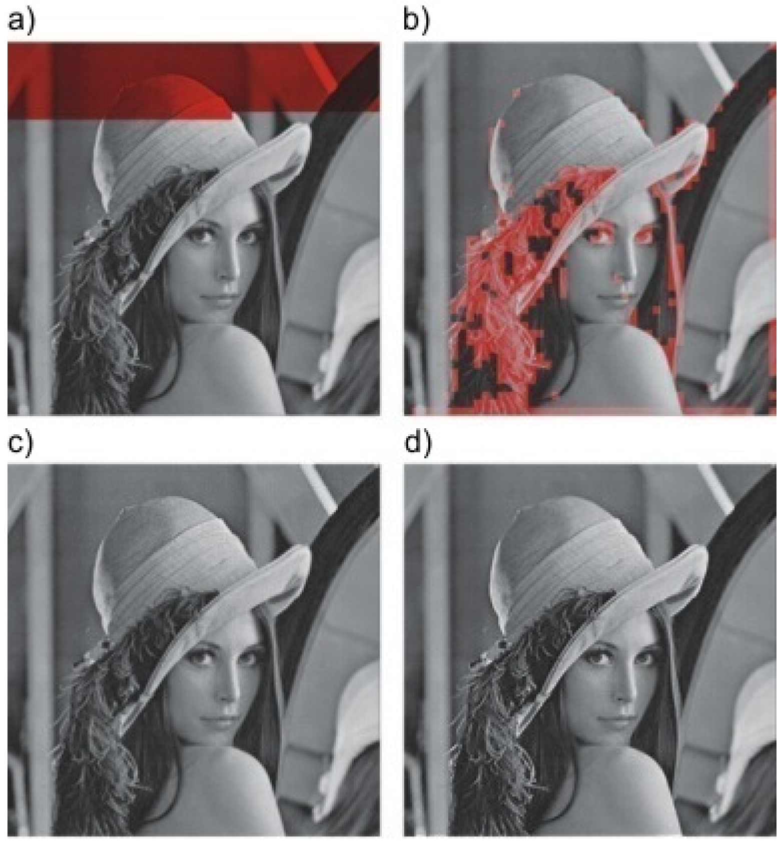
When a basic output is available, processing can be interrupted at any time, which leaves the output at the quality level obtained so far. Each priority processing algorithm has its own characteristic trend.

Schiemenz and Hentschel [33] introduce a more comprehensive deinterlacing priority processing algorithm. A *deinterlacer* fills in the missing lines of interlaced transmitted (or stored) video signals for representation on progressive pixel-oriented displays, such as plasma or Liquid Crystal Displays (LCDs) [33]. After creating a basic output by simple linear interpolation, the scalable part of the deinterlacing algorithm has two parts:

1. Motion adaptive interpolation in stationary areas of the video content.
2. Edge dependent motion adaptive interpolation in moving areas of the video content.

The deinterlacer algorithm analyses the video content and gives priorities on basis of extracted motion information. Each stage of the algorithm improves the signal to noise ratio of the picture content. Both stages are applied sequentially, because applying both algorithms consecutively on a single block would give visible artifacts when preliminary terminating the processing of a frame.

Knopp [23] implemented the scalable deinterlacing algorithm on the DM642 evaluation board of Texas Instruments and compares the results by the hardware accelerated measurements with the measurements obtained from the Matlab/Simulink environment. The qualitative results are good, however the algorithms run a lot



• *Figure 3: Example result of applying an scalable sharpness enhancement algorithm: a) The blocks of the image are processed in sequential order; b) The most important picture parts are identified and processed. In both images the red parts show the progress of the algorithm. c) The result of the sequential algorithm in which only background content is filtered and therefore no visible quality improvement is obtained. d) The result of the priority processing algorithm when the most important image parts are enhanced. Source: [6]*

slower on the hardware as on modern general purpose computers such that realtime performance is not achieved. The flexibility and the increasing computing power of modern general purpose computers give preference to implementing the algorithms in high level languages, instead of porting them to embedded platforms.

2.3 CONTROL STRATEGIES

Wüst et al. [39] describe strategies to adapt quality levels by balancing different parameters determining the user-perceived video quality. These parameters are:

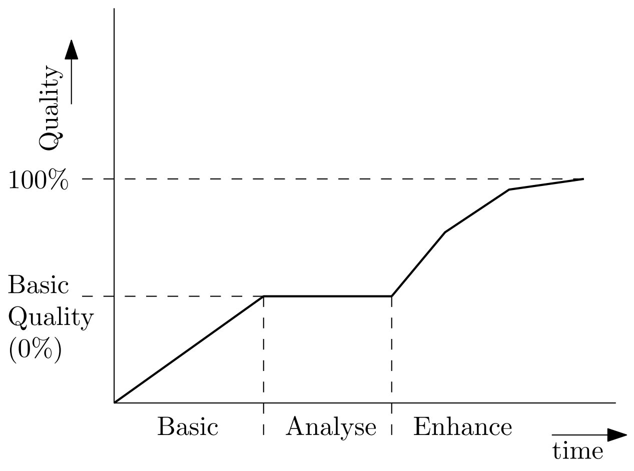
1. **Picture quality:** this is measured as perceived by the user and must be as high as possible;
2. **Deadline misses:** these should be as sparse as possible, because of the induced artifacts;
3. **Quality changes:** the number and size of quality level changes should be as low as possible.

Control strategies try to find optimal settings for these parameters, to maximize the perceived quality level.

2.3.1 BUDGET-BASED CONTROL

Wüst et al. [39] introduce a control strategy which selects a quality mode at the border of each frame-computation. Each quality level assumes a fixed availability of budget for a single algorithm. The selection of the quality level is designed as a discrete stochastic decision problem by means of a modified Markov Decision Process.

A Markov Decision Process is built from states and actions. The states incorporate the task progress and the previous quality level. The actions select a new quality level with an attached probability. The Markov Decision Process is solved off-line for particular budget values. During runtime when the control starts a new job, the controller decides its action by consulting the Markov policy by a simple table look-up.



• *Figure 4: Priority processing, as a function of time consumption versus output quality, can be divided in three time-frames: 1) produce a basic output at the lowest quality level; 2) Identify the most important image content; 3) Enhance the quality of the output by processing the most important picture parts first.*

Such a decision table can also be built during runtime by means of *reinforcement learning.* A reinforcement learning strategy starts with no knowledge and has to learn optimal behavior from the experience it gains during runtime [4]. State-action values in the table give estimates how good a particular action is in a given state. Given a state, the best action in the table is chosen during runtime. This action gives the new runtime quality mode. Wüst et al. [39] claim that only a small amount of pre-determined statistics is needed to get reasonably good start results.

Control strategies described by Wüst et al. [39] assume a *work preserving* approach, which means that a job processing a frame is not aborted if its deadline is missed, but is completed anyhow. The reasoning behind this strategy is that abortion of jobs would waste work already done. In this report, considering priority processing applications, the work preserving property is not applicable.

2.3.2 DECISION SCHEDULER

The strategy [39] selects a quality level for processing the next frame upon completion of the previous frame, i.e. synchronous with processing. Despite the fact that the priority processing approach does not have modes of operation, it also applies reinforcement learning strategies to assign budgets for the scalable video tasks [6, 22, 32]. The control of the priority processing application selects the algorithm to execute next upon completion of a time-slot instead of upon completion of a frame, i.e. synchronous with time.

For the control of priority processing algorithms, a scheduling method based on time-slots is used. A collection of time-slots define a periodically available budget for a frame-computation. The control (decision scheduler) receives the result from the reinforcement learning scheduling policy and enables the selected video task for the duration of one time slot with the entire resources. The decision scheduler allocates time-slots to the scalable algorithms [32] and preliminary terminates processing when the capacity is depleted. Because each algorithm must provide an output at the end of each period, we need appropriate mechanisms to preliminary terminate algorithms when the period ends (deadline is reached).

The allocation of time slots is dependent on the processing progress of individual algorithms. The scalable algorithms update their progress value at runtime. The so called decision scheduler decides an allocation on basis of progress values according to apportion of resources in a fair way, using methods from *reinforcement learning.* Basic implementations of the decision scheduler plus some scalable priority processing algorithms are available.

Schiemenz [32] compares a reinforcement learning scheduling approach with the round-robin approach and the minimum-decision approaches. Each deadline period in which a frame must be processed, is cut into time-slots. The amount of time-slots in a period is at least the amount of competing algorithms. The decision scheduler assigns the time-slots to the algorithms and makes the decision on the basis of obtained progress values. The goal is to assign the time-slots in such a way that the total progress of all algorithms is maximized. It is assumed that the progress of an algorithm, measured by the amount of processed blocks, is proportional to the perceived quality level:

$$Quality \sim Progress \tag{1}$$

Due to the learning characteristic of the reinforcement learning approach, this approach achieves better overall quality than other approaches like:

1. **Round-robin:** Every algorithm is alternately assigned a time-slot. Since progress trends are characteristic for an algorithm and data dependent, one algorithm might need more time than another to reach similar progress values.
2. **Minimum Decision:** The algorithm which has the lowest progress value is assigned the next time-slot. This approach only works with algorithms requiring a similar amount of resources. When an algorithm requires a lot of resources to reach a certain progress value, it can exclude all other competing algorithms. These other algorithms might require only a few resources to complete. Therefore the minimum decision approach does not maximize the total progress value.

Joost [22] investigated how to stabilize the reinforcement learning scheduling in priority processing applications, such that progress values show reduced fluctuations compared to an earlier implementation of Berger [6]. This stabilization leads to less fluctuation in quality. Joost [22] also implemented a prototype on the Texas Instruments DM642 evaluation board. Despite the labor-intensive, but successfully applied, task of mapping a priority processing application on this (relatively outdated) embedded platform, the implementation lacks a satisfactory performance result.

2.4 PLATFORM SUPPORT

In order to map the priority processing application successfully on an arbitrary platform, it is required to have extensive knowledge about the target platform. This section give an overview of the most important aspects of a platform. A *platform*, which is capable to run software components, is defined by [8]:

1. **Hardware:** provides the computational power, memory and possible additional components to speed up computations;
2. **Programming language:** a formalism to express how to use the hardware;
3. **Operating system:** a software layer which distinguishes the hardware layer from the software.
4. **Runtime-libraries:** standard libraries (provided by - or built on top of - the operating system) which provide basic functionality reusable by different components.

Especially the hardware and the operating system are given extra attention in this section. As a special type of system, real-time systems are briefly introduced.

2.4.1 HARDWARE TRENDS

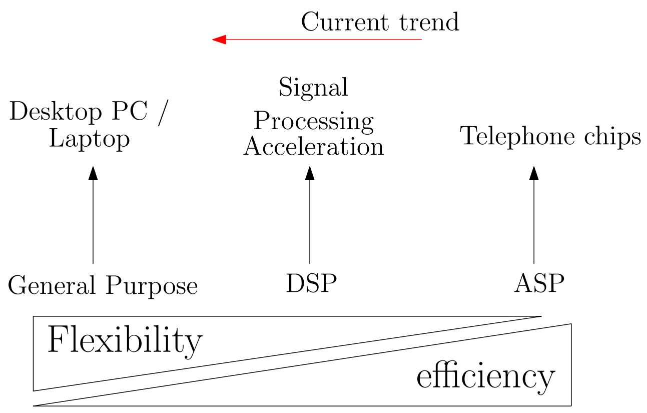
Considering the development of computational power in embedded devices in the last decades, a trend is shown towards increased parallelism [12, Section 2] on different levels in the design. A graphical overview of different classes in processor design, trading off flexibility for efficiency, is shown in Figure 5. Flexibility is a metric which indicates the programmability of the platform. Efficiency is measured in energy consumption.

Figure 5 shows three most important classes of processor architectures:

1. Application Specific Processor (ASP) consists of processing elements dedicated and optimized for a single application.
2. Digital Signal Processor (DSP) consists of relative simple hardware capable of efficiently executing signal processing applications.
3. General purpose processor are very dynamic and programmable platforms, allowing a wide range of applications to run, but at the cost of efficiency.

Production of programmable chips is only cost-effective when mass-production is possible. Therefore, a trend towards more dynamic architectures is visible, such that after production of the chip the functionality can be changed by means of software updates.

An example of devices containing application specific processors, is the first generation mobile-phones, which are capable of making phone calls and sending text messages. Hardware chips inside these mobile-phones are very energy efficient, but it is not possible to extend the mobile-phone with additional software applications.



• *Figure 5: Current processor architectures, embedded in current user electronic devices, show a trend towards more flexibility. By example of: H. Corporaal and B. Mesman (TU/e)*

These days, mobile-phones are very flexible and allow installation of software applications on user demand. This is at the cost of energy efficiency.

Energy savings is a topic getting a lot of attention in current research. Combining research in energy efficient computing with programmability means that the gap between general purpose and application specific processor designs becomes smaller.

2.4.2 ROLE OF THE OPERATING SYSTEM

An operating system provides an interface between hardware and software by providing [8]:

1. **Abstraction:** implement generic concepts and handle complexity. For example: implement basic hardware drivers, such that application engineers can abstract from hardware details.
2. **Virtualization:** from the point of view of an application or an user, it looks like the operating system provides each of them with the full system resources. Furthermore, the operating system provides the same abstraction for multiple underlying hardware systems.
3. **Resource management:** resource management aims at sharing resources among different application within the system. Sharing of resources includes allocating the resources as efficient as possible and prevent applications from abusing resources (protection of resources).

Important in operating system design is the distinction between policy, providing the abstraction, and the mechanism, providing the means. The decision scheduler, as introduced in Section 2.3.2, is an application specific component, which specifies the *policy* (or *strategy*) to distribute the resources. A *policy* describes how a system should behave. In order to realize a policy, appropriate mechanisms must be provided by the operating system or a layer built on top of the operating system. A *mechanism* provides the instrument to implement a policy.

Topic of research are resource management mechanisms to allow efficient allocation of processor resources for multimedia applications. The mechanisms required to dynamically allocate resources are built on top of the operating system, assuming that operating system sources are not available and aiming at general applicable approaches. This preserves platform abstraction for the application.

2.4.3 REAL-TIME SYSTEMS

A real-time system is not only characterized by its functional requirements, but also by its timing requirements. This means that tasks in such a system are bounded by a deadline. By violating this deadline, the functional output of this task becomes less valuable, irrelevant or even wrong. A lot of research has been done in order to achieve predictability in real-time systems, such that correctness in this temporal behavior can be guaranteed [11]. However, most common techniques described in literature assume that a system is formed by a fixed set of tasks. These tasks, assumed to be independent of each other, execute the system functionality, and are often categorized as periodic or a-periodic. A *task* is defined as a sequence of actions that must be performed in reaction of an event, whereas a *job* is defined as an instantiation of a task.

Guaranteeing timeliness constraints requires analysis of a real-time system before deployment. Methods for analysis described in literature, such as by Buttazzo [11], rely on a task model description based on the following task characteristics:

1. Timing parameters are known-upfront (for example: worst-case execution times, periods and relative deadlines). These parameters (dependent on the scheduling policy) are required to provide the analysis whether a feasible schedule is possible with the tasks composing the system.
2. Worst-case execution times are close to average-case execution time. When the difference between worst-case and average-case execution times is relatively high, it leads to low processor utilization. In order to guarantee timing constraints, the worst-case scenario must be taken into account. When accounting the worst-case scenario, it means that an average case job leaves a lot of unused processor time.
3. Each task is assigned a priority, which is determined by the scheduling policy. The priority can be fixed or dynamically assigned. Tasks are assigned to the processor according to their priority.
4. Tasks are assumed to be independent. Precedence relations or communication between tasks can influence the schedulability of the system.

Since the main task of a real-time operating system is to provide predictable task latencies, this requires appropriate implementations of system calls, memory management and offered primitives [11]. All kernel calls should have a bounded execution time. Memory management must be done deterministically. Constructs as semaphores and dynamic datastructures must be prevented, since these constructs do not allow analysis of resource allocation upfront.

On the one hand, high quality video applications also require precise timeliness constraints, for example: periodically a frame is computed with a predefined deadline. On the other hand, multimedia applications do not fit in the classical real-time model. Media processing applications are typically organized as chains of data driven tasks [38], which violates the assumption of independent tasks. Furthermore, the data-dependent computational load of a signal processing algorithm violate the assumption that worst-case execution times are close to average-case execution times. Therefore, scalable alternatives are developed to fit multimedia processing applications in a real-time environment, such as reservation based mechanisms (see Section 2.5) with corresponding quality modes.

It might be desirable for a system to accommodate changes of functionality or behavior to respond to external events. It is possible to include different flavors of tasks in a single task set, however this approach does not scale very well. Alternatively, the task set can depend on the *mode* the application is running in. An event in the system can trigger the application to switch between two application modes, which is called a *mode change request*. However, in literature several techniques are described in order to achieve predictability during the transition of application modes, see [31]. An overview of the applicability of investigated resource allocation mechanisms in applications supporting different modes of operating is described in Appendix A.

2.5 RESOURCE RESERVATIONS

Resource reservations is an approach to manage resources at the level of real-time operating systems, by means of providing a virtual platform among competing applications within the system [28]. A reserve represents a share of a single computing resource [29], for example: processor time, physical memory or network bandwidth. A reserve is implemented as a kernel entity, which means that it cannot be counterfeited. This report focusses mainly on resource management mechanisms for processor time, for which similar mechanisms are required as resource reservations.

As described by Rajkumar et al. [29], the resource reservation paradigm relies strongly on the implementation of mechanisms for:

1. **Admission Control:** This mechanism contains a feasibility test, which checks how much resources can be consumed. A reservation is only granted if it does not exceed the maximum amount of available resources and does not jeopardize overall system performance.
2. **Scheduling:** Defines the policy to distribute the available resources over competing applications.
3. **Enforcement:** These mechanisms must ensure that the amount of resources used by an applications does not exceed the reservation.
4. **Accounting:** Keeps track of the amount of resources used by each application. This information can be used by the scheduling and enforcement mechanisms.

Resource reservations provide temporal protection among different applications. If an application violates its timing constraints, other applications will not suffer from this. This allows independent design, analysis and validation of real-time applications. Each application can even use a local scheduler, which subdivides the acquired resources among its components. The integration of multiple applications in such an hierarchical scheduled system consists of schedulability test on the system level, verifying that all timing requirements are met [5].

2.5.1 HIERARCHICAL SCHEDULING

Behnam et al. [5] present an hierarchical Scheduling framework built on top of VxWorks, without modifying the kernel source code. The implemented framework provides temporal isolation of programs by means of periodic servers. These servers are the highest level tasks, which are scheduled by the *fixed priority scheduler* of VxWorks, and their budget is replenished every constant period.

Fixed Priority Scheduling (FPS) means that the highest priority ready task will always execute. If during runtime a task with a higher priority becomes ready to execute, the scheduler stops the execution of the running task and assigns the processor to the higher priority task.

Using the fixed priority scheduler of VxWorks, Behnam et al. [5] describe how to implement arbitrary scheduling policies. This is done by using a interrupt service routine which manipulates tasks in the ready queue. This interrupt service routine can add tasks to the ready queue, remove tasks from the ready queue and can change priorities of tasks. Since periodic servers can be scheduled as periodic tasks, the model is easily extended to provide an hierarchical scheduler. This local scheduler can be implemented by:

1. changing the ready queue, which means that if an server is preempted, all its tasks are removed from the ready queue and marked with a suspend flag;
2. decreasing the priorities of the tasks corresponding to the preempted server to the lowest priority and schedule the tasks of the higher priority server with higher priorities, according to the local scheduling policy.

The advantage of the priority manipulation approach is that unused processor time can be consumed by other servers. The disadvantage is that the overhead of the system scheduler will also increase due to larger ready queues, which have to be sorted.

2.5.2 MULTIMEDIA SYSTEMS

The advantage of independent development of applications and guaranteed resources provide a good basis for multimedia applications. Scalable multimedia applications can trade-off resource usage for quality output. By means of well defined interfaces, applications can require an amount of resources from the global system. This allows scalable multimedia applications to negotiate the amount of reserved resources for quality of service. If these resources demands are subject of change during run-time, this requires a run-time admission control mechanism. Therefore, we will assume a fixed amount of available resources per component per activation period. Steffens et al. [35] present propositions and considerations for resource reservations in different applications domains. Most important in the field of multimedia processing include:

1. How do multimedia application adapt their resource needs?
2. How are non-used resources efficiently re-allocated?
3. How do interfaces look like to monitor resource usage?
4. How do interfaces look like to negotiate resource requirements?
5. How is dealt with multiple resources, for example multi-processor systems?
6. What is the granularity at which the resources are allocated? This has influence on efficient use of cache, pipelined computations, and caused control overhead.

Successful application of a resource reservation scheme is based upon operating system support in cooperation with communication via well defined interfaces with the applications. Mercer et al. [26] describe processor reservations for multimedia systems. The analysis is based upon WCET estimations. When the computation time is not constant, a conservative worst case estimate reserves the necessary processor capacity, and the unused time is available for background processing. In multimedia processing, worst case execution times are typically not close to the average case computation times.

The reservation scheme presented by Mercer et al. [26] is implemented in the kernel of a real-time operating system, with the purpose to support higher level resource management policies. A quality of service (QoS) manager could use the reservation system as a mechanism for controlling the resources allocated to various applications. The QoS manager would translate the QoS parameters of applications to system resource requirements (including processor requirements), possibly with the cooperation of the application itself.

Pérez et al. [28] describe resource reservations for shared memory multi-processors for embedded multimedia applications. The processor reservation scheme is based upon a fixed priority hierarchical scheduling approach to be implemented in a real-time operating system environment, comparable to Behnam et al. [5]. In order to tackle the problem of cache poisoning in a shared cache between multiple processors and the bottleneck of a shared memory bus to access the main off-chip memory, the virtual platform as provided by the resource reservation scheme is extended with algorithms in hardware to divide the available memory resources. By managing the memory resources in an appropriate way, the processor utilization can be increased, due to less stall cycles. Memory reservation schemes are suggested to be applied in hardware as follows:

1. The cache reservation scheme consists of separated domains, which means that each applications is allowed to use a specific part of the cache. The cache partitioning scheme can be extended to overlapping domains, to obtain a better cache utilization.
2. The memory reservation scheme is based upon the distinction between periodic, high bandwidth requests versus sporadic, bursted, low latency requests. High bandwidth request have a fixed priority. The memory controller accounts the amount of cycles used by the low latency memory requests and manipulates the priorities according to its budget. When the budget is depleted for a low latency request, the priority for the low latency request is decreased, such that high bandwidth request get precedence.

2.6 APPLICABLE WORK

In this section two different approaches for scalable video applications are described:

1. Quality modes of operation, which are coarse-grained, discrete quality gradations. Each mode of operation requires a semi-static amount of resource availability, whereas mode transitions introduce additional complexity to the systems. A quality mode is typically selected on the boundaries of a frame and a frame computation is typically work-preserving.
2. Priority processing, which introduces fine-grained steps of increasing quality levels. Priority processing processes video content in decreasing priority order, allowing preliminary termination after availability of a basic output. The preliminary termination property makes this type of applications applicable to handle fluctuating resource availability in a self-adapting way. The output quality of priority processing algorithms is determined by the resources that are allocated during run-time.

When multiple scalable video algorithms are competing for a single processor resources, control strategies are required to balance the available resources as efficient as possible, In [39], a control strategy is presented for a single, work-preserving scalable video algorithm. The strategy selects a quality level for processing the next frame upon completion of the previous frame, i.e. synchronous with processing. When a deadline is missed, e.g. a frame is not completed within its budget, a frame might be either skipped or the previous frame can be repeated instead of showing no output at all. The latter option requires additional buffering of frames.

In this report, we consider mechanisms for dynamic resource allocation to multiple, scalable priority-processing video algorithms. The control strategy [32] selects the algorithm to execute next upon completion of a time-slot, i.e. synchronous with time. Because each algorithm must provide output at the end of each period, we need appropriate mechanisms to preliminary terminate the scalable video algorithms. The preliminary termination mechanism introduces system overhead due to the required time synchronization enforced by this mechanism. Whereas Wüst et al. [39] assume a fixed amount of resources for a single algorithm, the priority processing algorithms are capable of handling fluctuating resource availability.

Dynamic resource allocation has much in common with reservation-based resource management [29]. Resource reservations is an approach to manage resources at the level of real-time operating systems, by means of providing a virtual platform among competing applications within the system [28]. Guaranteed resources by means of reservation-based management can be an additional technique to dynamic resource allocation. The distinguishing characteristics with dynamic resource allocation approach on a time-slot basis, as applied in the priority processing application, are:

1. the lack of admission control. Admission control is superfluous for priority processing due to its capabilities to handle fluctuating resource availability.
2. the need for preliminary termination of scalable video algorithms, enforcing a synchronization between processing and time.

In case of priority processing, the roll-forward mechanism is a straightforward action upon preliminary termination, meaning that a new frame-computation is started upon preliminary termination. When a pending job is terminated, it means that its budget is depleted and that a next job must be started.

Summing up the demands, the priority processing applications must be analyzed and optimized for the following key aspects:

1. Efficient distribution of processor time over the competing algorithms on a time-slot basis, according to the pre-defined scheduling policy;
2. Preliminary termination of all signal processing tasks (active and waiting tasks) by an external signal from controller if the budget is depleted.

3. RESOURCE MANAGEMENT MECHANISMS

This section first presents assumptions on the priority processing application and its environment, see Section 3.1. Thereafter, a general architecture for scalable multimedia applications is presented, see Section 3.2.

Section 3.3 describes the taskmodel within a priority processing application. The task model is extended with mechanisms to efficiently manage resources, which relies on the mechanisms for monitoring, preliminary termination and resource allocation, as described in Section 3.4 through Section 3.6.

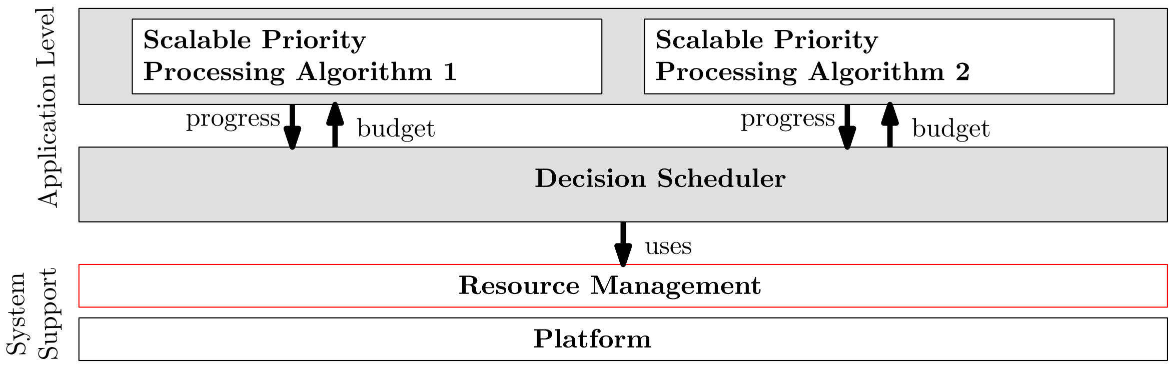
Finally this section summarizes different design considerations for the priority processing applications, which are investigated in Section 3.7.

3.1 ARCHITECTURE ASSUMPTIONS

A1. *An application is composed of a decision scheduler and one or more (priority processing) algorithms.*
A2. *All algorithms are known upfront by the decision scheduler.*
A3. *Each algorithm is mapped on an independent task.*
A4. *Priority processing algorithms are characterized by three phases: basic, analysis and enhancement.*
A5. *An algorithm can be preliminary terminated after the basic function, e.g.: during the analysis or enhancement phase.*
A6. *The decision scheduler divides the available resources of the algorithms.*
A7. *Each algorithm within the application is each period provided a budget.*
A8. *The budget is divided over the algorithms on the basis of fixed-size quanta, e.g.: time-slots.*
A9. *All algorithms are synchronous with the period, which means that each period the algorithms start with a new frame and at the end of a period the frame is terminated.*
A10. *Each period the input for the algorithms is available and the output can be written to an output buffer, which means that algorithms are not blocked by input and output.*
A11. *Each algorithm accounts its own progress and updates the progress during runtime.*
A12. *The application is mapped on a multi-processor platform.*
A13. *The platform provides a fixed priority scheduler.*
A14. *The algorithms have the complete processor at their disposal.*

3.2 GENERIC ARCHITECTURE

A generic architecture for scalable priority processing applications is shown in figure 6. In this model, a single application is assumed to consist of one or more algorithms. The algorithms are controlled by an application dependent controlling component, named the *decision scheduler*. The decision scheduler is expected to know all priority processing algorithms upfront. The algorithms on their turn are assumed to provide the decision scheduler with feedback information, such that the decision scheduler can make better decisions about the division of available resources among the algorithms. The decision scheduler uses the system support layer, which provides interfaces for mechanisms which are needed to enforce real-time requirements.



• *Figure 6: Layered reference architecture consisting of a system support layer with an application layer on top. Application components are indicated with a gray background. The resource management layer on top of the platform (red rectangle) indicates the area in which mechanisms are implemented to provide platform abstraction and efficient resource allocation and deallocation for the multimedia application. This is the main subject of research in this report.*

The layered architecture is based upon a *uses-relation*. The Application components, i.e.: algorithms and decision scheduler, can use system support components, which on its turn use platform internals. The system support layer consists of the resource management and platform layers. The resource management layer can be seen as a library, with corresponding interface definitions, which can be included by the application. The decision scheduler uses the library by using the *#include* statement in C/C++ and the compiler links the application against the corresponding library.

The following parts can be identified from this layered architectural application model:

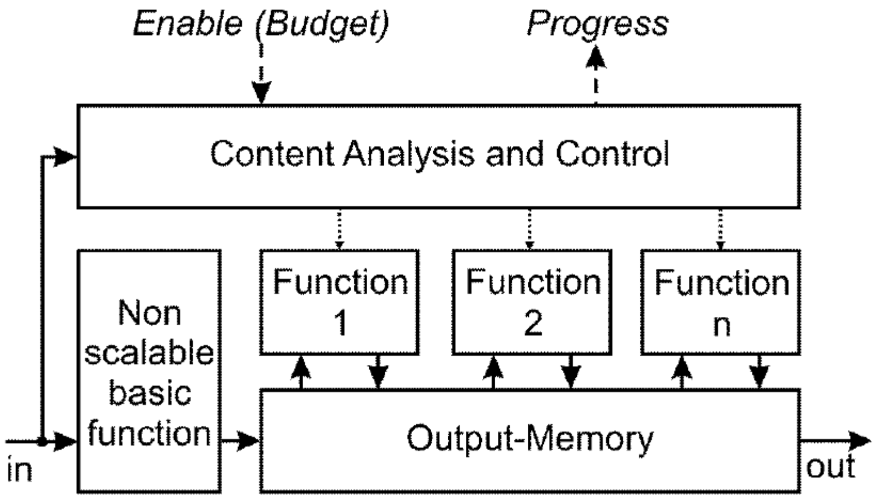
1. **Application Level**
   An application consists of components, which can either be control components running at the highest priority, or functional components (scalable algorithms) performing the actual work to be done.
   (a) **Decision Scheduler**
   The decision scheduler is assumed to be an application dependent control component running at the highest priority, which is capable to assign budgets and make decisions to trade-off the overall resource needs of the system versus the output quality of the system. This includes the application specific scheduling policy. This report abstracts from the internal design of the decision scheduler, but emphasizes on the mechanisms needed to support local resource management.
   (b) **Algorithms**
   An application consists of one or more scalable algorithms. An algorithm is mapped directly onto a task. The behavior of each of these algorithms is defined by a characteristic progress function and has real-time requirements. Section 3.3 describes the structure of a task in more detail.
2. **System Support Level**
   The system support level consists of two layers: the resource management layer is built on top of the operating system, which implement interfaces to abstract from the platform.
   (a) **Resource Management Layer**
   The decision scheduler needs mechanisms for dynamically allocating and deallocating processor resources over the scalable algorithms. An additional resource reservation scheme based upon mechanisms as described by Rajkumar et al. [29], and for example as implemented by Behnam et al. [5], can guarantee availability of resources for an application. This includes the availability of primitives to implement mechanisms for: scheduling, accounting, enforcement and admission control, as introduced in Section 2.5. However the mechanisms look similar, the mechanisms described in Sections 3.5 through 3.6 aim to provide efficient resource management mechanisms to dynamically allocate the available resources for competing algorithms within an application. The decision scheduler is allowed to define a policy to distribute the processor resources over the components, such that appropriate scheduling mechanisms are required.
   (b) **Platform layer**
   The platform provides the primitives which can be used by the decision scheduler (eventually via the resource management layer on top of the operating system) and the resource management layer itself to control the application. The platform consists of hardware and (optionally) a corresponding (real-time) operating system, as described in Section 2.4. In case of a resource centric operating system (kernel support is provided for resource reservations), the provided platform is virtual, which also entails virtual timing issues, which are briefly discussed in Section 6.2.2.

3.3 TASK MODEL

An application consists of several tasks, representing a priority processing algorithm. The tasks in multimedia applications typically behave periodic. A scalable priority processing application can be composed of several algorithmic stages, or functions. Figure 6 shows the general architecture of a priority processing application, whereas Figure 7 shows the structure of a single algorithm. Example video processing algorithms are enhancement filters and deinterlacing algorithms which use the scalable priority processing method, as described in Section 2.2. An example application is a picture-in-picture processing of a deinterlacer and a sharpness enhancement algorithm.

A task is composed from sub-tasks, which occur in the implementation as functions. An algorithm in the priority processing application is composed from the following functions, as shown in Figure 7, which are executed consecutively each period:

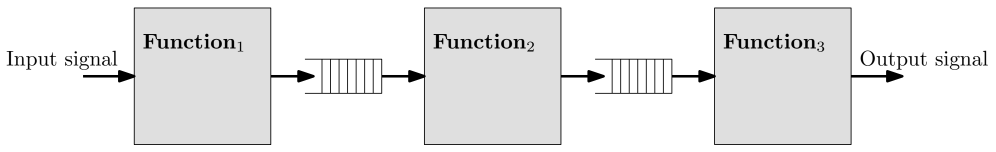
1. **Non-scalable basic function:** The non-scalable basic function generates a lowest quality frame output and can not be preliminary terminated. The lack of preliminary termination during this stage, requires at least the availability of enough budget to complete the basic functions of all algorithm during a period.



• *Figure 7: Structure of a scalable priority processing algorithm, source: [17]*

2. **Content analysis function:** The content analysis and control part sorts the content by importance, which is the priority order in which the scalable part will process the picture content. When the deadline expires during the content analysis function, the remaining work of the current frame, including the scalable functions, are skipped.
3. **Scalable functions:** Functions, numbered from $1 \ldots n$ (see Figure 7), are scalable sub-tasks, which enhance the output quality. Each of these functions include algorithmic signal processing complexity and represent a stage in the scalable part of the priority processing algorithm. When such a function is preliminary terminated, the remaining part of the pending frame is discarded and a roll-forward action is performed to the next frame, similar to the content analysis function.

In soft real-time video applications a typical component is composed of different algorithms as shown in Figure 8, which is similar for the composition of the functions described for priority processing algorithms. These functions implement the actual signal processing and each function represents a subtask.



• *Figure 8: Typical video application component composed of different algorithms which represent a subtask.*

All algorithmic stages, or functions, will be executed consecutively. When an algorithm is requested to terminate, the complete job, which represents an instantiation of a task, is terminated and reset to its initial point. This is a *roll forward* action, which allows to skip computational parts. Upon preliminary termination it must be ensured that all buffers are flushed and it must be decided which sub-task is going to be scheduled next. For example: consider the scenario in which the algorithm, sketched in Figure 7, with three scalable functions, as sketched in Figure 8, is currently running $function_2$ when a request arrives to preliminary terminate the component; The preliminary termination causes skipping the computation of $function_3$ and the basic function is the first to be scheduled to start with the next frame.

The computation time of a task i consist of a basic part and scalable parts and is denoted as C_i, see Equation 2. The analysis function is considered as a member of the scalable part of the algorithm. Note that it is not feasible to compute the worst-case computation time, since multimedia processing algorithms are characterized by heavily fluctuating, data-depend workloads, but for our real-time model this is neglected.

$$C_i = C_{i,basic} + C_{i,scalable} \tag{2}$$

All tasks in the application have the same phase, with activation time t_a; absolute deadline t_d and relative deadline D. The budget within a period is divided in fixed-sized quanta, named time-slots. The time-slots are assigned to the algorithms by the decision scheduler. It is assumed that periods, T, are equal to the relative deadline, D.

$$D = T \tag{3}$$

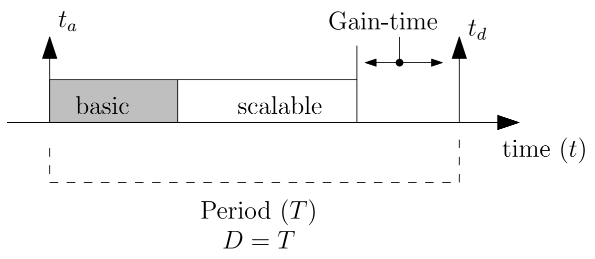
Assume that an application is composed of N algorithms. The basic sub-tasks are required to guarantee a minimal output. It is assumed that within a period, there is enough time to perform all N basic sub-tasks with computation time $C_{i,basic}$, see equation 4.

$$\sum_{i=0}^{N-1} C_{i,basic} \leq D \tag{4}$$

Given an amount of available time resources within period T, we can distinguish two cases:

1. The scalable algorithm does not consume its entire budget, which means that there is *gain-time* [39] available for other tasks in the system, see the example in Figure 9. The way of reusing gain time must be specified by the scheduling policy. Figure 9 sketches the scenario in which a job instantiation requires less computation time than the available time resources, formalized in Equation 5.

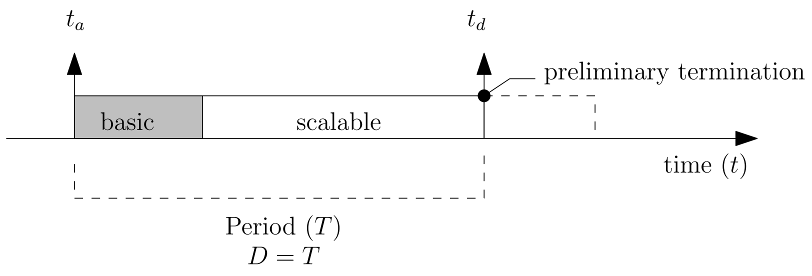
$$\sum_{i=0}^{N-1} C_i \leq D \equiv \sum_{i=0}^{N-1} C_{i,basic} + \sum_{i=0}^{N-1} C_{i,scalable} \leq D \tag{5}$$



• *Figure 9: Example of a task, composed of a basic part and a scalable part. The total computation time leaves gain time, which can be used by the scheduler in favor of other tasks in the system. The activation of a task is at time t_a. In this example, deadlines are equal to periods, which means that deadline t_d is the activation time of the next period.*

2. The available budget is not sufficient to complete the scalable algorithm entirely. Figure 10 shows an example of a task consisting of a basic part and a scalable part. The total worst-case computation time of the task, equal to the sum of both sub-taks, exceeds the available time within the deadline, as summarized in Equation 6. In case of expiration of the deadline, all non-completed tasks are preliminary terminated.

$$\sum_{i=0}^{N-1} C_i \geq D \equiv \sum_{i=0}^{N-1} C_{i,basic} + \sum_{i=0}^{N-1} C_{i,scalable} \geq D \tag{6}$$



• *Figure 10: Example of a task, composed of a basic part and a scalable part. The total worst-case execution time exceeds the dead-line. The activation of a task is at time t_a and deadlines are again assumed to be equal to periods.*

Table 2 summarizes the division of responsibilities between application engineering and system engineering, e.g.: policy versus mechanism. The priority processing application defines that upon expiration of a periodic deadline, a frame computation must be preliminary terminated. This requires appropriate mechanisms. The division of the resources among competing algorithms within the application is made by a quality manager (the decision scheduler), which requires appropriate mechanisms to allocate the (processor) resources. Finally, the allocation of resources is based upon monitoring information, indicating the state of the components. This requires accounting of consumed time by each component and activation of the decision scheduler on a time-slot basis.

Distribution of Responsibilities:

Application Specific (Policy)	**System Support** (Mechanism)
Roll-forward when deadline reached	Preliminary termination
Resource distribution (Reinforcement Learning policy)	Allocation of processor
Monitor Progress values	Account consumed time and activate decision scheduler (time-slots)

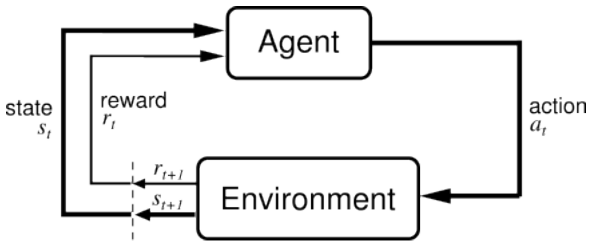
• *Table 2: Overview of application versus system specific responsibilities. The application specifies the policies to describe its desired behavior whereas the system should provide appropriate mechanisms to enforce the policy.*

3.4 MONITORING

Mechanisms for monitoring essentially keep track of two attributes:

1. the amount of processing time (resources) consumed by each algorithm.
2. the reached state of the algorithm, measured by the progress value.

The decision scheduler can be seen as an agent, which monitors the environment and tries to maximize overall performance based upon the gathered information. The monitoring scenario is schematically shown in Figure 11. This schematic is characteristic for reinforcement learning problems as introduced in Section 2.3. The decision scheduler is provided with objective quality metrics, which represents the reward, and tries to optimize the overall quality metric.



• *Figure 11: The agent observes the reached state of the environment and tries to maximize the overall reward over time by choosing appropriate actions. Source: [4].*

By accounting the amount of time consumed by each component and the reached progress value within an amount of time, a new state of an application is defined. The decision scheduler is responsible to choose an appropriate action (allocate resources for a component) based on the available information in the reached state. This implies that the accounting of consumed time is elementary for the resource allocation mechanism (see Section 3.5). Also preliminary termination (see Section 3.6) relies on the availability of timing information and therefore requires accounting of consumed time by all competing algorithms, i.e. periods.

Ideally, the operating system provides an interface to query the amount of time consumed by a task. If this is not the case, timers, as provided by (real-time) operating systems, can be used in order to measure the amount of time consumed by a component (task) or sub-task. Issues concerning time-measurement on X86 machines are discussed in Appendix C.

3.5 ALLOCATION OF RESOURCES

When a single processor has to execute a set of concurrent tasks, the processor has to be assigned to the various tasks according to a predefined criterion, called a scheduling policy [11]. The scheduling policy defines the way to distribute the available resources over competing components (algorithms) within an application. Mechanisms for allocating (processor) resources implement facilities to bind an unbind a task to the processor according to the scheduling policy. The decision scheduler within the priority processing application, implements different scheduling policies. In order to manage the allocation of resources according to the scheduling policy, the following mechanisms are considered:

1. **Cooperative, polling based scheduling:** the task checks on regular intervals a flag, indicating whether it is still allowed to use computational resources. This approach relies upon cooperation between control and application components, see Section 3.5.1.
2. **Suspending and resuming of tasks:** The ready-queue of the global fixed priority scheduler is altered by explicitly suspending and resuming tasks, see Section 3.5.2.
3. **Use global FPS by means of priority bands:** Manipulate the priorities of the tasks according to the local scheduling policy, see Section 3.5.3

3.5.1 COOPERATIVE SCHEDULING

By letting the application poll on regular intervals, the application is not fully preemptive. This approach is a trade-off between fully preemptive scheduling and non-preemptive scheduling. The analysis for real-time tasks with cooperative scheduling (or deferred preemption) is provided by Bril et al. [9].

The advantage of cooperative scheduling [10, Section 13.12.1] is that resources are deallocated by a component at predefined moments in time, such that mutually exclusive, shared resources other than the processor are easier to manage, without the need of additional resource access protocols.

The disadvantage of the cooperative scheduling approach is that dynamic allocation of resources can not be guaranteed. In a cooperative scheduled system the application has to give up its resources, which means that preemption can not be enforced by the scheduler. Furthermore, the polling based mechanisms disturb the typically pipelined computations of signal processing algorithms.

Another consideration is the scheduling on a time-slot basis, such as a Round-Robin (RR) policy for tasks having the same priority. In case of Round Robin scheduling, preemption of tasks might occur at a fine-grained intervals, which imposes high requirements to the polling mechanism.

Bergsma et al. [7] tested the preemption point overhead (polling) by implementing cooperative scheduling on a real-time operating system (RTAI) and shows that the overhead is in the order of 440 ns per preemption point. This is relative high compared to the small quantified time-slot based scheduling in the priority processing application.

Overall, it does not seem a straight-forward option to apply a polling based mechanism to support the scheduling policy. Although, this mechanism will be considered as a valid option to preliminary terminate a job, see Section 3.6.1.

3.5.2 SUSPENDING AND RESUMING TASKS

Explicitly suspending and resuming tasks, means that the ready-queue of the scheduler is changed. When suspending a task, it is removed from the ready-queue. Similarly, when resuming a task, it is added to the ready-queue. The suspend-resume mechanisms are described in the the context of an hierarchical scheduling framework by Behnam et al. [5].

The advantage of explicitly suspending and resuming tasks is that it reduces the length of the ready-queue when tasks are removed. A smaller ready queue means that the scheduling takes less overhead, such that associated system overhead for scheduling is reduced.

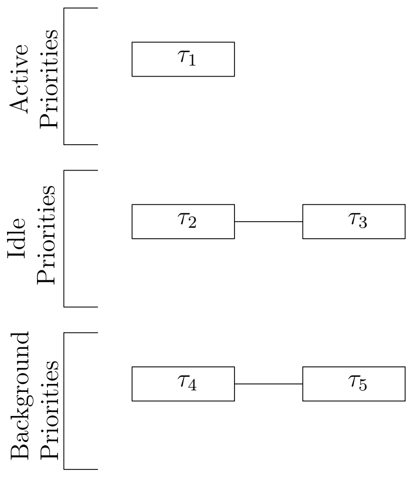
The disadvantage of this approach is that when other components leave gain-time, these expensive processor cycles are lost, because suspended tasks can not be scheduled.

3.5.3 PRIORITY BANDS

As an alternative to suspending and resuming tasks, the priorities of a task can be manipulated according to the local scheduling policy. The global fixed priority scheduler assigns per definition the processor cycles to the highest priority ready task. Gain time can be consumed by lower priority tasks.

The use of priority bands is shown in Figure 12, in which three *priority ranges* (bands) are shown, in which *each priority* has its dedicated ready-queue:

1. **Active Priority:** tasks in the ready-queue given the highest priority according to the scheduling policy. These tasks (on uni-processor scheduling most likely a single task) are assigned cpu time.
2. **Idle Priority:** when the active tasks do not consume their total budget, the gain-time is consumed by the tasks at idle priority. Note that this priority range can be used to include an artificial task which consumes all gain-time to prevent other tasks from consuming gain-time. This is especially usefull when tasks are assigned on basis of time-slots, such that the accounting mechanisms, see Section 3.4, are kept relatively easy.
3. **Background Priority:** the tasks in the background are waiting for processor time, but are blocked due to the higher priority tasks in the active and idle priority ranges.



• *Figure 12: Example of the use of priority bands in which three ranges of priorities are indicated: 1) active priority, which are the tasks in the ready-queue given the highest priority according to the scheduling policy; 2) idle priority, which are tasks consuming the gain-time of the highest priority, active tasks; 3) Background priority, which are the tasks waiting for processor time but blocked by active priority tasks or idle tasks.*

Note that when the scheduling policy permits, the approach of using priority bands is applicable for multi-processor scheduling. Scheduling must be supported by the policy and requires careful analysis. When using one or more additional processors for background processing, this can influence:

1. the schedulability of the task-set, as sketched by Buttazzo [11, Section 2.4];
2. the performance of the highest priority task due to cache pollution (in case of shared caches) and possible overload situations of the (shared) memory bus. Pérez et al. [28] propose a predictable solution for managing shared memory resources in embedded environments.

3.6 PRELIMINARY TERMINATION OF JOBS

Mechanisms for preliminary termination must ensure that the amount of resources used by a component does not exceed the amount of resources that the component is allowed to use. Each components must reside within the resource budget, as determined by the decision scheduler.

The controller (decision scheduler) assigns budgets to tasks in the video application. The task must stop its activities when the budget is consumed. This can be achieved by letting the tasks poll on regular intervals for the available budget. The task must give back control to the controller component when the budget is not sufficient to do additional work, e.g. upon expiration of the deadline.

An alternative approach is that the controller component signals pending tasks to stop their activities as soon as their budget is consumed. Signalling requires transfer of control between control component and the different tasks of the multimedia application.

3.6.1 COOPERATIVE PRELIMINARY TERMINATION OF JOBS

By cooperative termination of jobs, the application components can decide themselves when to terminate, such that a advantageous point in the code can be chosen. For termination of jobs, similar issues appear as described for preemption, see Section 3.5.1.

On the other hand, preliminary termination is assumed to take place at lower rates in time compared to preemption, such that there is a larger freedom to choose the granularity of termination. On regular intervals in the component code, there must be check to see whether the budget is depleted. This can be implemented by means of different mechanisms:

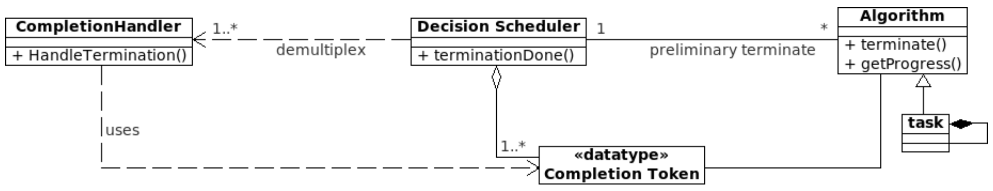
1. checking a shared boolean, variable between component and decision scheduler which indicates whether the budget is depleted.
2. checking whether an event occurred, where the event is triggered by the application and indicates the end of a period (for example: Microsoft Windows Events).
3. checking whether a message has been send by means of a mailbox system.

The granularity on which a termination indicator must be checked, allows a trade-off in computational overhead and termination latency. All implementations require the component to check at regular intervals for depletion of the budget, whereas the use of an appropriate mechanism is dependent on the platform. As a most general form we consider the shared variable solution, for which an example is shown in Section 3.7.

3.6.2 A-SYNCHRONOUS PRELIMINARY TERMINATION OF JOBS USING EXTERNAL EVENTS

The decision scheduler can request a component to preliminary terminate its activities. The termination request on it self is an a-synchronous event. Figure 13 shows the a-synchronous event handling pattern [34, Section 3]. The actors in this model are:

1. the decision scheduler, which initiates the termination request.
2. the components (or task), which is composed from sub-tasks. The component receives the termination request and must reset the complete pipeline, including buffers.

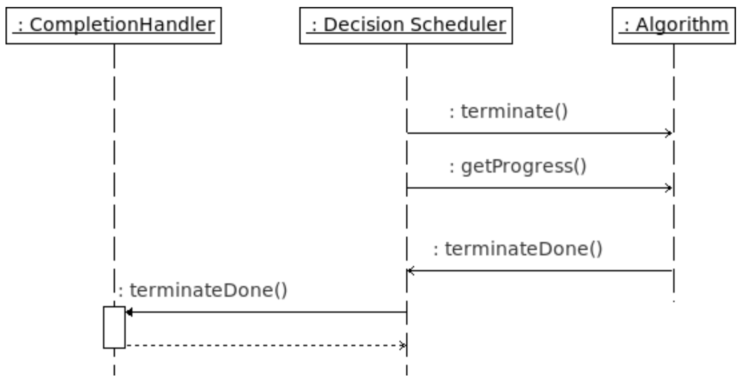


• *Figure 13: Classmodel for the a-synchronous request handling. Such an a-synchronous request can represent a job termination request or a mode change request.*

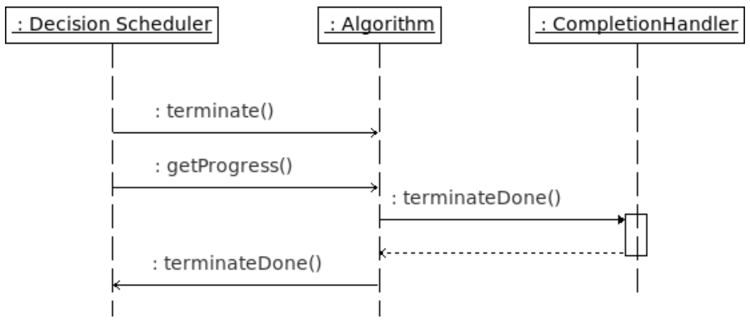
The behavior of an a-synchronous event handling mechanism can be described by two different scenarios, as summarized in Figure 14:

1. Before the decision scheduler performs a termination request, it has to create a completion token. This token is responsible for identifying the completion handler associated with the termination request. In practice, this token would appear as a pointer to a handler function occurring as an argument in the request function.
2. When request is performed, the token is passed to the component receiving the request.
3. The decision scheduler can proceed with processing operations while the component processes the request. Figure 13 shows that a component can only be associated with one token at a time. This means that multiple request to the same component will be discarded, except for the first request. A job can only be terminated once.
4. When the component completes the requested preliminary termination action, the component sends a response to the decision scheduler containing the completion token. Typically, a handler function has to be executed to complete the event, for example executing roll-back or roll-forward operations. There are two conceivable scenarios to process the event-handler:
   (a) the event handler is executed in the context of the main-process (which is assumed to implement the decision scheduler), see Figure 14a. When the event handler needs to reset buffers of a preliminary terminated component, locations of the buffers must be shared with the decision scheduler. Sharing private component information might be non-desired.
   (b) the event handler is executed in the context of the requested component, see Figure 14b. In this case there is no need to share internal component information with the decision scheduler.

   Note that it is desirable to do as less additional work as possible in the completion handler as it is extra system overhead.



(a) The event handler is executed in the context of the decision scheduler.



(b) The event handler is executed in the context of the requested component.

• *Figure 14: Behavioral sequence of an a-synchronous request initiated by the decision scheduler. Note that the involvement of the completion token is discarded, because it is assumed to be passed as an argument between component and decision scheduler.*

GNU/Linux operating systems provide signalling mechanisms as described by Thangaraju [36], which is a mix of both variants described above. The signal handler is executed in the context of the receiving thread, but thread internals are not accessible within the signal handler. Proper transfer of control is enforced by structuring the programs similar to the recovery block scheme proposed by Randell [30], see Appendix B.3. Since signalling can cause the receiving component to terminate at any moment in time, it is required to have independent components, such that the domino effect is prevented as a result of corrupted component states.

3.7 TOWARDS A MULTIMEDIA APPLICATION PROTOTYPE

In this section, the code structure of a scalable video task is considered. Next to the code structure, it is described what the typical modification and considerations are to optimize the control of the scalable tasks.

3.7.1 TOWARDS A PROTOTYPE FOR SCALABLE SUBTASKS

There are several methods to preliminary terminate sub-tasks. It is conceivable to poll on regular intervals whether there is a sufficient time budget to execute a new sub-job, which for example can represent a block computation. The pseudocode for these scalable functions is shown in Source Code 1, in which the *continue*() functions implements the budget polling mechanism.

```
void sub_task_i_j()
{
  initialSetup();
  while(continue() && !end_of_frame())
  {
    process_next_block();
  }
}
```

• *Source Code 1: Expected structure of a scalable sub-tasks in multimedia applications. This function includes a polling function continue() which checks the remaining budget at regular intervals (in this case at the granularity of a block-*

computation).

The polling mechanism is considered time consuming, and therefore we want to get rid of this polling function. This is achieved by introducing a signal handling approach as shown in Figure 13, in which the control block sends a signal to a function block. Since it is most likely that every function is implemented using separate threads, inspecting the remaining budget at regular intervals, as shown in Source Code 3, implies the inspection of non-local flags. This is probably the most expensive part of the polling function.

This signal causes a handler to interrupt the job and perform a roll-forward action. This software interrupt is triggers the completion task. In the easiest case, this completion handler this jumps to the corresponding progression point. This means that the function has to go to its recovery point upon such an termination request. Source Code 2 shows a candidate recovery point. Assuming that some initial setup must be done on a per frame basis, this point in the code is the only correct place to start a new frame.

```
void signalHandler(int signal)
{
  /* Jump to progression point */
}

void new_function_i()
{
  /* (**) Progression Point */
  initialSetup();

  while(!end_of_frame())
  {
    process_next_block();
  }
}
```

• *Source Code 2: Target structure of a scalable function in a priority processing algorithm in which the polling function continue() is replaced with a signal mechanism initiated by the control block. This reduces computational overhead caused by the flag polling.*

Depending on the data consistency requirements, the amount of work to be done in this progression point might differ. For example: the way of transferring data to external memory influences the amount of work to be done in the progression point.

3.7.2 DATA TRANSFER TO OUTPUT MEMORY

The *process_next_block*() call in Source Code 2 processes a block of a frame and the results of this computation are typically transferred to the output memory which results visible output. There are two straightforward options to handle with data transfers, which both might deliver desired behavior:

1. **Write-behind:** Local computations are first stored locally. After finishing a block, the computational results are transferred to output memory. This approach has advantages on platforms where accessing local memory is cheaper compared to directly writing every bit to the output memory. This approach automatically preserves data consistency upon preliminary termination of the block computation, assumed that the block transfer from local to external memory can be done atomically. A straightforward manner to implement the write-behind approach is to put a *memcopy*() action in the progression point.
2. **Write-through:** Writing actions are performed directly to the output memory. On platforms having just a single memory space, the splitting in local versus external memory locations does not pay-off. In case that partially completed block computations lead to visible artifacts in the output, it is desirable to cast this memory access approach to the *write-behind* approach. When this is not the case, this approach might even give a performance enhancement on some platforms by saving memory copy actions.

Note that the *write-behind* approach requires time synchronization. In case of dependent jobs, time synchronization involves copying of data before a new job can start. In case of multi-processor systems both mechanisms for data management require time synchronization, since it must be guaranteed that data-buffers are accessed mutual exclusively.

3.7.3 FINETUNING GRANULARITY OF TERMINATION

There are situations in which it is an advantage to defer termination until a predefined moment in time using preferred termination points, as defined in Section 3.3. For example, the write-behind approach might cause additional control overhead. By allowing a small amount of work to complete before terminating the pending job, it might be possible to transform the write-behind approach to a write-through approach.

Another consideration to defer termination of a job, is that immediate termination causes the waste of invested computational resources. It is conceivable to give a job some extra budget to complete, and reduce the capacity of the next occurrence of the task as a pay-back mechanism.

In order to allow deferred termination of a job, the polling and signalling approach can be combined. When a signal occurs to terminate a pending job, the event handler sets a local flag and returns executing the job. In case of the budget polling approach, which is considered expensive, it is assumed that the available budget for a job is not available locally in a task. Assuming that polling of a local flag is cheaper, signalling in combination with local polling gains in performance compared to the pure budget polling approach.

The backside of deferred termination of a job is the increased reaction latency, which is the time that a preliminary termination takes (see Figure 10). The polling mechanism allows the programmer to trade-off reaction latency versus computational overhead. For example, when a task is polling for a termination request on a per block basis, the computational overhead is smaller than polling on a per pixel basis, however reaction latencies also suffer from the data dependent computation times.

3.7.4 MAPPING PRIORITY PROCESSING TASKS TO THREADS

When considering the priority processing application, an application can either consist of a single algorithm, which is allowed to consume the entire available budget, or can be composed of multiple independent algorithms, which are scheduled by an application specific policy. Operating systems typically use threads as a scheduling unit. When deploying the priority processing application on an operating system, the priority processing algorithms can be mapped in different ways onto threads:

1. **Map a job onto a thread:** Each job (a frame-computation) can be executed by starting a new thread. When the budget is depleted, the job is terminated and therefore also the thread is destroyed.
2. **Map a task onto a thread:** All jobs corresponding to a task are multiplexed to a single thread. A thread is reused by consecutive jobs of the same task, such that for each frame computation the cost of creating and destroying threads is saved. Instead of creating and destroying threads periodically, addition time-synchronization mechanisms are required. Upon preliminary termination of a job, all internals are reset and a new frame computation is started, discarding all additional work of the pending frame.

Mechanisms for allocating processor resources to threads are described in Section 3.5. Time synchronization is required for the priority processing algorithms by means of preliminary termination, requiring each algorithm to deliver its output to the screen. The following implementations can be considered:

1. Preliminary termination by *budget polling* and *write-through* data access. Polling mechanisms allow termination of algorithms at predefined moments in time, such that data consistency can be guaranteed by smartly choosing preliminary termination points.
2. Preliminary termination by *budget polling* and *write-behind* data access. On the one hand, this approach will cause additional overhead due to memory copy actions. On the other hand, when implementing the priority processing application on an embedded platform, it can be an advantage to store results locally By performing memory copy operations to copy larger chunks of data at once to the output buffer, instead of writing relative small amounts of data (obtained from small computational parts) immediately to the output buffer, performance can be gained.
3. Preliminary termination by *a-synchronous signalling* and *write-through* data access. Signalling can cause preliminary termination at any moment in time, such that data consistency must be ensured. Therefore, the most obvious approach is to use a write-behind approach.
4. Preliminary termination by *a-synchronous signalling* and *write-behind* data access. This approach will cause the same additional overhead due to memory copy actions as the previous option. A memory copy action must be performed atomically to ensure data consistency.

In the priority processing application, the models are implemented using a write-behind approach. Due to advantages for consistent output results and the available implementation, this is the only option we consider. Furthermore, the output buffer is considered as being non-local to the algorithms. Note that changing the write-behind approach to a write-through approach (or the other way around), requires application specific knowledge, since the algorithmic code need to be changed.

Another approach combining the signalling and cooperative termination mechanisms is to split a subtask in two threads:

1. A high priority task with preemptive scheduling, which executes the loop by calling the low priority thread to execute the body of the loop. This task receives the termination request and signals the other thread.
2. A low priority task with deferred preemption, which executes the loop body.

The high priority task is responsible for handling the termination request. The request is deferred at the granularity of the loop body, for example on block level. Although this is a valid approach, it gives rise to additional thread scheduling overhead (context-switches), which probably causes more overhead than a simple flag-polling approach. Therefore this approach is discarded in further investigations.

In Section 4 an example priority processing application will be investigated. This class of applications explicitly makes use of preliminary termination of jobs. It therefore is suitable for comparing different objectives of preliminary termination. Furthermore, we will investigate efficient mechanisms to support scheduling of independent, scalable algorithms.

4. EXPERIMENTAL SETUP AND IMPLEMENTATION

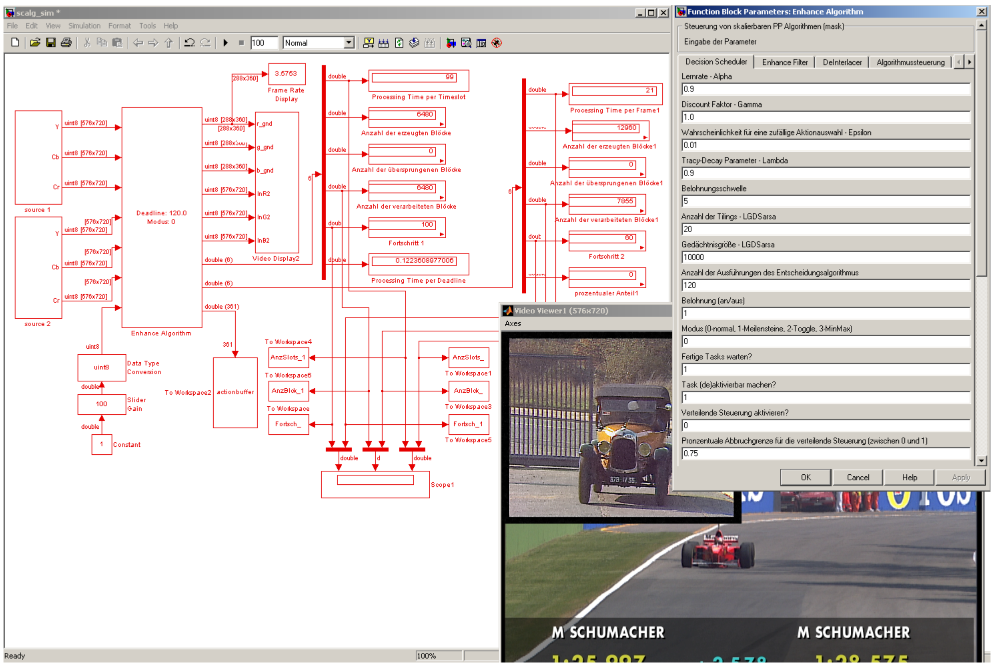
This section describes the simulation environment and discusses the model implementations of the priority processing applications. Thereafter, modifications to the provided models are discussed. Finally, the questions to be solved are summarized.

4.1 SIMULATION ENVIRONMENT

The priority processing application is prototyped in a Matlab/Simulink environment. The video processing algorithms are implemented using the C++ language. There are three different models available to take measurements:

1. **Scalable deinterlace with adaptable quality level**
   This model runs the priority processing algorithm for the deinterlacer algorithm until a user determined quality level (measured in percentage of processed blocks per frame) is reached. This model contains no worker threads and is suitable for measuring approximate WCETs of (parts of) algorithms.
2. **Scalable deinterlacer with adaptable deadline periods**
   This model runs the priority processing algorithm for the deinterlacer algorithm and applies a deadline constrained on the scalable part of the algorithm. This model creates and destroys a worker thread for every frame, which performs the scalable deinterlacer algorithms.
3. **Dual component application with decision scheduler**
   This model contains a decision scheduler and the deinterlacer algorithm competing for resources with an enhancement filter algorithm. Both algorithms work independent on different video streams (visualized picture-in-picture).

The Matlab/Simulink environment provides standard components to build the model, which enforces the application developer to implement application specific functions in a supported programming language (in our case implemented using C++). These functions define the relation between the input and the output pins of the pre-defined block components. When running the Matlab simulations, the environment ensures the right sequence of actions implied by the block model. A screenshot of the Matlab/Simulink environment for the dual-component application is shown in Figure 15.



Function Block Parameters: Enhance Algorithm

Steuerung von skalierbaren PP Algorithmen (mask)

Eingabe der Parameter

Decision Scheduler | Enhance Filter | DeInterlacer | Algorithmussteuerung

Lernrate - Alpha
0.9

Discount Faktor - Gamma
1.0

Wahrscheinlichkeit für eine zufällige Aktionauswahl - Epsilon
0.01

Tracy-Decay Parameter - Lambda
0.9

Belohnungsschwelle
5

Anzahl der Tilings - LGDSarsa
20

Gedächtnisgröße - LGDSarsa
10000

Anzahl der Ausführungen des Entscheidungsalgorithmus
120

Belohnung (an/aus)
1

Modus (0-normal, 1-Meilensteine, 2-Toggle, 3-MinMax)
0

Fertige Tasks warten?
1

Task (de)aktivierbar machen?
1

Verteilende Steuerung aktivieren?
0

Pronzentuale Abbruchgrenze für die verteilende Steuerung (zwischen 0 und 1)
0.75

OK Cancel Help Apply

• *Figure 15: The main window shows the block diagram, which defines sequence of the actions defined by a relation between inputs and outputs. The small dialog (right-top) shows an example of changeable model parameters. The video-window (right-bottom) shows a preview of the simulation for a dual-component priority processing application, visualized picture-in-picture.*

The input videos feeded to the models are collected from the Video Quality Experts Group (VQEG) [14]. This group provides standard video sequences, which are widely used and accepted in signal processing research.

4.2 OPERATING SYSTEM DEPENDENCIES

The models can run as well on Linux as on Microsoft Windows XP machines by adapting the thread implementation. Although, some minor differentiations have to taken into account.

4.2.1 MAPPING OF COMPONENTS

A *process* is a program in execution that defines a data space and has at least one associated thread. A *thread* is a unit of concurrency and a unit of scheduling. The priority processing application is mapped onto a process. Each scalable component is mapped onto a (worker-) thread. These threads are contained in a process, together with the run-time threads of Matlab. These threads, which run the scalable parts of the priority processing video algorithms, can be newly created or repeatedly restarted each period. The worker threads, which execute the scalable algorithms, are scheduled according to the local, application specific scheduling policy; for example by means of reinforcement learning or round robin scheduling.

The periodically activated decision scheduler shares a thread with all basic priority processing functions within the application. This setup ensures that the basic parts are executed first, before the scalable part is executed. The scalable functions of a single priority processing algorithm is mapped on a separate thread. In total, for a priority processing application composed from N priority processing algorithms, there are $N + 1$ threads implementing the application's functionality.

4.2.2 USING FIXED PRIORITY SCHEDULING

In order to implement a scheduling policy for the distribution of resources over competing priority processing algorithms, we need the availability of a fixed priority scheduler. As well Microsoft Windows as GNU/Linux provide a fixed priority scheduler for threads in the so called real-time class. These threads are scheduled with the highest priority and can possibly cause starvation of other threads in the system. However, we have to take into account, that these operating systems do not provide strict real-time guarantees and that other system processes running in the background can interfere with the priority processing application. In the simulation environment, the correct implementation of the task model causes some challenges to cope with:

1. In case of Linux based operating systems, it is required to have super-user access in order to have permission to change thread priorities. This means that the process containing the simulation threads (Matlab) must run with super-user rights. However, this conflicts with the policy of Matlab, which forbids Matlab to run as superuser. In order to run Matlab under Linux, first a license daemon must be started, which checks whether the user on that machine is a registered user. This daemon does not allow starting Matlab with superuser rights. The preferred platform to run simulations is the Windows platform. Therefore, we will focus in this document on the Windows variant. We will look to some simpler models under Linux, in order to test the preliminary termination by means of a-synchronous thread signalling.
2. In case of the Windows operating system, it is allowed to manipulate priority levels of threads, when a user has administrator rights. Windows assigns a thread a base priority and a dynamic priority. The base priority is indicated by the priority-class. In case that the priority class is set to the real-time class, the Windows scheduler does not manipulate the dynamic priorities, but keeps them static. This is exactly what we desire, however a priority-class can only be assigned to a process. This means that the complete Matlab process will run in real-time mode, including all its child threads which inherit the base priority. This means that during simulation, between 30 and 40 threads are running in the real-time class (which is on itself not a problem).

4.3 DIAGNOSTICS

The priority processing applications, as provided initially, are implemented to simulate the video algorithms and compare the output qualitative to other scalable video algorithm approaches. The lack of a fundamental real-time task model requires to review the implementation.

4.3.1 REVIEW THE TASK MODEL

When looking at the scalable priority processing algorithms, these algorithms can be modeled as periodic tasks. Every period, a new job is instantiated. There are two implementations to be considered:

1. In the initial version of the priority processing application, each period new threads are created and destroyed at the end of the period.
2. Instead of periodically creating-destroying threads, it is conceivable to reuse a thread, such that the overhead for creating and destroying is reduced. The newly introduced overhead consists of temporarily

activating and sleeping of the threads, as explained in Section 3.7.4.

The decision scheduler, which assigns the time-slots to the threads implementing the scalable algorithms, must be assigned the highest priority. The scalable algorithms must be assigned a lower priority. In the original implementation the scalable algorithms were assigned the highest priority within the normal Windows scheduler, which causes a semi-fair scheduling approach. The worker threads, which run the scalable algorithms, were assumed to work in the real-time class, using the Windows fixed priority scheduler. However, this was implemented in a wrong way, since only processes support priority classes after which the child-threads can be assigned priorities within the range of the priority class. The process containing the application was not put explicitly in the real-time scheduling class, such that its child threads can not be assigned real-time priorities.

4.3.2 REVIEW ON MULTI-PROCESSORS

In order to successfully simulate the Matlab/Simulink models, a multi-processor system is required. Next to the threads implementing the scalable video processing algorithms, a lot of threads are started by the Matlab environment. These Matlab threads include threads to extract the simulation results, such that concurrent thread execution is required.

In earlier research results, parts of the signal processing algorithms are boosted in performance by the use of OpenMP [3]. The OpenMP library allows software-pipelining of loops, by distributing the workload over the available processor elements. In order to obtain concurrency, OpenMP creates worker threads. In order to pipeline the loops, no control code (for example flag polling) is allowed. This means that the OpenMP library does contradict the priority processing property of allowing preliminary termination at any moment in time.

4.4 DYNAMIC RESOURCE ALLOCATION FOR PRIORITY PROCESSING

This section presents the different trade-offs for resource management mechanisms, as described in Section 3. In order to prototype an efficient resource management scheme for the priority processing application, we assume that the operating system provides a fixed priority scheduler. Dynamic resource allocation has much in common with reservation-based resource management [29], see Section 2.5. The distinguishing characteristics of our approach are the lack of admission control and the need for preliminary termination of priority processing algorithms, enforcing a synchronization between processing and time.

4.4.1 RESOURCE USERS

It is assumed that the priority processing application is granted a certain amount of processor cycles. These provided processor resources are used by the scalable algorithms and the decision scheduler, which functions as the control (see Section 3.2). In our simplified model, the resources can be entirely consumed by the scalable algorithms, without the need of sharing cycles with the control. The decision scheduler is mapped onto its own dedicated core, enabling the signal processing algorithms to use their own dedicated (single) core. Therefore, the availability of a multi-core machine is assumed.

It is assumed that the application has the entire processor at its disposal. In case of resource sharing among competing applications, virtualization of resources can provide guaranteed resources for each application. For example, the resource reservation schemes (as introduced in Section 2.5) assigns each application in the system a partition of the total resources. Resource partitioning introduces additional challenges for ensuring correct timing behavior within an application, since time is relative within the assigned budget instead of absolute as considered in this report. Virtual platforms are considered as a topic for future research and therefore the problem is discussed in more detail in Section 6.2.2.

4.4.2 BUDGET DEFINITION

The decision scheduler assigns resources in terms of fixed length time-slots, Δt_s, to the competing, independent priority processing components. A time-slot has a fixed length, of $\Delta t_s = 1\ ms$, in our simulation environment. Note that the choice for an appropriate length of a time-slot is platform dependent. The length of a time-slot is chosen based upon earlier research result [32]. It is assumed that during this time-slot, a component has the entire processor at its disposal.

Different lengths for time-slots, Δt_s, for the reinforcement learning policy are investigated by Schiemenz [32]. Smaller time-slots implies more control overhead caused by the scheduler, whereas bigger time-slots can give unbalanced resource divisions. Trading-off the length of a time-slot is policy dependent and consequently also application dependent. This report concentrates on fixed-length of time-slots with a length of $\Delta t_s = 1\ ms$.

It is assumed that the scalable algorithms have the entire processor at their disposal. Each period, of length T, contains N time-slots with a fixed size Δt_s, i.e.

$$T = N \times \Delta t_s \tag{7}$$

A *budget* is defined as the amount of processor resources available within a period.

4.4.3 RESOURCE ALLOCATION

For the control of independent, scalable priority processing components, a reinforcement learning policy is developed, as described in Section 2.3. In order to verify simulation results, a more deterministic policy by means of Round Robin is used. The Round Robin policy allocates time-slots in an alternating way over the competing algorithms, independent of their progress. The reinforcement learning policy makes use of feedback values from the algorithms, which indicate their reached progress. The progress values of the algorithms are used to determine which algorithm is allowed to consume the next time-slot. Contrary to the round-robin policy, the division of the time-slots by the reinforcement learning is not known upfront. The mechanisms to support the scheduling policy are independent of the policy it self. It is investigated to see what the influence is of the underlying mechanisms on the efficiency of the policy, for example: the possibility to consume gain-time.

In order to dynamically allocate processor resources, a task is assigned to the processor by implementing the mechanisms as described in Section 3.5:

1. suspending and resuming of the task;
2. manipulation of task priorities.

The latter option allows the consumption of gain-time, as explained in Section 3.3.

4.4.4 PRELIMINARY TERMINATION

Scalable video processing algorithms must be preliminary terminated in a controlled way and perform a roll forward to the next frame, when their budget is depleted. Preliminary termination can be done by means of:

1. cooperative termination by means of budget polling on regular intervals. At a predefined granularity, typically block- or pixel-based, a flag indicating budget depletion is checked.
2. a-synchronous signalling of tasks by the decision scheduler control component. Upon arrival of a signal, which indicates depletion of the budget, a signal handler is executed to terminate the current job which processes a frame.

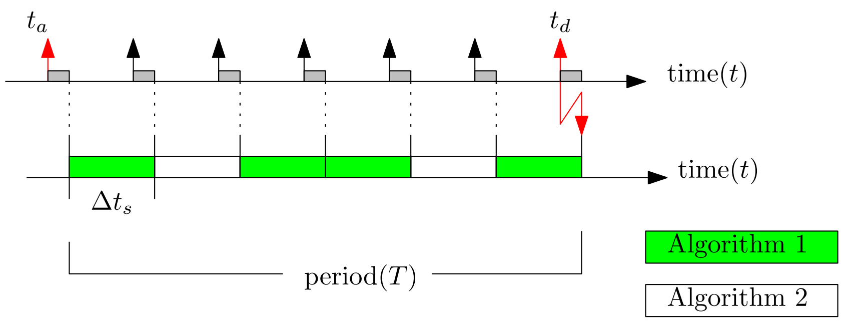
The decision scheduler has to trigger that the deadline is expired, but also has to ensure that upon exipration of the deadline all tasks are assigned a fraction of additional processor time to handle the request. For cooperative scheduling, this means that a computation can be finished. For signalling approach, the task has to process the incoming signal.

4.4.5 MONITORING

The allocation of the time-slots are subject of change during the period, hence the name dynamic resource allocation. Dynamic allocation of time-slots requires that activation of the decision scheduler must be ensured periodically, see Figure 16, which can be achieved by each of the following implementations:

1. putting the decision scheduler in a highest priority task which is periodically activated, e.g. after each time-slot;
2. activating the decision scheduler by means of a software-interrupt.

The decision scheduler is put on a separate host processor. In an embedded platform it is conceivable that the host processor assigns time-slots of a streaming processor (for example a DSP or VLIW) to the scalable algorithms. Similarly, both time-lines shown in Figure 16 can be mapped on a separate core. The advantage of mapping both the decision scheduler and the video processing algorithms on a separate processor is that the signal processing pipeline is not unnecessarily flushed every time-slot and that a general purpose processor is more suitable to run the scheduler than a streaming processor. This mapping of the decision scheduler to a



• *Figure 16: A period T, defined by time interval* $[t_a; t_d]$*, is divided in time-slots. Prior to a time-slot the decision scheduler decides (upper time-line), which algorithm to assign the time-slot (lower time-line). In this example two competing algorithms are shown, which both require preliminary upon expiration of the deadline, e.g.: at the time* t_d*.*

general purpose host processor fits in current hardware development trends, which move towards heterogeneous multiprocessor on chip designs, as explained in Section 2.4.1.

The decision scheduler implements a reinforcement learning policy, which monitors the priority processing algorithms. Assignment of time-slots is based upon reached progress values relative to the consumed amount of time-slots. The progress of an algorithm is accounted by the algorithm itself and can be queried by the decision scheduler, see Section 3.4. Progress is defined in terms of number of processed blocks by the priority processing algorithm. Processing of blocks means that the output buffer is changed, e.g.: there is a visual change in the output observable. This means that content analysis functions, which are part of the scalable priority processing algorithms and consume time, do not deliver any additional progress. For example: the scalable priority processing algorithm for deinterlacing video content consists of two stages, with in-between a content analysis filter function. This filter function consumes several time-slots without changing anything in the output buffer and therefore the progress value does not increase during the execution of this filter function. Note that the reinforcement learning algorithm is aware of these algorithm dependent characteristics, but this might give rise to additional complexity when comparing performance differences for different mechanism implementations.

The progress values are relative to the amount of time that an algorithm spent to reach that progress. Accounting mechanisms keep track of the consumed time. The amount of time consumed by a scalable priority processing algorithm is measured in terms of time-slots allocated to that algorithm. We assume fixed size time-slots, $\Delta t_s = 1\ ms$ as explained in Section 4.4.2.

After each time-slot the decision scheduler makes a decision to which algorithm the next time-slot is assigned. A time-slot is accounted to an algorithm upon assignment by the scheduler, such that gain-time is accounted to a gain-time provider rather than its consumer. The monitoring mechanism, providing the priority processing time-synchronization, provides important information to the decision scheduler. The decision scheduler assigns time-slots based on the progress values of the algorithms and the amount of used resources to arrive at that progress value, as discussed in Section 3.4.

4.4.6 APPLYING SUBTASK-STRUCTURE MODIFICATIONS

Preliminary termination of tasks, indicated in Figure 7 as numbered functions, is currently implemented by polling on regular intervals whether thethe deadline is expired. The pseudocode for these scalable functions is shown in Source Code 3, in which the *continue*() function implements the budget polling mechanism. The polling mechanism is considered time consuming, and therefore we want to get rid of this polling function. Note that this is similar to the model provided in Section 3.7.1 and will be compared with the signalling approach shown in Section 3.7.1, Source Code 2.

```
void old_function_i()
{
  initialSetup();
  while(continue() && !end_of_frame())
  {
    process_next_block();
  }
}
```

• *Source Code 3: Structure of a scalable function in a priority processing algorithm. This function includes a polling function continue() which checks the remaining budget at regular intervals.*

The *process_next_block*() function on its turn can also check on a per line or per pixel basis whether the budget has not expired by checking the *continue*() flag. This gives the opportunity to trade-off computational overhead and reaction latencies.

Finetuning Granularity of Termination
A coarse grained polling variant is relative cheap in terms of computational overhead, but suffers from deviated reaction latencies due to data dependent computation times. On the other hand, the signalling approach incorporates operating system calls, which are relatively expensive compared to setting a polling flag. Signalling is independent of the granularity of termination and is expected to be able to compete at least with the block polling variants in terms of reaction latencies.

Data Transfer Characterization
The priority processing application uses the write-behind approach, see Section 3.7.2, in its current implementation. Therefore it is most straight forward to compare the following models: When considering the priority processing application, the following implementations for preliminary termination can be compared amongst each other:

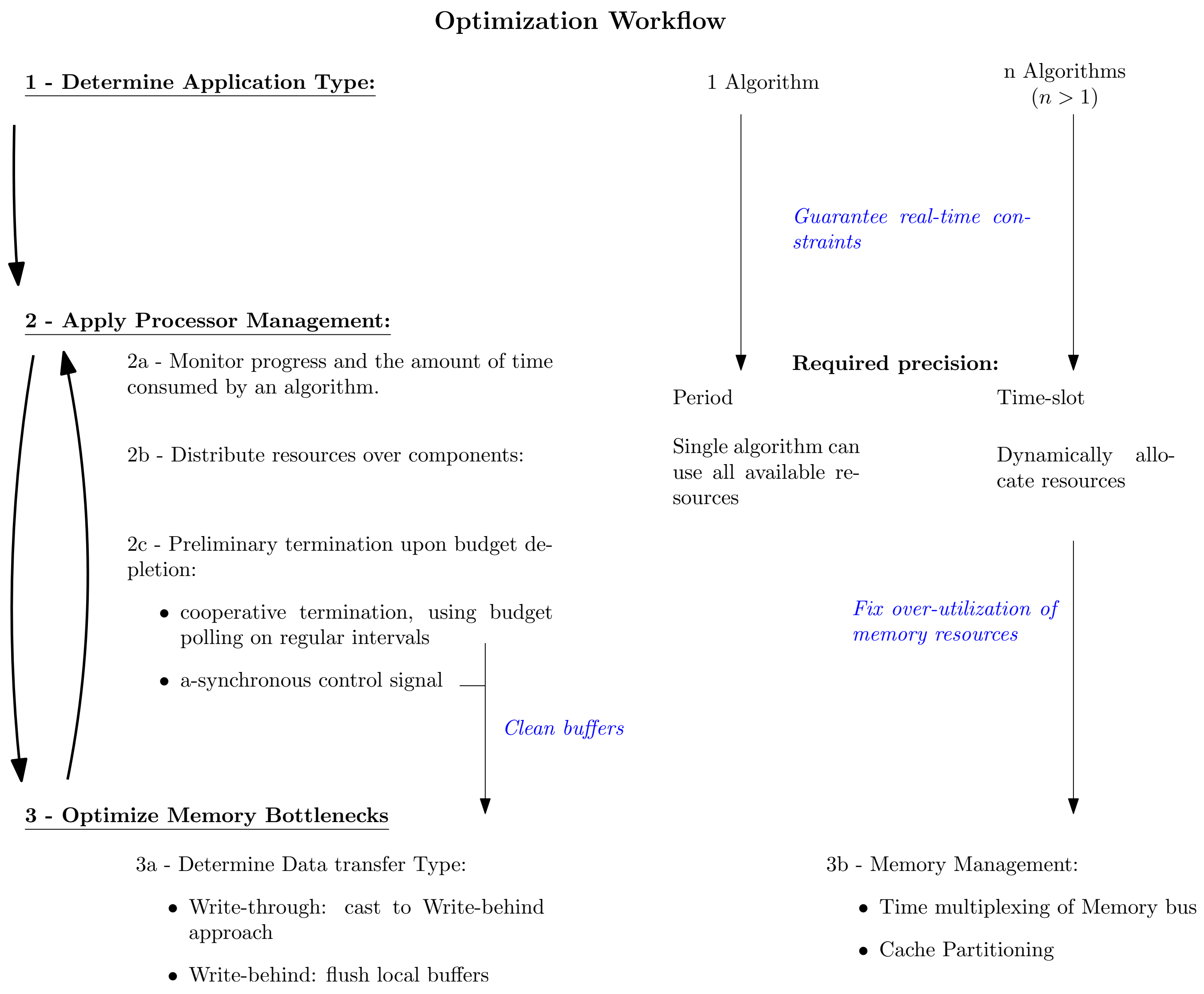
1. *budget polling* and *with write-behind* data access.
2. *a-synchronous signalling* and *write-behind* data access.

Note that the memory copying actions corresponding to write-behind memory handling approach consume time. In a real-time environment the time for writing the data to the output has to be taken into account and decremented from the budget assigned to the algorithms. In our simulation environment this issue is neglected, since the measurements exclude the time consumption of memory copy actions.

4.4.7 SUMMARIZING THE OPTIMIZATION WORKFLOW

The optimization workflow for the control of priority processing applications, as described in this report, is sketched in Figure 17. First, we determine whether the priority processing application is composed from one, or from more competing, independent components. A single component application can simply assign its periodic budget to the single component, whereas a multi-component application requires dynamic resource allocation of the time-slots within a periodic budget. Upon depletion of the periodic budget, preliminary termination requires resetting of buffers. Dependent on whether a write-behind approach is used, copying of the buffers is also required.

Mapping multiple components on a single processing element can cause overload of the memory bus and cache polluting. Technical solutions to solve these issues can change the parameters of the algorithms, e.g. computation time, which means that similar progress values can be reached within a smaller period. After optimizing memory resource management (for example by reservation-based mechanisms as explained in Section 2.5.2), it is required to perform a feedback loop, re-analyzing the processor allocation scheme. This optimization step is discarded during this project due to the lack of control over the hardware memory controller.


Optimization Workflow
1 - Determine Application Type:
1 Algorithm
n Algorithms ($n > 1$)
Guarantee real-time constraints
2 - Apply Processor Management:
2a - Monitor progress and the amount of time consumed by an algorithm.
Required precision:
Period
Time-slot
2b - Distribute resources over components:
Single algorithm can use all available resources
Dynamically allocate resources
2c - Preliminary termination upon budget depletion:
• cooperative termination, using budget polling on regular intervals
• a-synchronous control signal
Fix over-utilization of memory resources
Clean buffers
3 - Optimize Memory Bottlenecks
3a - Determine Data transfer Type:
• Write-through: cast to Write-behind approach
• Write-behind: flush local buffers
3b - Memory Management:
• Time multiplexing of Memory bus
• Cache Partitioning

• *Figure 17: Summary of the optimization workflow for the control of priority processing applications.*

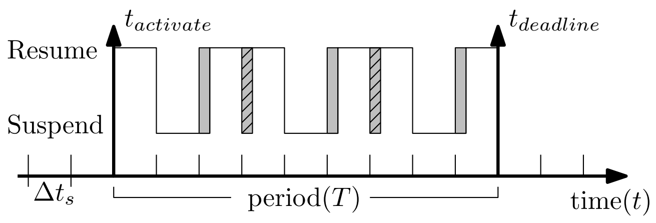
4.5 MODEL IMPLEMENTATION ON MICROSOFT WINDOWS XP

This section describes issues and justifications for the implementations of the dynamic resource allocation mechanisms within the priority processing applications, using the Microsoft Windows XP platform.

4.5.1 RESOURCE ALLOCATION

The scheduling policy, as initially implemented by Berger [6], makes use of the Microsoft Windows APIs [27] to suspend and resume threads. As an alternative, priority manipulation can be used to schedule the threads by making use of the Windows fixed priority scheduler. Priority manipulation is provided by the Windows API [27], but prohibited for users without administrator rights within Windows XP. Further more, the threads which run the scalable algorithms are restricted to a single cpu, by setting the thread affinity mask using the Windows API. This prevents the use of multiple cores, such that uncontrolled background processing is prevented.

Next to preventing uncontrolled background processing, it is required to prevent other applications within Windows to interfere with the priority processing application. Since the priority processing application is assigned a real-time priority, which is a high priority compared to normal applications running on Windows, the interference of third party applications should be very limited. The only threads that are allowed to preempt the priority processing threads are high priority system threads and high priority (critical) system services. It is assumed that the overhead caused by other real-time threads within the Windows system is negligible.



• *Figure 18: Grey areas show the time lost due to context-switching upon resuming an thread. This overhead can be reduced, when multiple consecutive slots are assigned to a thread (i.e. the shaded-grey areas can be removed). At the end of the period, $t_{deadline}$, all pending algorithms must be preliminary terminated.*

Additionally, the initial implementation causes unnecessary context-switching overhead, due to explicit suspension of all threads after each time-slot. When multiple, consecutive time-slots are assigned to the same algorithm, advantage can be taken from reduced context-switching. By only suspending an algorithm in favor of resuming another algorithm, context-switches can be reduced as shown in Figure 18.

4.5.2 MONITORING

We deviate from the task model described in Section 3.3, by assuming that there is enough time to process the basic parts of the algorithms. Only the scalable parts are constrained by timing bounds. In case of a single component, there is no scheduling involved during the period of the scalable part. This means that we can simply put the scheduling thread in sleep mode during the time that the single worker thread will perform the scalable algorithm.

In case of multiple competing components, the decision scheduler has to be activated after every time-slot of 1 ms. When putting a thread into sleep mode for such a small time, the system overhead is relative high. The same holds for the use of Windows timer interrupts at such relative high rates. Therefore, a busy waiting loop is performed using high performance counters, similar to mechanisms used to perform benchmarking as described in Appendix C. The implication of the busy-waiting implementation is that a multi-processor system is required, since the decision scheduler requires a dedicated processor.

4.5.3 PRELIMINARY TERMINATION

After each period, all scalable priority processing algorithms must be preliminary terminated. This requires that all competing algorithms are allowed to execute for a short time, to execute the termination request. For cooperative preliminary termination by means of a block-based polling variant, this means that the algorithm will complete its pending block before it terminates.

Polling is platform independent, since polling enforces time synchronization in the code and therefore predictable application behavior. However, the cost for polling a flag might be considerably higher on a streaming processor than on a general purpose processor. As an alternative, an a-synchronous signal can be send by the control to the algorithms. A-synchronous signalling of tasks requires operating system support. Since a signal can arrive at any moment in time, the state of the task must be saved before executing the signal handler.

The API of the Microsoft Windows operating system defines several options for a-synchronous signalling. However, none of these options are fully a-synchronous from the perspective of the programmer. Most noticeable is the *A-synchronous Procedure Call (APC)*, which is the Windows variant of POSIX signals. APCs are stored in a queue and executed within a specific thread. However, this thread must be in an *alertable* state, which means that the programmer has to call a sleep or a wait function. An APC can be used to implement callbacks, but is not so general as Linux signals. Therefore the APC is not suitable for preliminary termination of jobs.

Another option that is provided by the Windows API is the use of *events*. Windows events can be triggered and executed a-synchronously within the context of a process. This means that the event is executed within the Matlab context and consequently there is no control over the thread running the scalable video algorithm. Therefore the event mechanism provided by Windows is also not suitable to preliminary terminate jobs.

As an alternative, a thread can be explicitly killed from outside. This enforces the job to stop its activities, but has the following disadvantages:

1. for each frame a new worker thread must be created for each scalable algorithm. The thread-descriptor, which registers the thread within the process, is not cleaned up when the thread is killed from outside. Since a process can only have a limited amount of thread descriptors, the simulation crashes after a while, because of exceeding its thread limitation.
2. For the period that the simulation is able to produce video output, the destruction of a thread is a very costly operation. The latency for killing a thread from outside is very high and causes visual video artifacts.

Other language constructs provided by the Windows API are built on top of the APC mechanism and event-system, for example: a timer triggers an event when it fires. Consequently, there is no option to preliminary terminate jobs with an external a-synchronous signal on the Windows platform. Therefore, the only option to preliminary terminate jobs in a predictable and stable way on the Windows platform is by means of cooperative methods.

4.6 PORTING THE MODEL TO GNU/LINUX

Due to issues with the Matlab/Simulink environment in combination with superuser access (see Section 4.2.2), which is required for changing thread priorities, we only consider models with a single component. This means that no resource allocation mechanism is required, such that it is enough to implement mechanisms for accounting the amount of time within a period and preliminary terminate the single priority processing algorithms upon depletion of the deadline.

Another side-effect of porting the code to the Linux platform is the extensive code-cleanup. Due to the fact that the C++ compiler from GNU Compiler Collection (GCC) is more restrictive than the Microsoft Visual C++ compiler, a lot of minor changes and redundancies have been fixed. Examples include: casting of variables; removal of duplicated code and fixing a possible out-of-bounds error in the extensively used quick-sort function.

4.6.1 MONITORING

We assume a similar timing model as in the Windows variant, see Section 4.5.2. The thread containing the decision scheduler is put into sleep mode for the period of the scalable part. The case of multiple components is not considered for the Linux models.

4.6.2 PRELIMINARY TERMINATION

Due to missing support for preliminary termination of jobs in the Windows operating system, see Section 4.5.3, the Linux signals are considered. Linux provides the possibility to a-synchronously signal threads and attach a signal handler to the signal. The use of POSIX threads and signals is described in more detail in Appendix D.

Although, the signalling approach is less reliable than polling-based methods, some measurement data is gathered to compare both implementations (signalling and polling) of the preliminary termination mechanism.

4.7 HYPOTHESES

Experiments, which compare different implementations for the mechanisms monitoring; preliminary termination and resource allocation, are expected to fulfil the expectations summarized in this section.

4.7.1 PRELIMINARY TERMINATION

The first goal is to obtain statistics about the system overhead caused by different methods of job termination control. Handling a job termination request is expected to be a trade-off between latency and the block size.

1. Polling mechanisms as a mechanism to allow preliminary termination, will cause data-dependent termination latencies. Termination latencies when polling on the granularity of a block-computation cause more deviated values, than polling on the granularity of a pixel-computation.
2. Termination latencies caused by the implementation of a-synchronous signalling should be data independent, but at the cost of operating system overhead. This overhead includes switching between user and kernel mode and a polling mechanism at the kernel level in order to check whether a signal is delivered for a specific thread. The latency of polling (by the kernel) is therefore dependent on the granularity on which the operating system handles the signals.
3. The mapping of tasks to threads by reusing threads over multiple jobs saves periodic creation and destruction of threads. The overhead caused by destructing threads is dependent on the thread implementation provided by the operating system. In worst case, the overhead for destructing threads can dominate the required latency for synchronizing the signal processing flow with time.

4.7.2 RESOURCE ALLOCATION

The second goal is to reduce control overhead by considering implementations of mechanisms to efficiently allocate resources.

1. Only deallocate the resources for a threads reduces the amount of context switches, especially when multiple consecutive slots are allocated for the same algorithm. In this case, the algorithm can run as if it obtained one big time-slot. Reduction of context-switching overhead enables the algorithms to process more video-content.
2. Restricting the algorithms explicitly to a single core will increase the context switching and system overhead. Multi-cores solutions are possible, but disallow the comparison of different scheduling policies. The implementation using suspending-resuming of threads explicitly cause context-switching, since a thread which is suspended gives up the processor resources. Therefore it is expected that this change will not cause any observable differences.

4.8 RESEARCH QUESTIONS

Experiments, which compare different implementations for the mechanisms monitoring; preliminary termination and resource allocation, are expected to answer the questions described in this section.

4.8.1 PRELIMINARY TERMINATION

Experiments for analyzing preliminary termination are performed using the priority processing applications for deinterlacing video content, as described in Section 2.2. By performing simulations with the deinterlacer priority processing application, the following questions are solved:

1. *What is the computational overhead caused by polling a termination flag?*
   In order to solve this question, we run the deinterlacing application and measure the time to deinterlace a complete frame at the highest quality. The processing times for the algorithm including budget polling and a clean version are compared.
2. *What is the reaction latency upon a termination request for flag polling?*
   The time is measured from the moment that the budget is expired, setting the expiration flag, until the worker thread gives the control back to the decision process.
3. *What is the reaction latency upon a termination request for signalling a job?*
   The time is measured from the moment that the budget is expired, signalling the worker thread, until the worker thread gives the control back to the decision process.

4.8.2 RESOURCE ALLOCATION

Experiments for analyzing resource allocation, to support the scheduling policy, are performed using the dual-component priority processing applications, as described in Section 2.2. By performing simulations with the priority processing application, the following questions are solved:

1. *Is an performance difference observable between implementing the scheduling policy using priority manipulation or suspend-resume?*
   When disallowing gain-time consumption, the performance between both implementations should be approximately the same. If a context-switch occurs, then a context-switch requires an amount of processing time which is independent of the scheduling policy. Since the reinforcement learning policy reduces the amount of context-switches compared to the round robin policy, it is investigated how much gain is provided by this advantage.
2. *How much benefit can be obtained from gain-time consumption?*
   Priority processing applications are suitable for using gain-time. The use of gain-time allows background algorithms to use the time which is not needed by the algorithm which is assigned the time-slot. The amount of available gain-time is expected to be relative low. Maximal once per frame computation, a fraction of a time-slot appears as gain-time, assuming that algorithms which completed a frame are not assigned a time-slot.

4.8.3 MONITORING

Mechanisms for preliminary termination and resource allocation rely on the availability of accounting information. Therefore the following questions must be investigated:

1. *How can we enforce time-slots of 1 ms as precise as possible?*
   After each time-slot the decision scheduler is called. This requires high-resolution timers, either to periodically activate the highest priority task containing the decision scheduler, or to fire a software interrupt which executes the decision scheduler in its handler.
2. *Is there an observable performance difference in activating the decision scheduler by a software interrupt or a periodic task?*
   A software interrupt always runs at the highest priority such that it can not be interrupted. The same behavior can be achieved by putting the decision scheduler in the highest priority task and temporarily deactivating software interrupts while the decision scheduler is active. In our simulation environment these solutions can not be tested, due to the lack of high resolution timers within Windows and Linux.

5. MEASUREMENTS AND RESULTS

In this section experimental measurements and results are presented, which clarify the questions stated in Section 4.8. In order to perform measurements, video sequences are used from the Video Quality Experts Group, VQEG [14]. Appendix E provides details about the platform on which the simulations are performed.

5.1 MEASUREMENT ACCURACY

First results regarding the measurements of small time intervals in the range of micro-seconds or lower, suffer from peaks in measurement data. An example of a figure, which still shows inaccuracies is Figure 19. Figure 19 shows the time required to deinterlace a video frame. The video sequences is repeated two times, which explains the periodic behavior of the graphs. However, when comparing the scene measurements with relative high computation times (i.e.: approximately frame 250 till 400 and frame 650 till 800), occasionally some peaks can be identified which only occur in one of both instances of the same measurement. In order to cope with these inaccuracies, benchmarking methods are applied as described in Appendix C. The results are essentially corrected by means of using a platform dependent library for benchmarking and repeating the simulations multiple times.

New measurements, with more reliable results, are made by:

1. running the same experiment (with the same video content) multiple times, e.g. three times and if required (lot of errors occur in the results) five times.
2. applying a median filter over the gathered results from the different runs.

This approach will work, based upon the assumption that multiple repetitions of the same experiment will obtain almost identical results. When one of the experiments gives a wrong result, it will be discarded. The value in the middle will be chosen as the correct value. Based upon the earlier assumption, this middle value is representative for a correct measurement.

Furthermore, it is observed that deinterlacer algorithm originally suffered from a big variance in termination latency at the range of completion of 50% of the work. The deinterlacer consists of two scalable parts, with in-between a content analysis filter-function which consumes typically 3∼4 ms. This filter lacked polling the termination flag on regular intervals. This means that trying to preliminary terminate a frame while it is performing the filter gives rise to extraordinary high latencies. The filter is now extended with polling statements, correcting the behavior of the application allowing for preliminary termination. This mistake in the implementation also explains additional short-comings apart from the computational overhead caused by polling variants. Polling statements require careful placement within the code by the application programmer, which can distract from the actual program flow.

5.2 PERFORMANCE COMPARISON FOR PRELIMINARY TERMINATION MECHANISMS

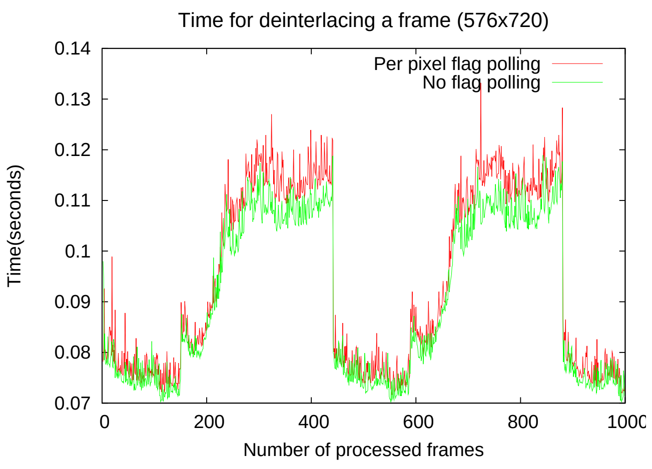
The basic principle of priority processing is the preliminary termination of a job, which processes a frame, upon depletion of the budget. In this section, polling and external signalling mechanisms are compared to support the preliminary, controlled termination of jobs. The most important measurement criteria for termination of jobs are:

1. the latency measured from the time that the polling flag is toggled until the time that the algorithm gives back control to the control component;
2. the computational overhead caused by regularly checking whether the flag has been set.

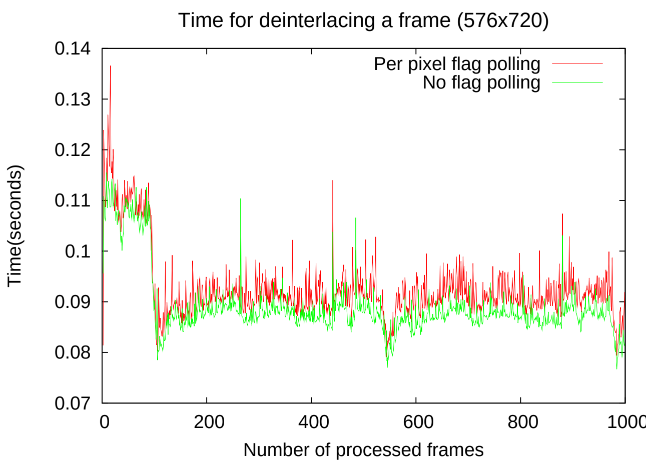
The algorithms support preliminary termination by means of a budget polling mechanism, as shown in Source-Code 3. The termination latency corresponding to polling based mechanisms, is dependent on the granularity of polling. In order to predict typical values for the termination latency of polling based mechanisms, a reasonable estimate of worst-case computation times of computational extensive code fragments is desired.

5.2.1 WCET FOR VIDEOFRAMES AND PIXELBLOCKS

Obtaining the real worst-case execution times (WCETs) of video algorithms is typically not feasible due to the high load fluctuations and data-dependent computation times. Although, by testing different, representative video sequences semi-worst-case computation times can be extracted for the processing of a complete video frame. Therefore, we can simply run the deinterlacer at full quality level and measure the total time needed to compute single frames in a diversity of video sequences, see Figure 19.



(a) WCETs for VQEG source 6, without noise adaption



(b) WCETs for VQEG source 10, with noise adaption

• *Figure 19: Computation times for deinterlacing video frames at the highest quality level, measured with and without polling overhead.*

The computation times for the two video sequences shown in Figure 19 show three interesting properties:

1. In the VQEG 6 sequence, see Figure 19a, there is a big difference in computation time between scenes. The slope of the computation times during the transition between scenes is very steep.
2. The deinterlacer makes use of a noise adaption function. For the first frames of the video sequence, the computation times are considerably higher than the following frames. In the beginning, the noise level measurement mechanism assumes that the video content contains no noise and therefore all blocks are processed. Thereafter, the blocks which have a high noise level are skipped. This can drastically reduce the computation time of a frame as shown in Figure 19b. After approximately 50 frames, the noise adaption converges to a stable approximation of the noise level in the video content. The algorithm is based upon the idea that the noise level within a video sequence is relative constant.
3. The computational overhead of polling, as a mechanism to allow preliminary termination, seems to be very limited, which gives rise to further investigations.

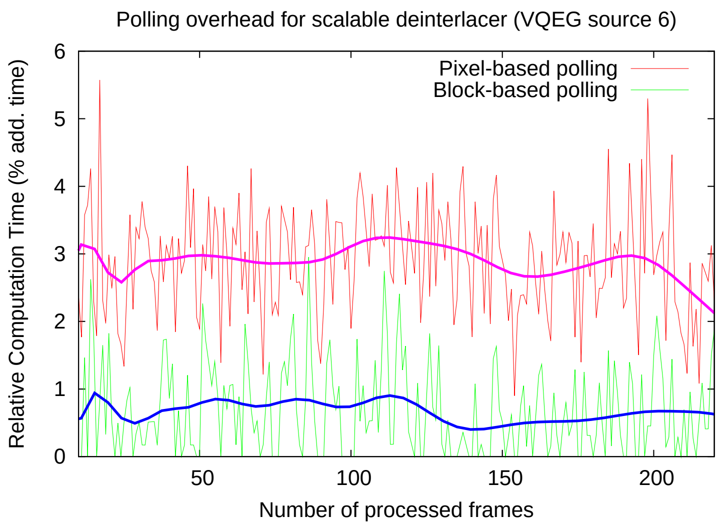
Implementations of the scalable algorithms contain flag polling at specific places in code to allow preliminary termination. By comparing measurements with and without flag polling, values for polling overhead in terms of computational time can be obtained. It is expected to see a trade-off in terms of latency versus computational overhead, imposed by the granularity on which the polling mechanism is applied.

In the first experiment, the following alternatives are compared:

1. **Per pixel polling:** At the beginning of each pixel computation, the termination flag is polled in order to check whether the budget is sufficient to continue processing.
2. **Per block polling:** The video screen is divided in blocks of $8x8$ pixels. Most video algorithms work at the granularity of a pixelblock. At the beginning of each block computation, the termination flag is polled in order to check whether the budget is sufficient to continue processing.
3. **No polling:** The whole video frame is processed without any flag polling, which means that preliminary termination of jobs is not permitted.

Figure 20 shows the relative time needed to deinterlace a frame, normalized against the implementation with no polling. It is shown that the computational overhead is relative low on general purpose machines, around 3% for pixel-based polling and 1% for block-based polling, whereas we can expect a big difference on embedded streaming processors.

Computation times for pixelblocks are content dependent, which causes the high fluctuations in frame computation times. When measuring these block computation times for video sequences, typical plots as shown in Figure 21a are obtained. Figure 21a shows the computation times for all blocks per frame in a sample video sequence. Per processed frame, block computations show a decreasing trend. The blocks requiring most computation times, are the most important blocks, see Figure 21b. These blocks are processed first by the priority processing algorithms.



• *Figure 20: Relative computation times to complete the scalable parts of the deinterlacer application for VQEG source 6, for a scene with relative high overhead. Flag polling on the granularity of a per pixel and per block computation is compared with an implementation without polling.*

When terminating a thread via flag-based polling, the reaction latency is dependent on the granularity on which the algorithm polls the termination flag. Polling more regularly means more computational overhead, but a reduced latency and vice versa. The video algorithms work on a per block basis, normally of 8x8 pixels. When polling the termination flag on a per block basis, the WCET for a pixelblock indicates in which range we can expect reaction latencies for terminating a pending frame.

To obtain results which are less influenced by the data-dependent character of the algorithms, different video content is tested. Per frame, we are interested in the block with the highest computation time. In Figure 22, the maximum WCET over all tested video sequences for pixelblocks are plotted over processed frames.

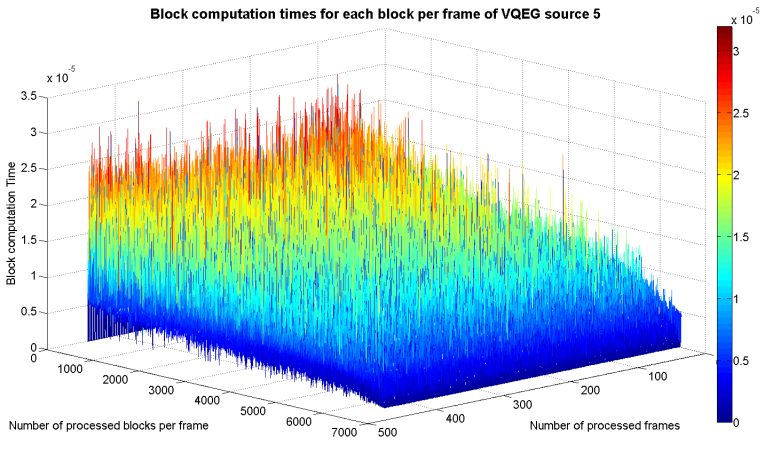
To get a good estimate of WCETs for block-computations, as plotted in Figure 22, the following experiment is executed:

1. Run the deinterlacer over a video sequence and gather all block-computations times. Per frame of 576×720 pixels there are 6480 blocks of 8×8 pixels and a video sequence consists of 440 frames. The measurement is repeated three times to filter measurement inaccuracies (see Section 5.1).
2. Now we define a function, $f(x)$, where x iterates over the frames of the video. For each frame, x, we have a set of block-computation times, named $S_{blt}(x)$. To extract the block with the highest computation time within a frame, we define
$$f(x) = max(S_{blt}(x)) \tag{8}$$
3. The above two steps are repeated for different video sources: VQEG source 3, 5, 6, 9 and source 10. For every video source, a function $f_i(x)$ is defined, where the suffix $i \in \{3, 5, 6, 9, 10\}$ defines the video sequnce.
4. The WCET estimate as plotted in Figure 22 is defined by a function, $W(x)$, over the frames which are again indexed with x. The function definition for the plot in Figure 22 is shown in Equation 9 and takes the maximum computation time measured over all sequences.

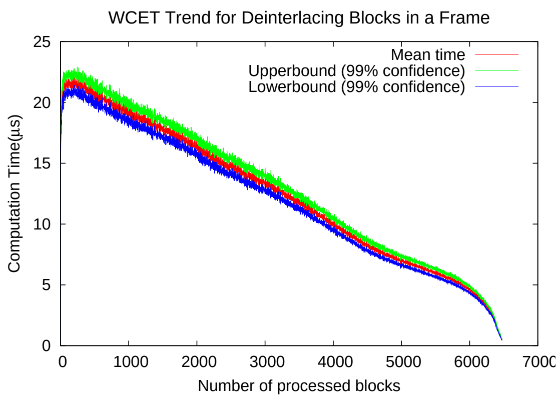
$$W(x) = (max\ i : i \in \{3, 5, 6, 9, 10\} : f_i(x)) \tag{9}$$

Measurement results for different video sequences show a WCET, $W(x)$ of approximately $25\ \mu s \leq W(x) \leq 35\ \mu s$. This WCET is measured for the last stage of the deinterlacer, which is the most computational part of the scalable deinterlacer application.

We assume a worst-case estimate of 35 μs to process a block of 8x8 pixels, based upon the results in Figure 22. When polling a termination flag at the granularity of a block-computation, it is expected to have a termination latency of approximately the same length, e.g. 35 μs. When decreasing the granularity of polling to the level of a pixel-computation, it is expected to measure latencies of approximately: $\frac{35\ \mu s/block}{64\ pixels/block} = 547\ ns/pixel$.

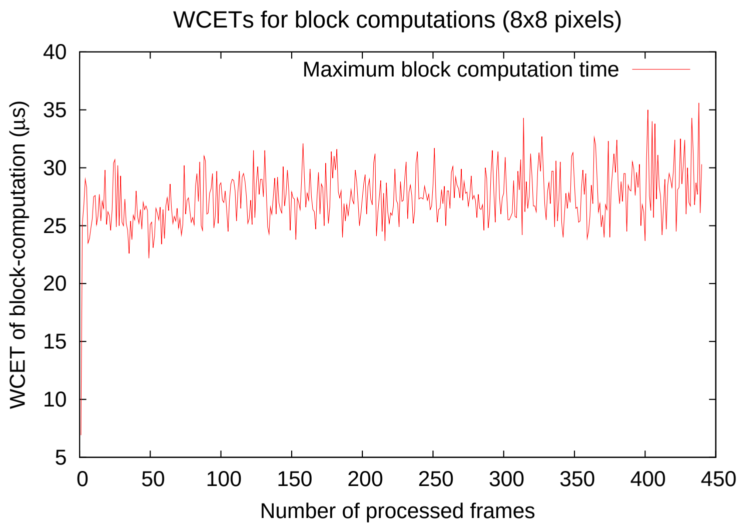


(a) Block computation times for single video sequence.



(b) Average block computation times over frames for corresponding block-indices.

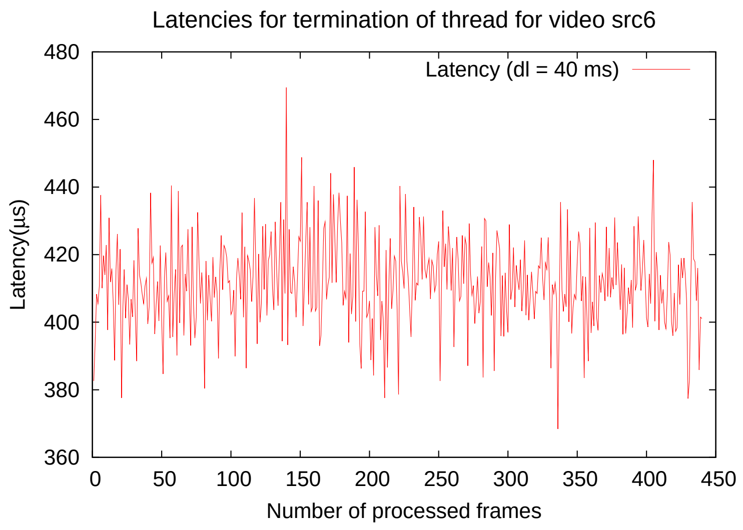
• *Figure 21: Computation times for pixelblocks of 8x8 pixels for the motion compensated scalable deinterlacer part. The computation times are measured for VQEG source 5, for all blocks per frame.*



• *Figure 22: Worst-case execution estimation for block computations for the motion-compensated deinterlacer part. The maximum values are extracted from simulations over different VQEG sources: source 3, source 5, source 6, source 9 and source 10.*

5.2.2 Thread Termination Latencies

The computation times for pixelblocks gives an indication for the expected reaction latencies. In this case we are especially interested in preliminary termination, since latencies are negligible in case a frame is completed. Original implementations of the models contain periodic creation and destruction of threads, as explained in Section 4.3.1. Periodic creation and destruction of threads includes system overhead for synchronizing and terminating of the worker threads.



• *Figure 23: Termination latencies for Windows threads show values around 400 μs. This graph shows experimental results for the deinterlacer application with VQEG src6, measured for deadlines of 40 ms. Further tests have shown that this latency is independent from deadline or thread priority.*

First the block-based polling variant is measured. Termination latencies, which include the termination of a thread, show values around 400 μs. This is a factor 10 compared to the block computation time, and therefore dominate the block-computation execution times. The measurements are repeated on the Windows platform using an example application, see Source Code 4, which includes only polling of a termination flag. Latency measurements are done by making use of methods described in Appendix C.

```
// Number of samples (#thread terminations):
const int N = 1000;
// Polling flag for preliminary termination of thread:
volatile bool running;

// Thread function which continously polls the termination flag:
unsigned __stdcall ThreadFunc(void* param) {
    while (running);
    return 0;
}

int _tmain()
{
    // Thread identifier:
    unsigned int uiThreadEnhID;

    // Benchmark thread termination N times:
    for (int i = 0; i < N; ++i) {
        // Set flag:
        running = true;
        // Create thread in suspended mode, with ThreadFunc as main loop:
        HANDLE hThreadEnh = (HANDLE)_beginthreadex(
          NULL, 0, ThreadFunc, NULL,
          CREATE_SUSPENDED, &uiThreadEnhID );

        // resume the thread:
        ResumeThread( hThreadEnh );
        // Wait 1s before termination:
        Sleep(1000);

        // Start benchmarking counter:
        clock_counter.start();
        // Toggle termination flag:
        running = false;
        // Wait for termination of thread:
        WaitForSingleObject(hThreadEnh,INFINITE );
        CloseHandle(hThreadEnh);
        // Stop the benchmarking counter:
        clock_counter.stop();

        // Output results:
        std::cout << "Latency:␣" << c.time_elapsed() << std::endl;
    }
    return 0;
}
```

• *Source Code 4: Test application to benchmark thread termination latencies on the Windows platform*

Source Code 4 shows the benchmarking of thread termination latencies. The results confirm the measured latencies in Figure 23. Similar, by benchmarking the creation and destruction of Windows threads, we discovered that creating and destroying threads consumes approximately 500 μs. This overhead is caused by the Windows operating system overhead.

The original implementation includes the overhead for creating and destroying threads on a per frame basis, which influences parts of the measurements. It is beneficial to adapt the worker-thread implementation such that creation and destruction of threads on a per frame basis is avoided, by reusing the initially created threads.

5.2.3 INTERMEZZO: REIMPLEMENTING WORKER THREADS ON WINDOWS

In order to avoid the creation and destruction of threads on a per frame basis in the Matlab simulation, we create a worker thread for the scalable priority processing algorithm upon initialization of the simulation. Similarly, the thread is destroyed upon finalization of the simulation.

The content of the original worker thread is encapsulated by an infinite loop, such that it keeps processing frames for its entire lifespan. The output function controls the worker thread using two binary semaphores:

1. the first semaphore indicates whether new input data is ready to be processed.
2. the second semaphore prevents the thread from working too fast. In case a frame is completed within its deadline, it must wait until the next period for new data.

A prototype of the thread implementation is shown in Source Code 5.

```
unsigned __stdcall ThreadFunc(void* param)
{
  // Start infinite loop (for each frame do)
  while(1) {
    // Measure reaction latency (first measurement is worthless):
    clock_count.stop();
    termination_latency = clock_count.time_elapsed();

    // Prevent thread from working too fast
    // (Block in case previous frame reached 100%):
    WaitForSingleObject(hMutex1 , INFINITE);
    ReleaseMutex(hMutex1); // Immediately release mutex!

```

```
      // Wait until basic function provides input:
      // (Block until input data is ready)
      WaitForSingleObject(hMutex, INFINITE);
      ReleaseMutex(hMutex); // Immediately release mutex!

      resetJobVariables();
      runAlgorithm();
    }
    return 0;
  }
```

• *Source Code 5: Prototype of the thread implementation for scalable parts of the priority processing algorithms.*

Source Code 5 shows that each job first stops a counter for measuring the termination latency. The counter is started at the same position as shown in Source Code 4. Since the synchronization point due to termination of the thread has disappeared, the stop position of the counter is moved to the point just before a new job starts.

5.2.4 TERMINATION LATENCIES CONTINUED

By repeating simulations once again after adapting the thread implementation, we can obtain latency values caused by preliminary terminating frame computations, see Figure 26. The block-polling variant now shows reasonable values. Statistical analysis shows that average latencies for block-based polling, with a 95% confidence interval, are 45.2 $\mu s \pm 0.127$.

Mechanisms for cooperative, preliminary termination are expected to show reaction latencies which are dependent on the computation times. Figure 26, showing the results for block-based polling, and Figure 27, showing the results for pixel-based polling, show comparable latencies in which the standard deviation shows a difference between both variants. Pixel based polling shows average reaction latencies with a 95% confidence interval of 42.2 $\mu s \pm 0.039$.

For both polling variants, the figures show the graphs for periods of 20 ms, 40 ms and 60 ms. The length of the period is indicated in the graphs by deadline (dl). The 20 ms period makes the scalable deinterlacer algorithm terminate during its first stage. The 40 ms period typically terminates the deinterlacer during the filter in between both deinterlacing stages. The 60 ms period makes the algorithm typically terminate during the second stage of the deinterlacer. By choosing these periods, the influence of different algorithmic stages of the scalable algorithm is investigated.

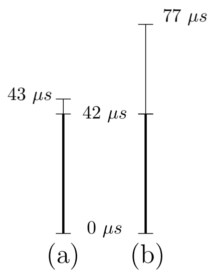
As shown in the graphs of Figure 27 for pixel-based polling, the fluctuations of the latencies are relative small. For block-based polling, the different algorithmic stages are observable in the results of Figure 26. Especially in Figure 26b the transition between the scenes is observable. After the scene-switch, the termination latencies show a reduced value, which corresponds to the analyzing filter function. This filter function does not change any output content and therefore does not process any blocks, such that the granularity of the polling (and the corresponding latencies) differ a bit from the actual deinterlacer algorithms.

Most surprising are the measured latency values for pixel-based polling. Whereas the block-based polling mechanism shows termination latencies in the order of 45 μs, which is a little bit higher than the estimated WCET for block-computations (see Section 5.2.1), the pixel-based polling is expected to show a latency which is approximately a factor of 64 smaller. As shown in the results of Figure 27 and Figure 26, the difference in average latency is approximately 3 μs.

The slightly increased value of the termination latency for block-based polling mechanisms compared to the estimated WCET of a block computation, can be explained by the additional operating system overhead. The polling mechanism enforces a synchronization in time, in our case implemented using binary semaphores (mutexes). The relative high termination latencies measured for pixel-based polling compared to block-based polling can be caused by (a combination of) the following scenarios:

1. The operating system overhead causes most of the latency, such that the relative low(expected value of the termination latency (approximately 0.5 μs) of pixel-based polling is dominated by this overhead.
2. Worst-case reaction latencies for block-based polling are never measured.

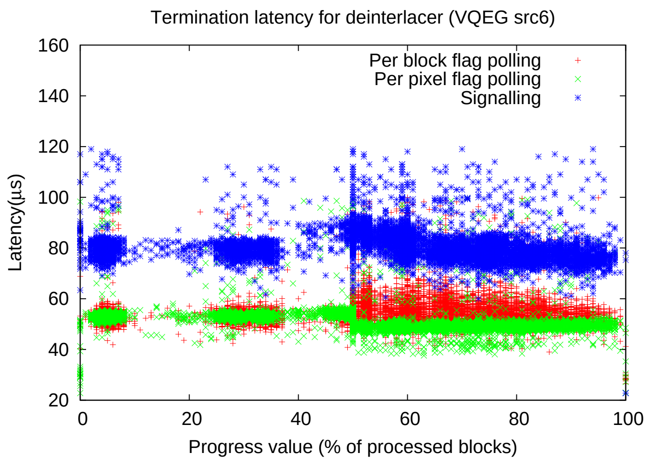
Assume that the operating system overhead is approximately 42 μs. When adding the worst case execution time of a pixel-computation to this time, we obtain a worst-case latency of approximately $42 + 0.5 < 43\mu s$. When applying the same math with block-computations, we obtain a worst-case latency of $42 + 35 = 77\mu s$.



• *Figure 24: Adding WCET for (a) pixel- and (b) block-computations to the synchronization overhead of the operating system, a bigger difference in termination latency might occur between both polling variants as is actually shown by the results of Figure 26 and Figure 27.*

Figure 24 shows the difference in estimated worst-case latency for both polling variants. However, based upon gathered measurements we can conclude that block-based polling variants show a very good trade-off in as well computational overhead as termination latency.

Comparing signalling using the Linux Operating System
In the results so far there is lack of comparison with signalling mechanisms to preliminary terminate jobs. Since the Windows platform does not allow preliminary termination by means of signalling, as described in Section 4.5.3, some additional measurements are done on the Linux platform to investigate feasibility of the signalling approach by means of POSIX signals. Similar experiments as done earlier in this section on the Windows platform for the deinterlacer are performed on the Linux platform to compare polling variants with signalling variants. Figure 25 shows the termination latency times versus progress in percentage of processed blocks.



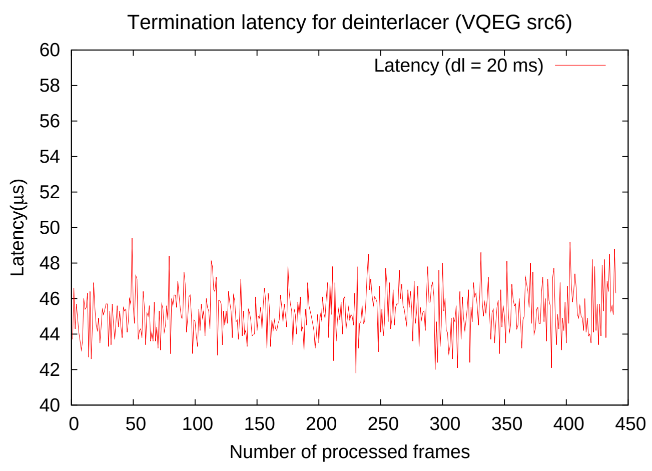
• *Figure 25: Reaction latencies upon preliminary termination of a job which processes a frame, for every stage of the algorithm is, measured in percentage of processed blocks. The progress values have a limited influence on the termination latencies. Three variants are compared: block based polling, pixel based polling and the signalling approach. The experiment uses the VQEG source 6 video.*

Figure 25 is created using collected experimental data for deadline periods, T, in the range: $T \in [10; 120]$ ms. Although the priorities of the threads can not be changed, similar latency trends for polling variants are shown as in the results on the Windows platform. The graph shows a bigger deviation for block-based polling compared to pixel-based polling. Next to this observable difference in deviation, the termination latencies for the polling variants on the Linux platform are slightly higher compared to the Windows variant. This higher latency values are caused by the implementation of the threads, which are not reused in the Linux implementation (as in the original Windows implementation). This means that for every frame-computation a new thread is created and destroyed. The surprising result is that opposite to the Windows platform, the termination latency on the Linux platform is not dominated by the operating system overhead.

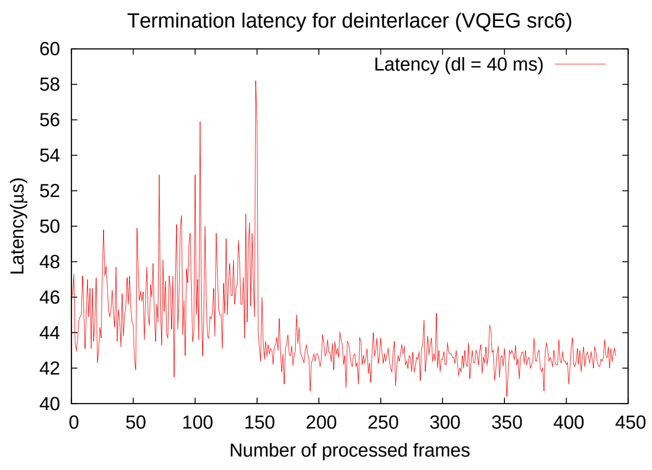
A more surprising result is that the signalling approach gives an even higher latency compared to the block-based polling approach, as well as a higher deviation. The termination latency is in the order of 80 μs, which is almost twice as high as the polling based implementations. This can be explained by investigating the way the signal mechanism provided by the Linux kernel works, see Appendix D. In the background, the signalling approach implements a way of polling (at the granularity of the kernel clock) to check whether a signal has arrived, with the extra overhead of the system call. Since Linux does not provide any guarantees about when a signal is handled (a signal is queued to the receiving thread), latencies can highly fluctuate due to unpredictable timing of the completion of the signal-handler.

It must be noted that the termination latencies corresponding to the signal-based preliminary termination as shown in Figure 25 are filtered. All values above 120 μs are not plotted, however there are a lot of samples exceeding the plotted range. Although signal-based preliminary termination provides advantages in terms of computational overhead, it shows very unpredictable behavior in terms of termination latency. An additional disadvantage is that signalling is not supported on most platforms, including Windows.

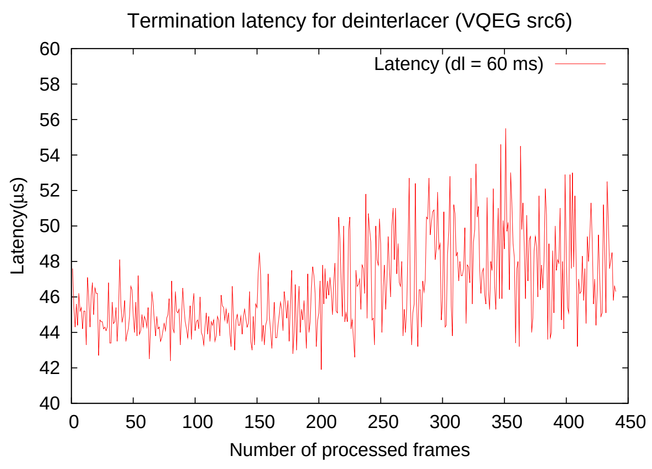
In correspondence with earlier simulation results gathered on the Windows platform, the block-based polling variant has again shown to be an attractive alternative when comparing latencies of different polling granularities.



(a) Termination of jobs by block based polling with deadlines of 20 ms. All frames are terminated during the first phase of the scalable deinterlacing algorithm and therefore the latencies show stable values.

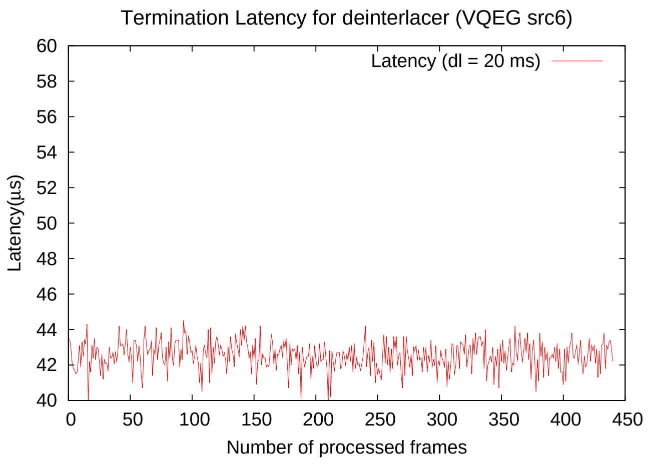


(b) Termination of jobs by block based polling with deadlines of 40 ms. All frames reach a progress around 50%. In the last part of the video, the progress is a little bit higher than 50% and therefore the algorithm is terminated during the filter in-between the two scalable deinterlacing stages. Here the latencies show a minor decreased value.

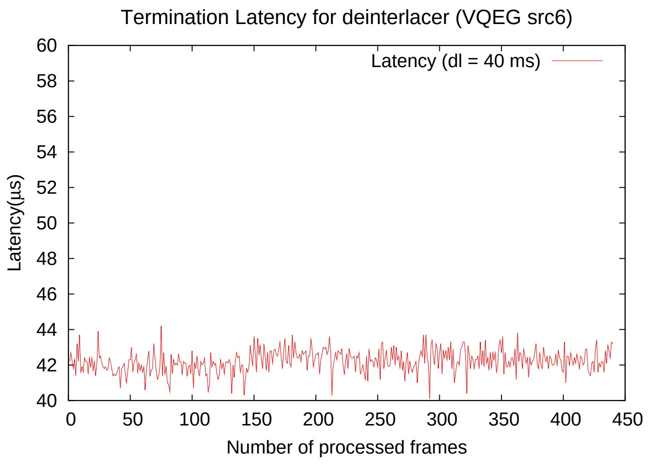


(c) Termination of jobs by block based polling with deadlines of 60 ms. All frames exceed the progress of 50% completion and are therefore terminated during the second deinterlacer stage.

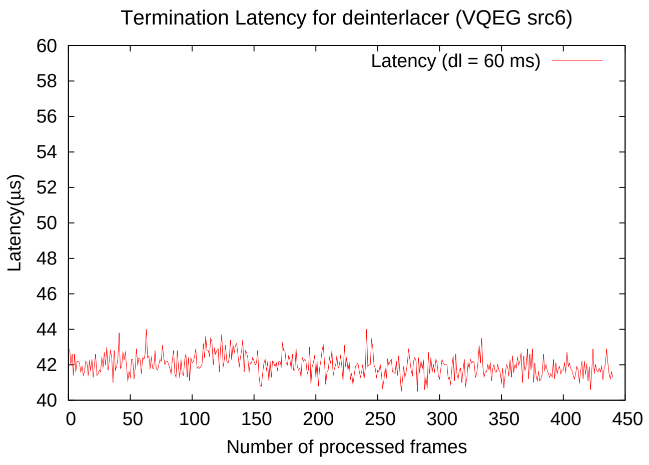
• *Figure 26: Latencies for preliminary terminating frame computations by block-based polling in the deinterlacer application, measured for deadlines of 20 ms, 40 ms and 60 ms for the VQEG src6 video sequence.*



(a) Termination of jobs by pixel-based polling with deadlines of 20 ms.



(b) Termination of jobs by pixel-based polling with deadlines of 40 ms.



(c) Termination of jobs by pixel-based polling with deadlines of 60 ms.

• *Figure 27: Latencies for preliminary terminating frame computations by pixel-based polling in the deinterlacer application, measured for deadlines of 20 ms, 40 ms and 60 ms for the VQEG src6 video sequence. Latencies show stable trends, with less fluctuations than block polling results, see Figure 26.*

5.3 OPTIMIZATION OF RESOURCE ALLOCATION MECHANISMS

Experiments in this section make use of block-based polling as a mechanism for preliminary termination, since Section 5.2.4 has shown that the block-based polling mechanism give the best trade-off in latency versus computational overhead on the simulation platform. Whereas simulations trading-off the performance of mechanisms for preliminary termination where done by a priority processing application with a single deinterlacer component, now we will consider an application with two independent, competing components, as introduced in Section 4.1. This example application is composed from two independent priority processing algorithms for deinterlacing and sharpness enhancement of video content, which have to share the same processor resources.

Schiemenz [32] investigates multiple strategies for dynamically allocating time-slots to competing algorithms, from which we especially consider two strategies: one based on Round Robin (RR) and another based on Reinforcement Learning (RL). These strategies are implemented within the decision scheduler. The decision scheduler is activated after every time-slot and decides upon activation to which algorithm the next time-slot is assigned based upon the upfront defined strategie.

The advantage of performing measurements in an environment using the round-robin scheduling policy is that it is know upfront how many time-slots are allocated for each scalable algorithm. Suppose that an application consists of m amount of scalable algorithms competing for processor time; and that a period, in which a frame-computation must fit, consists of N timeslots. Applying a round robin schedule on the scalable algorithms means that every algorithm gets in total at least $\lfloor \frac{N}{m} \rfloor$ time-slots in each period, leaving $N \; mod \; m$ time-slots to be allocated such that an algorithm might receive at most one time-slot more than another algorithm within the application. The round robin scheduling policy divides the time-slots alternating over the competing priority processing algorithms, such that after every time-slot context-switches appear. The allocation of consecutive time-slots to the same algorithm is not applicable in round robin scheduling, such that reduction of context-switching, as explained in Figure 18 (Section 4.5.1), is also not applicable.

Opposite to round-robin, the reinforcement learning policy makes its decisions based upon a probabilistic model, as explained in Section 2.3.2. The reinforcement learning policy relies on the availability of progress values relative to the amount of consumed time-slots. Based upon this information, a time-slot is assigned to an algorithm, which means that the total amount of time-slots given to an algorithm (within a period to process a frame) can change per period, e.g.: frame-computation.

To allocate CPU time, a task implementing a scalable priority processing algorithm, is assigned the processor by means of two comparable implementations (as explained in Section 3.5):

1. suspending and resuming the task;
2. manipulating the task priority such that the native fixed priority scheduler (FPS) of the platform can be used.

The latter option allows the consumption of gain-time, see Section 3.3. A subject of investigation is the possible gain of priority processing algorithms due to provisioned gain-time by other algorithms within the applications. Furthermore, the resource allocation mechanisms are implemented such that context switches are reduced when possible, as described in Section 4.5.1.

In order to compare different implementations of scheduling mechanisms, a measurement unit is required. The amount of work, W, in a frame f is defined in terms of amount of processed blocks per time-slot, see Equation 10.

$$W(f, t_s) = \frac{\#\text{Processed Blocks}}{\#(\Delta t_s)} \tag{10}$$

Note that the notion of processed blocks in a frame is algorithm dependent, since only stages of the algorithm which change the output buffer increase the number of processed blocks value (see Section 4.4.5). The measurement unit W allows to define the *relative gain* between two different implementations. The *relative gain* is the fraction between the work done in a new implementation and a reference implementation, as shown in Equation 11.

$$\text{relative gain}(f, t_s) = \frac{W_{new}(f, t_s)}{W_{ref}(f, t_s)} \tag{11}$$

The relative gain gives insight in the additional amount of work within a frame, that an algorithm is capable to perform within the same amount of time compared to a reference implementation (the implementation provided initially). Each priority processing algorithm has a characteristic function defining the progress over time, see Section 2.2, such that the progress is not linear in time. Although our notion of relative gain gives a performance metric to compare the influence of different control implementations on the progress of a single priority processing algorithm, the relative gain performance metric does not allow comparison between different algorithms.

When considering the relative gain with respect to the available period, the computed relative gain for each frame in the test sequence is averaged. This is repeated for different periods T. Equation 12 shows the average relative gain for a sample video sequence consisting of M frames.

$$\text{relative gain}_{avg}(f, t_s) = \frac{1}{M} \sum_{f=0}^{M-1} \frac{W_{new}(f, t_s)}{W_{ref}(f, t_s)} \tag{12}$$

In order to express the relative gain in percentages, the formulla becomes:

$$\text{relative gain}_{avg}(f, t_s) = (-1 + \frac{1}{M} \sum_{f=0}^{M-1} \frac{W_{new}(f, t_s)}{W_{ref}(f, t_s)}) \times 100 \tag{13}$$

5.3.1 SUSPEND-RESUME VERSUS PRIORITY MANIPULATION

As a first experiment we want to compare suspend-resume to priority manipulation in an equal environment, meaning:

1. gain-time consumption is disallowed. In the priority manipulation implementation gain-time consumption is prevented by introducing an task with a priority in-between the highest and background priority, such that it consumes all gain-time.
2. use the round-robin scheduling policy implemented within the decision scheduler, such that there is no gain from reduced context-switching.

By using the round robin scheduling policy, the divided time-slots over the algorithms is fixed for all frames. The fixed amount of time for an algorithm per frame allows the relative gain to be calculated by simply collecting the processed blocks per frame. The processed blocks per frame in two different implementations can be simply divided, see Equation 14. Equation 14 saves a division over the time-slots compared to Equation 11 and therefore gives lower rounding errors in the measurement results.

$$\text{relative gain}(f) = \frac{W_{new}(f, t_s)}{W_{ref}(f, t_s)} = \frac{\#\text{Processed Blocks}_{new}}{\#\text{Processed Blocks}_{ref}} \tag{14}$$

Figure 28 shows the relative gain of an optimized implementation with priority manipulation relative to the original implementation, which uses suspend-resume to allocate time-slots to algorithms.

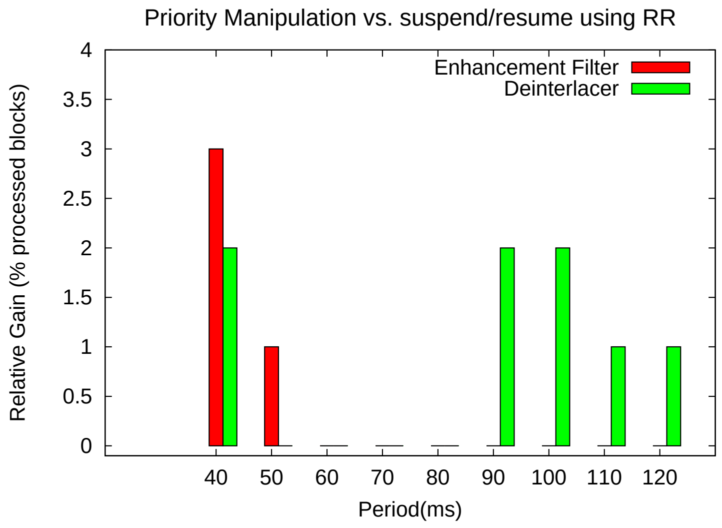
When looking at the relative gain presented in Figure 28, there is a small gain observable. The enhancement filter has finished processing after 50 ms. Therefore the enhancement filter can not gain anything for increased periods. The deinterlacer needs more time to complete a frame. Meanwhile, the deinterlacer does not show any gain in the range where the period $T \in [60; 80]$ ms. During this range, most frames end up around the 50% progress range. The 50% progress range is the transition between the two stages of the deinterlacer. An analysis filter function, which consumes considerable amount of milliseconds of time, is executed between the two stages and no progress in terms of processed blocks is accounted during this filter (as explained in Section 4.4.5). Therefore no gain is observable in Figure 28 for the specified period lengths.

Overall, it can be concluded that disallowing gain-time and no reduced context-switching gives almost similar results in terms of control overhead using priority manipulation as for suspending-resuming of threads.

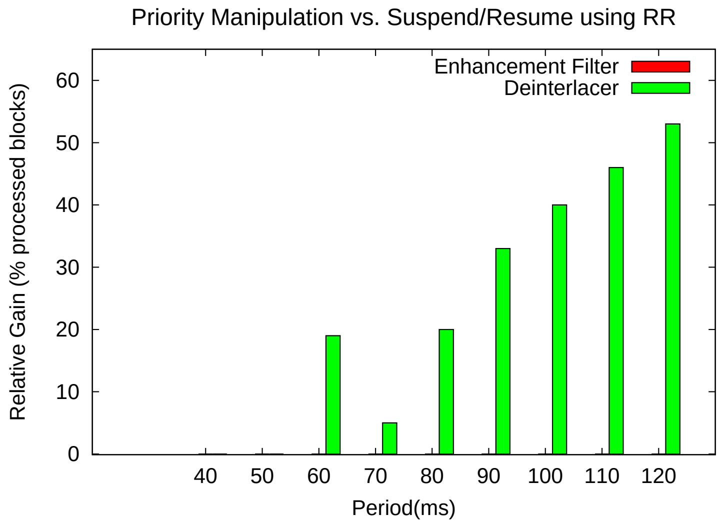
5.3.2 GAIN-TIME USAGE

Due to the characteristics of the round robin policy that the divided time among the algorithms is strictly fixed, it can be tested very easily whether gain-time is consumed. In the next experiment we allow consumption of gain-time, such that the slots which are allocated for the enhancement filter, while the enhancement filter is already finished, can be consumed by the deinterlacer. An strictly increasing trend is expected from the point that the enhancement filter finished its task. Figure 29 shows the results, in which similar to Figure 28 no gain is measured for small periods in which both algorithms run competitive. Thereafter, when the enhancement filter finished processing, the deinterlacer effectively uses the gain-time left by the enhancement filter. Due to the analysis filter function between the two deinterlacer stages, there is a local minimum visible in Figure 29 for a period with a length of 70 ms.

It can be concluded that the gain-time consumption is working as expected and that priority processing algorithms are very suitable for consuming gain-time.



• *Figure 28: Relative gain for the round robin (RR) strategy of scheduling based on priority manipulation compared to suspend-resume. Priority manipulation gives a small improvement over suspend-resume due to reduced control overhead. Gain-time consumption is disallowed.*



• *Figure 29: Relative gain for the round robin (RR) strategy of scheduling based on priority manipulation compared to suspend-resume, allowing gain-time consumption. Priority manipulation allows effective use of gain-time.*

5.3.3 OPTIMIZED CONTROL

Comparing implementations for processor allocation mechanisms have shown so far that priority manipulation and suspending-resuming of tasks perform equally well. As an advantage of priority manipulation mechanism it has been shown that gain-time can be effectively used by background tasks. These earlier experiments make use of the round robin scheduling policy.

The reinforcement learning monitors the algorithms and assigns time-slots on basis of observed progress values. Unlike the round robin policy, the reinforcement learning policy typically assigns multiple consecutive slots to the same algorithm. The decision scheduler runs on its own processor and learns from previous results. This learning characteristic typically leads to the result that multiple consecutive slots are given to the same algorithm. Assignment of multiple consecutive slots to the same algorithm means that we can reduce context-switching overhead, as explained in Section 4.4.5.

The available gain-time in an application using the reinforcement learning scheduling policy is expected to be relatively low. Two different scenarios of gain-time provisioning can be identified:

1. When an algorithm completes a frame during a time-slot, the remaining fraction of the time-slot becomes available to other algorithms within the application. This situation occurs maximal once per frame-computation and cosnsists of a relative low amount of gain-time.
2. When the decision scheduler assigns a time-slot to a blocked algorithm, a complete time-slot is available as gain-time. This situation occurs very occasionally as the result of the exploration step [4] within the reinforcement learning algorithm. The exploration step, made randomly with a small probability within the reinforcement learning policy, is to investigate whether more optimal solutions are reachable. This random exploration step can also lead to assignment of a time-slot to an algorithm which already completed its frame.

When gain-time consumption is not allowed, the wrongly assigned time-slots are wasted. It might be desirable that wrong decisions influence the results, since the amount of bad decisions indicate the capability of the decision scheduler, compared to other policies, to make appropriate decisions. It is also conceivable to allow background tasks to continue processing ahead, for example: start already with the basic function of the next frame. These are details left for later research. To investigate a relative amount of possible gain, we allow other priority processing tasks to consume the gain-time.

The final experiment consists of a setup in which:

1. Block-based polling is applied as a mechanism for preliminary termination;
2. Priority manipulation using the Windows fixed priority scheduler is applied;
3. Reduced context-switching is applicable when multiple consecutive time-slots are assigned to the same algorithm by the reinforcement learning policy;
4. Gain-time consumption is allowed.

When comparing the new setup with the originally implemented priority processing application, a significant improvement can be observed, as shown Figure 30.

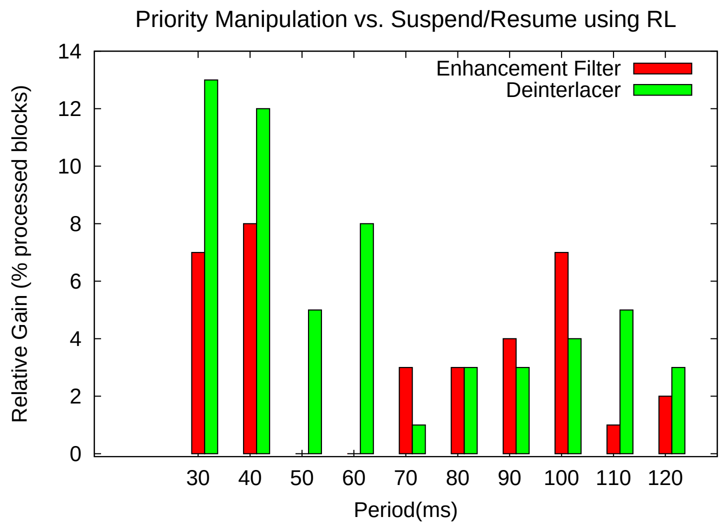
5.3.4 PROCESSOR UTILIZATION

During simulations, the processor utilization of the cores is observed using the Windows Task Manager Performance visualizer. The processor on which the scalable algorithms run is utilized for approximately 40%, which means that the algorithms are limited in using full processor capacity by other factors.

A possible disturbing factor is the Matlab/Simulink environment, which runs a lot of background threads, for example to gather measurement information. This causes inter-thread communication, causing time-synchronization overhead.

Another conceivable distortion is non-optimal memory usage. The priority processing algorithms use a sorted order to process the video content, which differs from the traditional approach in which content is processed in consecutive order. The sorted order leads to non-linear memory access, which causes a high I/O bound.

The combination of both factors can cause a even worse scenario. Since multi-core machines make use of a shared-cache, it is possible that the background threads of Matlab pollutes the cache of the video-processing threads and vice versa. In order to investigate efficiency of memory allocation, mapping of the priority processing



• *Figure 30: Relative gain for the reinforcement learning (RL) strategy of scheduling based on priority manipulation compared to suspend-resume. Priority manipulation gives a significant improvement over suspend-resume due to reduced control overhead and gain-time consumption.*

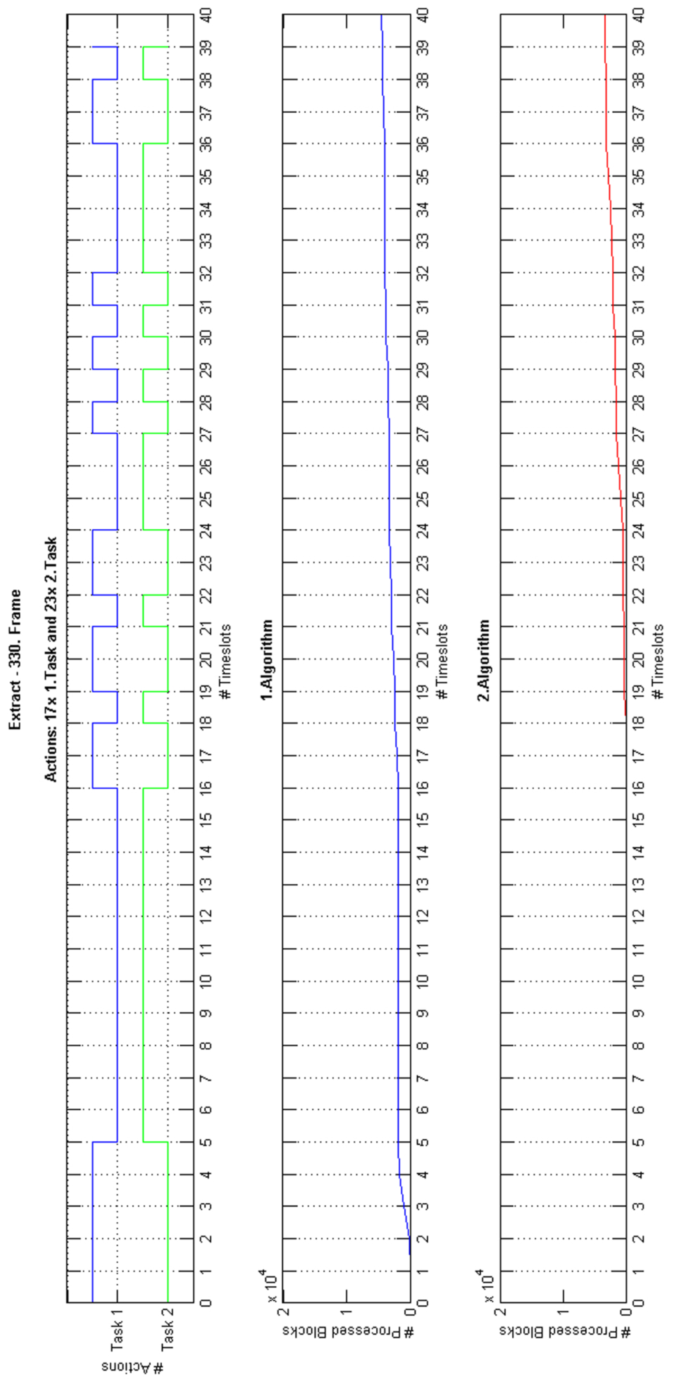
applications is required to an embedded platform, such that the Matlab/Simulink interference is eliminated. The competing algorithms, which are dynamically allocated the same processor resource, are also using the same cache. When the system engineer has sufficient control over the cache or local memory, it is conceivable to apply a cache partitioning scheme, such as described by Pérez et al. [28].

5.4 MONITORING

Monitoring mechanisms keep track of the consumed time on a time-slot scale. Inspecting of the progress values of the priority processing algorithms is done by the decision scheduler upon activation after each time-slot. Due to timing issues with timer-interrupts on the Windows platform, periodic activation of the decision scheduler is implemented by means of a busy-waiting loop in our simulation environment, as explained in Section 4.5.2. The busy-waiting loop reads the Time Stamp Counter (TSC) [1] via a RDTSC instruction to keep track of time. Similar methods are used for benchmarking, as described in Appendix C. In a real-time environment it is more straightforward to implement the decision scheduler with a software interrupt by making use of high-resolution timers.

Another consideration is the behavioral correctness of the priority manipulation implementation compared to the suspend-resume implementation. By disallowing gain-time consumption, only the algorithm which is assigned to a time-slot is allowed to make progress, as was the case in the original implementation. The original implementation simply assumes this property and therefore only the progress value of the algorithm which is assigned the latest time-slot is updated. By using priority manipulation, it is conceivable that a background priority consumes processor cycles when higher priority tasks are blocked. Therefore, the monitoring mechanism is extended such that the progress values of each competing algorithm (instead of only the activated algorithm) is updated after each time-slot.

An example schedule is shown in Figure 31 for a random frame during processing of a video sequence. The schedule in Figure 31 also shows that the reinforcement learning assigns multiple consecutive time-slots to a single algorithm. Both algorithms have a start-up time in which they are assigned time-slots, but do not show any progress. The deinterlacer has a start-up time of approximately ten milliseconds, such that periods smaller than (or close to) the start-up time do not give useful progress statistics. Furthermore, it is shown that only the task with the highest priority makes progress, such that original system behavior is preserved by priority manipulation disallowing gain-time consumption.



• *Figure 31: The decision scheduler assigns time-slots to the algorithms. Only the algorithm assigned to a time-slot is allowed to make progress. Multiple consecutive time-slots are typically assigned to the same algorithm.*

6. CONCLUSIONS

Flexible signal processing on programmable platforms are increasingly important for consumer electronic applications and others. Scalable video algorithms using novel priority processing can guarantee real-time performance on programmable platforms even with limited resources. Dynamic resource allocation is required to maximize the overall output quality of independent, competing priority processing algorithms that are executed on a shared platform.

Since the optimization of the priority processing applications has become the most important contribution of this project, this report emphasizes on this subject. Research work related to the project, done in advance to the internship period, is available in the appendices as additional information.

6.1 OPTIMIZING THE CONTROL OF PRIORITY PROCESSING

The priority processing application is introduced as an example application requiring dynamic resource allocation mechanisms. We described three basic mechanisms for dynamic resource allocation to competing priority processing algorithms:

1. **Preliminary Termination:** Upon expiration of the periodic deadline, all scalable priority processing algorithms must be preliminary terminated, such that a new frame computation can be started.
2. **Resource Allocation:** Multiple competing priority processing algorithms are sharing a single processor, which requires mechanisms to allocate the processor resources.
3. **Monitoring:** Assignment of the processor resources is done on a time-slot scale by the decision scheduler. The decision scheduler needs progress values and the consumed time of the algorithms in order to make an appropriate decision on the time-slot assignment.

Since the priority processing application is prototyped in a Matlab/Simulink environment on a non-real-time platform, Windows XP, it took a lot of effort to bound the interference of the simulation environment and other background processes. The simulation environment of the priority processing application requires a multi-core machine to give accurate simulation results. Nevertheless, different implementations for these mechanisms are implemented within the Matlab/Simulink simulation environment (described in Section 4) on general purpose machines, allowing us to make performance comparisons for different implementations of the resource allocation mechanisms. Measurement data is presented in Section 5.

For preliminary termination, cooperative termination is preferred, since most platforms, including Windows, do not support reliable termination of tasks through signalling. Simulation results show that the polling overhead is relatively low on general purpose machines compared to the total computation times to process a full frame. The block based polling variant gives a slightly more deviated reaction latency compared to pixel based polling, but is a relative good alternative when comparing it with the computational overhead of pixel based polling and the relative high latency of the signalling approach. Cooperative preliminary termination mechanisms, such as polling, allow the trade-off between latency and computational overhead, but have the disadvantage of the high effort for code maintenance.

Preliminary termination by means of signalling is tested by means of the POSIX signals provided by the Linux kernel. This required porting of the priority processing application to the Linux platform, with the side-effect that a lot of failures are fixed. Apart from this effort, it is shown that the termination latency obtained from measurements from signalling, are non-predictable. The unpredictable latency behavior of signals shows the lack of real-time guarantees within Linux systems.

Resource allocation can be efficiently implemented by priority manipulation, using the native operating system's scheduler, e.g. fixed priority scheduler of Windows. Although there is no observable difference in performance between suspending-resuming of threads and priority manipulation, it is shown that the priority processing applications can effectively use gain-time. Gain-time consumption is enabled by the use of priority manipulation, such that background threads can consume the otherwise wasted processor cycles.

Monitoring consists of querying the progress of the algorithms after each time-slot. The time-slot scale must be ensured, since the scheduling policy (decision scheduler) together with the resource allocation and preliminary termination mechanisms rely on correct timing information. Accounting of time-slots relies on the availability of high-resolution timers. Due to the absence of high-resolution timers on the Windows platform, the models are simulated by mapping the scheduling thread on a separate core. The scheduler thread keeps track of the time by performing a busy-waiting loop.

In our simulation environment, the processor utilization for the core running the scalable algorithms was approximately 40%. This relative low processor utilization can be explained by a combination of the following factors:

1. Inefficient memory allocation by the signal processing algorithms. Due to the data-extensive character of video processing, optimizing the way memory is accessed might give considerable advantage. Optimizations for memory allocations require control over the memory controller, which is typically not available on general purpose operating systems such as Windows and Linux, but is available on embedded platforms.
2. Interference of the simulation environment. The Matlab/Simulink environment runs a lot of threads in the real-time class during the simulation. These threads can cause extra overhead due to enforced time-synchronization for extracting measurement results, as well as cache pollution due to shared caches.

A next step is to implement priority processing applications in an real-time environment, such that:

1. actual overhead of polling mechanisms can be investigated, which is expected to be much higher on embedded platforms using streaming processors;
2. the overhead of the simulation environment is reduced, i.e. Matlab threads and background processes running in Windows;
3. it is shown that the application is feasible to realize real-time video processing, e.g. mapping on an embedded (real-time) hardware platform.

A first approach towards deployment of priority processing applications in a real-time environment is sketched in Section 6.2.

6.2 FUTURE WORK

In this section we will indicate further directions of research in the field of the priority processing application and do a proposal on implementing the priority processing on accelerating hardware. This includes important considerations which have to be taken into account on hardware platforms other than the general purpose platforms on which the Matlab/Simulink models are implemented.

6.2.1 TOWARDS SYSTEM DEPLOYMENT: MAPPING ON HARDWARE ACCELERATED PLATFORMS

Until now, the priority processing applications are simulated in a Matlab/Simulink environment. The concept of priority processing shows promising results as well in a qualitative manner as the effective use of computational resources. Current research lacks an implementation of the priority processing algorithms proving the capability to perform in real-time. Therefore, a straightforward direction for further research is the mapping of one or more algorithms to a hardware accelerated platform, which is widely available.

Trends in embedded platform design move towards Multi-Processors System on Chip (MPSoC). MPSoCs can be subdivided in heterogeneous, composed of multiple different processing elements on chip, and homogeneous, composed of multiple equal processing elements. A typical trade-off in MPSoC design is to put a general purpose processing element for control purposes, such as an ARM or MIPS, and multiple streaming processors for signal processing, such as VLIWs and DSPs.

It is therefore an attractive option to map the decision scheduler on a general purpose processor, controlling the streaming processors by means of appropriate mechanisms for preliminary termination, resource allocation and monitoring. The resource allocation mechanism makes the system more complicated, due to multiple algorithms sharing the same computational resource. The mapping of priority processing applications on embedded platforms can be split in two stages:

1. Map an application with a single component to the platform. This only requires appropriate mechanisms to account the time within a period and to preliminary terminate a job upon depletion of the periodic budget. This implementation allows further optimization of the algorithm on the platform, for example: optimize memory usage.
2. Map a multi-component to the platform. After optimizing the single components, it can be investigated how multiplexing of computational resources can be optimized. Since streaming processors typically do not support preemption of tasks, current research is directed towards parallel processing.

An example of an attractive platform to map the priority processing application, is the Cell Broadband Engine [2] (Cell BE), developed by Sony, Toshiba and IBM. The Cell BE platform is extensively used in industry, the academic world and hobbyists, due to its wide availability of documentation and open architecture descriptions. On the one hand the Cell BE architecture provides very powerful streaming processors to perform signal processing computations. On the other hand, it addresses the complexity of efficiently transferring data over a shared bus and dealing with limited local memory capacity.

The implementation of a dynamic resource allocation scheme includes the following efforts concerning the three identified mechanisms:

1. **Preliminary Termination:** Preliminary termination can be applied by means of cooperative termination using the message-box mechanism. These are special registers which allow efficient communication between host processor and streaming processors. A trade-offs in granularity of termination is expected to show a clear difference in overhead, since the polling mechanism disturbs the signal processing pipeline (opposite to general purpose machines).
2. **Resource Allocation:** It is unlikely that a time-slot-based scheduler is implemented on this platform. Dividing time-slots on the streaming processors causes a lot of additional overhead for preempting algorithms. Streaming processors typically do not allow preemption, due to the enormous context-switching overhead and run-time data-dependecies. Furthermore, it requires a lot of effort to map the single components to a streaming processor. Next to the implementation effort, the Cell BE consists of six streaming processors, which allows parallel execution of a priority processing algorithm on each of them. This comes at the additional cost of managing the shared communication bus, which can be seen as an additional optimization objective since the bus allows multiple
3. **Monitoring:** Since scheduling of algorithms on a time-slot basis is not required, the periodic deadlines have to be accounted to support the preliminary termination mechanism. This can be achieved by setting timers on regular intervals. When a timer fires, it enforces the algorithms to terminate processing of the current frame and perform a roll-forward to the next frame.

The Cell BE is a typical consumer electronic platform which is capable of displaying high performance video, but does not run a real-time operating system. In this report, we considered an implementation of priority processing applications based upon availability of fixed priority scheduling. More complex systems move to partitioning of available resource capacity, by providing virtual platforms.

6.2.2 VIRTUAL PLATFORMS

To investigate real-time performance in a shared resource environment, it is conceivable to run the application on a virtual platform. The virtual platform can be provided by applying a resource reservation scheme, as described in Section 2.5. Resource reservations allow independent development of applications, requiring a system wide feasibility test upon integration of the application within the system. The reservation scheme is based upon four mechanisms Rajkumar et al. [29], as described in Section 2.5: *admission control*, *scheduling*, *enforcement* and *accounting*.

Virtual platforms divide available resources in partitions and guarantee the assigned budget. A disadvantage of these virtualization techniques is the introduction of additional complexity for the monitoring mechanism within the priority processing application. The monitoring mechanism must ensure time-slots of fixed size, typically 1 ms, after which the decision scheduler must be activated. Activation of the decision scheduler can not be considered as strictly periodic relative to the absolute time within virtual platforms.

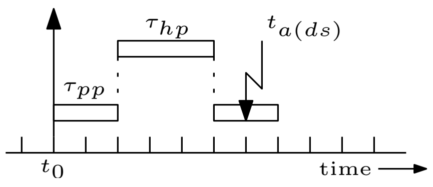
As an example assume a system with two independent applications, which equally share a single processor:

1. An application implemented in a task, τ_{hp}, at the highest priority;
2. A priority processing application implemented in a lower priority task, τ_{pp},.

Assume that within the priority processing application, it is desired to fire the decision scheduler every 3 time-units. Figure 32 shows a scenario in which the virtual time is not equal to the real time.

The accounting mechanism of the resource reservation system, accounts the time used by each application. It is possible to implement a virtual timer based upon this accounting mechanism. Assume we have the possibility to implement a hook function upon preemption and re-activation of the priority processing application, timer interrupts can be used as follows:

1. Upon preemption of the priority processing application, all timers have to be reset.
2. Upon re-activation of the priority processing application, the consumed budget must be queried and the new activation time must be derived to set a timer.



• *Figure 32: Activation of the decision scheduler must be ensured at $t_{a(ds)}$. Virtual time $\neq$ Real-time: in virtual time 3 time units expired counted from t_0, whereas in real-time is 6 time-units expired.*

Since a budget is typically defined in Giga Operations Per Second (GOPS), the derivation of activation times requires knowledge about the platform. When the amount of operations that the processor can compute per second; the amount of operations assigned per period; and the activation frequency of the decision scheduler is known, the virtual time can be translated to a real time.

REFERENCES

[1] Using the rdtsc instruction for performance monitoring. Technical report, Intel Corporation, 1997. URL http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM.

[2] Cell broadband engine resource center, 2009. URL http://www.ibm.com/developersworks/power/cell/index.html.

[3] Openmp.org, 2009. URL http://openmp.org/.

[4] Alex M. Andrew. Reinforcement learning: An introduction by richard s. sutton and andrew g. barto, adaptive computation and machine learning series, mit press (bradford book), cambridge, mass., 1998, xviii &plus; 322 pp, isbn 0-262-19398-1, (hardback, 31.95). *Robotica*, 17(2):229–235, 1999. ISSN 0263-5747.

[5] Moris Behnam, Thomas Nolte, Insik Shin, Mikael Åsberg, and Reinder J. Bril. Towards hierarchical scheduling on top of vxworks. In *Proceedings of the Fourth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'08)*, pages 63–72, July 2008. URL http://www.mrtc.mdh.se/index.php?choice=publications&id=1489.

[6] Jens Berger. Steuerung von skalierbaren Priority-Processing-Algorithmen. Master's thesis, Brandenburgische Technische Universität Cottbus, July 2007.

[7] M. Bergsma, M. Holenderski, R.J. Bril, and J.J. Lukkien. Extending rtai/linux with fixed-priority scheduling with deferred preemption. In *Proc. 5th International Workshop on Operating System Platforms for Embedded Real-Time Applications (OSPERT)*, page 5 14, June 2009.

[8] Lubomir F. Bic and Alan C. Shaw. *Operating Systems Principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2002. ISBN 0130266116.

[9] Reinder J. Bril, Johan J. Lukkien, and Wim F. J. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited. In *ECRTS '07: Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 269–279, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2914-3. doi: http://dx.doi.org/10.1109/ECRTS.2007.38.

[10] Alan Burns and Andrew J. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0201729881.

[11] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004. ISBN 0387231374.

[12] Henk Corporaal. *Microprocessor Architectures: From VLIW to Tta*. John Wiley & Sons, Inc., New York, NY, USA, 1997. ISBN 047197157X.

[13] Matteo Frigo. cycle.h, August 2007. URL http://www.fftw.org/cycle.h.

[14] Video Quality Experts Group. Test sequences, 2007. URL ftp://vqeg.its.bldrdoc.gov/SDTV/VQEG_PhaseI/TestSequences/.

[15] Robert S. Hanmer. *Patterns for Fault Tolerant Software*. Wiley, Chichester, England, 2007. ISBN 978-0-470-31979-6.

[16] Neil B. Harrison and Paris Avgeriou. Incorporating fault tolerance tactics in software architecture patterns. In *SERENE '08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems*, pages 9–18, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-275-7. doi: http://doi.acm.org/10.1145/1479772.1479775.

[17] C. Hentschel and S. Schiemenz. Priority-processing for optimized real-time performance with limited processing resources. *International Conference on Consumer Electronics (ICCE), 2008. Digest of Technical Papers.*, Jan. 2008. doi: 10.1109/ICCE.2008.4588059.

[18] C. Hentschel, R.J. Bril, and Yingwei Chen. Video quality-of-service for multimedia consumer terminals - an open and flexible system approach. *Communications, Circuits and Systems and West Sino Expositions, IEEE 2002 International Conference on*, 1:659–663 vol.1, June-1 July 2002.

[19] Mike Holenderski, Reinder J. Bril, and Johan J. Lukkien. Swift mode changes in memory constrained real-time systems. In *Proc. IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC09)*, 2009.

[20] IEEE and The Open Group. Ieee std 1003.1-2008, 2008. URL http://www.unix.org/2008edition/.

[21] Stephen F. Jenks, Kane Kim, Emmanuel Henrich, Yuqing Li, Liangchen Zheng, Moon H. Kim, Hee-Yong Youn, Kyung Hee Lee, and Dong-Myung Seol. A linux-based implementation of a middleware model supporting time-triggered message-triggered objects. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 0:350–358, 2005. doi: http://doi.ieeecomputersociety.org/10.1109/ISORC.2005.2.

[22] Sven Joost. Untersuchung von Methoden des Reinforcement Learning (RL) zur Steuerung skalierbarer Priority Processing Algorithmen. Master's thesis, Brandenburgische Technische Universität Cottbus, October 2008.

[23] Björn-Oliver Knopp. Implementierung eines skalierbaren Deinterlacers auf das Evaluation-Board TMDX-EVM642 von Texas Instruments. Master's thesis, Brandenburgische Technische Universität Cottbus, September 2008.

[24] Paulo Martins and Alan Burns. On the meaning of modes in uniprocessor real-time systems. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 324–325, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-753-7. doi: http://doi.acm.org/10.1145/1363686.1363770.

[25] Dave McCracken. Posix threads and the linux kernel. In *Ottawa Linux Symposium*, pages 330–337, Austin, Texas, June 2002. IBM Linux Technology Center.

[26] Clifford Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *IEEE International Conference on Multimedia Computing and Systems(ICMCS)*, pages 90–99, May 1994.

[27] Microsoft Developer Network. Processes and threads, 2009. URL http://msdn.microsoft.com/en-us/library/ms684841(VS.85).aspx.

[28] Clara Otero Pérez, Martijn Rutten, Liesbeth Steffens, Jos van Eijndhoven, and Paul Stravers. *Resource Reservations in shared-memory multiprocessor socs*, volume 3, chapter 5, pages 109–137. Philips Research, 2005. ISBN 1-4020-3453-9.

[29] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proc. SPIE, Vol. 3310, Conference on Multimedia Computing and Networking (CMCN)*, pages 150–164, January 1998.

[30] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM. doi: http://doi.acm.org/10.1145/800027.808467.

[31] Jorge Real and Alfons Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Syst.*, 26(2):161–197, 2004. ISSN 0922-6443. doi: http://dx.doi.org/10.1023/B:TIME.0000016129.97430.c6.

[32] S. Schiemenz. Echtzeitsteuerung von skalierbaren Priority-Processing Algorithmen. *Tagungsband ITG Fachtagung - Elektronische Medien*, pages 108 – 113, 17–18 March 2009.

[33] S. Schiemenz and C. Hentschel. De-interlacing using priority processing for guaranteed real time performance with limited processing resources. In *Proc. Digest of Technical Papers. International Conference on Consumer Electronics ICCE 2009*, pages 1–2, 10–14 Jan. 2009.

[34] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects.* Wiley, 2000.

[35] Liesbeth Steffens, Gerhard Fohler, Giuseppe Lipari, and Giorgio Buttazzo. Resource reservation in real-time operating systems ? a joint industrial and academic position. In *International Workshop on Advanced Real-Time Operating System Services (ARTOSS)*, pages 25–30, July 2003.

[36] Dr. B. Thangaraju. Linux signals for the application programmer. *Linux J.*, 2003(107):6, 2003. ISSN 1075-3583.

[37] Guo-Song Tian, Yu-Chu Tian, and C. Fidge. High-precision relative clock synchronization using time stamp counters. pages 69–78, 31 2008-April 3 2008. doi: 10.1109/ICECCS.2008.39.

[38] M.A. Weffers-Albu, J.J. Lukkien, and P.D.V. van der Stok. Analysis of a chain starting with a time-driven component. *Proceedings 3rd Philips Symposium on Intelligent Algorithms, SOIA 2006*, December 6-7 2006.

[39] Clemens C. Wüst, Liesbeth Steffens, Wim F. Verhaegh, Reinder J. Bril, and Christian Hentschel. Qos control strategies for high-quality video processing. *Real-Time Syst.*, 30(1-2):7–29, 2005. ISSN 0922-6443. doi: http://dx.doi.org/10.1007/s11241-005-0502-1.

[40] J. Xu, B. Randell, and A. Romanovsky. A generic approach to structuring and implementing complex fault-tolerant software. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 0:0207, 2002. doi: http://doi.ieeecomputersociety.org/10.1109/ISORC.2002.1003704.

Dynamic Resource Allocation in Multimedia Applications

Part I
Appendices

A. MODE CHANGE PROTOCOLS IN REAL-TIME SYSTEMS

In this report it is assumed that an application can adapt its quality level depending on the available system resources. A general approach, and more coarse grained approach compared to priority processing, is to design an application with different run-time *modes of operation*. A mode of operation is directly mapped onto a software component, representing a quality level with associated resource demands.

The notion of a mode of operation, as described in Section 1 in the context of scalable video algorithms, asks for a more formal definition. In existing literature, several definitions of a mode or a mode change request appear, but do not comply. For example, Martins and Burns [24] define a mode change as follows:

> "A mode change of a real-time system is defined as a change in the behavior exhibited by the system, accompanied by a change in the scheduling activity of the system."

The behavior of a system is an informal term, and therefore lacks a formal definition. From an application point of view, a mode change causes a change in the external behavior as a consequence of the change of the schedule. Real and Crespo [31] try to come up with a more precise definition, which includes reasoning about the scheduling behavior:

> "Each mode produces different behavior, characterized by a set of functionality that are carried out by different task sets."

This definition does not take into account visible artifacts from the end-user point of view. An example is a video decoder that can process an MPEG video-stream at different quality levels. The change of the quality level is visible for the end-users. On the opposite, from a system perspective a mode change only causes a change in the scheduling activity, which in real-time environments asks for temporal guarantees during this transient stage. The definition of a mode change from a system perspective does not address visual artifacts in the output.

An event which triggers a mode transition is defined as a *mode change request*.

Several *mode change protocols* in fixed priority preemptive scheduled real-time systems are compared to each other by Real and Crespo [31]. In this paper, three main features for handling mode changes are identified which classify a protocol:

1. When a mode change request arrives, old mode tasks must be aborted. There are different approaches to deal with task abortion:
   (a) Immediate abort all old mode tasks: When a task is not part of the new mode, it is assumed that its output is useless in the new mode. However, this might introduce data inconsistency.
   (b) Complete all old mode tasks: This approach makes the data inconsistency disappear. On the other hand, this might give rise to longer mode change latencies.
   (c) Combine both extremes: It might be desirable to make a trade-off in latency and output quality.
2. The activation pattern for unchanged tasks during the mode transition. Some protocols preserve the periodicity of unchanged tasks in the system, while others might delay a task or alter the activation rate. This might introduce data inconsistency.
3. The ability to combine the execution of new and old mode tasks:
   (a) **Synchronous** mode change protocols do not release new mode tasks until the last activation of the old mode tasks have finished. Therefore, no overload situation will occur during the mode switch, provided that the old mode is schedulable.
   (b) **A-synchronous** mode change protocols allow old and new mode tasks to be combined in time in order to provide faster transitions. This can cause overload situation and therefore requires specific schedulability analysis of the transition, but decreases the mode change latency such that a better promptness is achieved.

Furthermore, mode change protocols must deal with the following requirements, which can be used as criteria in order to compare different mode switch approaches:

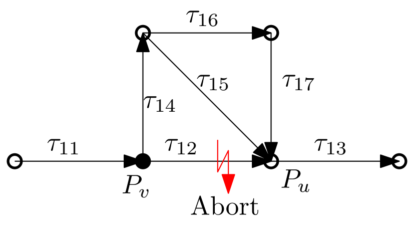
1. **Schedulability:** A mode change request might give rise to newly arriving tasks within the system. During the mode switch, the old tasks as well as the new mode tasks must meet their deadline. Therefore, the new arriving tasks might cause an overload situation.
2. **Periodicity:** During the mode change it is desirable that the activation pattern of unchanged tasks remains unaltered.
3. **Promptness:** This criteria models the expired time after a mode change request before a new mode task has completed. A new mode task might have a deadline, for example when moving to an emergency mode.
4. **Consistency:** Shared resources must be used in a consistent way. A consuming job should not access data which is not fully complete due to preemption of the producing job.

Holenderski et al. [19] describes the use of a synchronous protocol in resource constrained systems and provides the analysis for fixed priority preemptive scheduling (FPPS) as well as fixed priority deferred scheduling (FPDS). FPDS has a lower mode change latency bound.

When considering multimedia applications allowing multiple modes of operation, it is conceivable to distinguish mode changes resulting in a quality mode increase and a quality mode decrease. When the mode of a multimedia application is increased, it typically means that additional work needs to be done in order to obtain the increased quality level. When a mode is decreased, it typically means that part of the work corresponding to the current mode becomes superfluous and can therefore be discarded. This scenario provides the possibility to preliminary terminate jobs in case of a quality mode decrease within a multimedia application.

A.1 TASK STRUCTURE

An application consists of one or more tasks, which can use one or more components. In the model presented by Holenderski et al. [19], a mode is directly mapped onto a component, describing the component's resource requirements. These tasks each can be represented by a directed a-cyclic task graph. An example is shown in Figure 33. The nodes in this graph represent *decision points*, in which the following subtask is selected. Note that a cyclic task graph would possibly instantiate a job requiring infinite amount of computation time. It is possible to allow cycles in the task graph, as long as all tasks within the cycle allow preliminary termination. Although, a better alternative is to model these cyclic task graphs as periodic tasks, which simplifies the schedulability analysis.



• *Figure 33: Example task graph defining a the task's behavior in terms of sub-tasks and decision-points. The termination signal requires sub-job τ_{12} to preliminary terminate its activities. The previously saved application state is from decision point P_v.*

The *decision points* coincide with *preferred preemption points* and *preferred termination points*. A *preferred preemption point* is a position in the task, where we explicitly allow preemption. Similarly, a *preferred termination points* is a position in the task, where we explicitly allow termination. In case of Fixed Priority Deferred Scheduling (FPDS), preemption and termination of an application is limited to these points only. In case of Fixed Priority Preemptive Scheduling (FPPS), a task may be preempted at arbitrary moments in time. By allowing a task to be terminated at an arbitrary moment in time, it must be ensured that intermediate results are masked for the outside world. This can be done by adapting fault tolerance techniques as described in Appendix B. Therefore, a preliminary terminative task is distinguished from non-preliminary terminative tasks.

A.2 PRELIMINARY TERMINATION OF JOBS

Mechanisms for preliminary termination must ensure that the amount of resources used by a component does not exceed the amount of resources that the component is allowed to use. Each components must not exceed its resource budget, as determined by the Quality Manager Component (QMC).

The controller (QMC) assigns budgets to tasks in the video application. The task must stop its activities when the budget is consumed. This can be achieved by letting the tasks poll on regular intervals for the available budget. The task must give back control to the controller component when the budget is not sufficient to do additional work.

An alternative approach is that the controller component signals pending tasks to stop their activities as soon as their budget is consumed. Signalling requires transfer of control between control component and the different tasks of the multimedia application.

Different considerations concerning mechanisms for preliminary termination of tasks are described in Section 3.6. In the model of Holenderski et al. [19], it is assumed that tasks running in a mode when a mode change request arrives, can not be terminated at arbitrary moment in time, but only at preliminary termination points. A more fine grained approach for adaptable quality levels is the priority processing method, see Section 2.2. Priority processing allows arbitrary preliminary termination, once a basic output is delivered. We can reduce the mode change latency in the model of Holenderski et al. [19] upon a mode change request inducing a reduced quality level, by allowing preliminary termination at arbitrary moments in time upon such a request. Such a special case of arbitrary preliminary termination can be implemented using signalling methods, for example as explained in Appendix D.

A.2.1 MANAGING PRELIMINARY TERMINATION OF SUB-TASKS

Predictable results upon arbitrary preliminary job termination are achieved by bounding the preliminary terminative subtasks to extra requirements:

A15. Subtasks are modeled as recovery blocks, which means that all actions performed by a subtask must be revertible.
A16. The internal state of a subtask must be reset upon preliminary termination, including the internal variables as well as the program counter. This is typically work to be assigned to the mode change completion handler, see Figure 13.

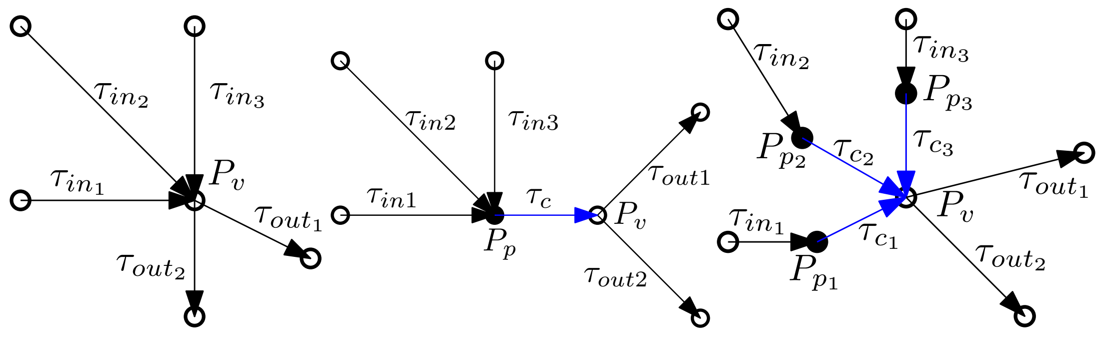
By associating a preemption handler to each decision point, it is possible to extend these points with the optional creation of checkpoints, as presented in Section B. It is possible to create checkpoints on every decision point in the taskgraph. However, this would lead to a disproportionate amount of overhead during runtime in order to obtain a data consistent environment upon preliminary termination. For example, it is very counter-intuitive in memory constraint environments to make checkpoints in an application, which require additional memory resources.

In order to avoid this extra overhead of storing and restoring checkpoints, an alternative approach is to slightly modify the original concept of recovery blocks [30]. It is a task for the programmer to identify subtasks and decision points, see Figure 34. When a preliminary terminative task completes successfully it will end up in an *progression point* in the graph. Note that this progression point differs from the earlier decision points. These progression points are introduced to show that the original task graph, for example see Figure 33, suffers from hidden work to be done inside the decision points. This work also consumes time, which has to be taken into account when analyzing the results.

Considering the example of Figure 33, subtask τ_{12} is modeled as a preliminary terminative task. Decision point P_u hides time consuming actions to be performed. Two types of termination can be considered:

1. **Preliminary termination:** The sub-job is terminated and a jump to the next progression point is made. In this case all internals of the subtask are reset. Note that the next progression point can be a jump back to the initialization of the same task, skipping the part of a frame computation and proceeding with the next frame which instantiates a new job.
2. **Successful Termination:** When the sub-job successfully completes its pending work, there might be some results to be published to other sub-tasks in the application. This means that an incremental update on the total system state is performed. This publish-action is assumed to be short and atomic, since it is a form of intertask communication and therefore is only allowed by non-preliminary terminative tasks.

The subtask responsible for the work to be done upon task completion, is called the *completion subtask*. The completion subtask is not allowed to be preliminary terminated. The work assigned to a completion-subtask is for example writing the computed results from a local buffer to an output buffer (write-behind approach as described in Section 3.7.2). It is the task of a programmer to divide the program properly in preliminary terminative and non-preliminary terminative subtasks. Figure 34 shows how a taskgraph, consisting of preliminary terminative



(a) Task graph consisting of decision points which hide time consuming work.

(b) Introduce a progression point, P_p, after which the task completion is enforced by subtask τ_c.

(c) Introduce multiple progression points, P_{p_1}, P_{p_2} and P_{p_3}, for each incoming subtask, followed by their corresponding completion subtasks τ_{c_1},τ_{c_2} and τ_{c_3}.

• *Figure 34: Task graph modifications enable to show hidden work in decision points. The introduced progression points are indicated as black dots in the extended graph representations. The last graph modification is most accurate, because each subtask differs in the amount of work to complete.*

subtasks, can be transformed to a graph which incorporates the work to be done in order to enforce the completion of a subtask properly. Both successful as well as preliminary termination of a subtask require a proper completion by a transactional, non-preliminary terminative subtask.

The completion task, which starts in a progression point and ends in the original decision point, can be modeled in two ways as shown in Figure 34b and Figure 34c. The advantage of the solution presented in Figure 34b is that it gives smaller task graphs. However, the computation time of a completion task may be dependent on its preceding subtask. Therefore, it might be easier to represent the task graph as shown in Figure 34c, such that the analysis is easier to derive from the graph.

Note that the completion of a preliminary terminative subtask is performed by a non-preliminary terminative subtask, and the computation time of the completion task will appear as *termination latency*. This does not mean that this completion task is non-preemptable. A completion task is a task which is not allowed to preliminary terminate and is scheduled normally to regulate the progress of an application, optionally in conjunction with resource management protocols.

A.2.2 FINETUNING GRANULARITY OF TERMINATION

There are situations in which it is an advantage to defer termination until a predefined moment in time using preferred termination points, as defined in Section 3.3. For example, the write-behind approach might cause additional control overhead. By allowing a small amount of work to complete before terminating the pending job, it might be possible to transform the write-behind approach to a write-through approach.

Another consideration to defer termination of a job, is that immediate termination causes the waste of invested computational resources, which coincides with the work-preserving approach discussed in Section 2.3. It is conceivable to give a job some extra budget to complete, and reduce the capacity of the next occurrence of the task as a pay-back mechanism.

In order to allow deferred termination of a job, the polling and signalling approach can be combined. When a signal occurs to terminate a pending job, the event handler sets a local flag and returns executing the job. In case of the budget polling approach, which is considered expensive, it is assumed that the available budget for a job is not available locally in a task. Assuming that polling of a local flag is cheaper, signalling in combination with local polling gains in performance compared to the pure budget polling approach. As can be extracted from measurement results from the priority processing application, on general purpose machines the overhead for polling a flag is relative low. Therefore, polling within preferred termination points for a mode change request, seems a straightforward implementation.

The backside of deferred termination of a job is the increased reaction latency (reduced promptness). The polling mechanism allows the programmer to trade-off reaction latency versus computational overhead. Deferring termination to a predefined moment in time is similar to the approach taken by Holenderski et al. [19].

B. STATE CONSISTENCY USING FAULT TOLERANCE MECHANISMS

First of all, note that the terminology of fault tolerance might be a little confusing in the context of multimedia applications. Fault tolerance patterns describe ways to handle mistakes occurring in software in a structured way. In real-time multimedia applications it is not about dealing with software faults. Partial completed tasks might give undesirable effects, but these computations are not wrong. An example of undesired effects in multimedia applications is the appearance of artifacts in the output. In order to prevent these effects, similar approaches as used for obtaining fault tolerance can be used.

One of the arising problems when preliminary terminating a job is data inconsistency. Also during the execution of a mode change (see Appendix A) data inconsistency is an issue due to possible interference of different task sets. In literature several fault tolerant tactics are described [16].

Xu et al. [40] present a layered, object oriented architecture using coordinated atomic actions in order to ensure data consistency in fault tolerant environments. A coordinated atomic action is defined as [40]:

> “a multi-threaded transactional mechanism which as well as coordinating multi-threaded interactions ensures consistent access to (external) objects in the presence of concurrent and potential faults.“

The concept of coordinated atomic actions is strongly based upon the system structure concepts, as presented by Randell [30]. In [30], the basic concepts of recovery blocks, interprocess communication and fault tolerant interfaces are discussed as well as the influence of these concepts on the program flow. In order to avoid confusion about the atomicity of actions in the field of real-time systems, this report will use the term recovery block. An overview of the recovery block concept is given in Appendix B.3.

Furthermore, Xu et al. [40] introduce two types of redundancy:

1. **Dynamic Redundancy:** Several redundant components are capable of replacing each other when a fault occurs. Only one of these components is active at a time. External objects of the components might be checkpointed for the purpose of action recovery.
2. **Masking Redundancy:** Several concurrent actions are executed. An adjudicator component determines which action produces the correct output. This additional software component masks undesired or fault results from the environment. In case all actions fail, external objects are left unchanged.

Note that mode changes on itself can be seen as an instantiation of dynamic redundancy. A component can run in different modes, represented by different (sub-)tasks, aiming for the same ultimate goal. It is more interesting to match the concept of masking redundancy. Only one action can be performed at a time. The external behavior of this single action must be masked from other tasks in the system in case of preliminary termination.

B.1 RECOVERY PREPARATION

General patterns preparing for redundancy needs are checkpointing and data reset [15]. A checkpoint periodically saves state information of an application. When no checkpoint is available or not enough information is available to reconstruct the state of the checkpoint, the system has to be reset to its initial state. The information to be saved in a checkpoint incorporates all information that is of interest to the system as a whole. This information must preferably be saved in a centrally available location. This helps to reduce the time until a next task starts its execution, by limiting the time needed for recovery.

The state of a system might change while creating a checkpoint. Therefore dynamic and static checkpoints are distinguished:

Dynamic checkpoints are created by the independent processes or tasks when they appear to be valuable.
Static checkpoints are only created at the start of an execution sequence.

In the domain of video processing algorithms, the memory requirements for checkpointing can be large. A straightforward way of applying checkpoints is to create a checkpoint before a component starts to alter shared data. Inside the component, each subtask must ensure data consistency by only applying incremental updates.

B.2 FAULT DETECTION AND RECOVERY

A mode change request causes the running task set to change. This might cause unexpected termination of (sub-)tasks, which on its turn can consequently cause data corruption. The mechanisms described in this section are introduced to prevent data corruption. These mechanisms are described in the literature as fault recovery mechanisms [15].

The following actions can take place upon a fault detection, or a preliminary termination request in our context:

1. **Roll-back:** A checkpoint which represents a state of the system before the error occurred is restored. The program must resume with a different strategy than the one causing the error, [15, Section 32].
   For example: in priority processing a task may receive a request to preliminary terminate its pending activities. When such a request arrives, some pending work can be discarded and the results of processing a frame until that moment in time are output. By discarding a piece of pending work, a roll-back action takes place.
2. **Roll-forward:** In some situations it is known a priori that returning to a previous checkpoint will lead to the same error again or will result in incomplete computations. For example when there is not enough time to perform a recomputation. In this case it might be useful to progress in advance to a next synchronization point by skipping a certain computation, [15, Section 33].
   Although the preliminary termination request of a job in priority processed multimedia applications is described above to be associated with a roll-back action, it is more natural to associate the application behavior after an abortion request with a roll-forward action. By preliminary terminating a job, the application jumps forward to the next frame by skipping the remaining work of the current frame.
3. **Continue:** There might be situations in which a fault is not very harmful or has limited impact on the actual result. In these situation it is conceivable to continue processing without taking any action.
   In the domain of priority processing, it might be a cheaper alternative, compared to the actions involved of immediate job termination, to finish processing of a small piece of work before actually terminating a task.

B.3 RECOVERY BLOCKS

Randell [30] describes mechanisms of structuring software systems to gain fault tolerance. The basic mechanism, presented in this article are recovery blocks. A recovery block is defined as a conventional block of sequential basic operations, which is provided with an acceptance test and zero or more additional alternative execution blocks. The acceptance test is a condition which can be evaluated without side-effect in order to check whether the executed alternative has performed acceptably. If the acceptance test fails, the next alternative is executed. Before the alternative block is entered, the state of the process is restored as it was before the recovery block. When the last alternative fails, the recovery block falls back to the enclosing block. When the outermost block fails to execute all its alternatives, an error occurs in the system.

The recovery block scheme is not dependent on a particular block structuring. However, to prevent for improper nesting of blocks, it is required that:

1. the acts of entering and leaving an operation are explicit
2. operations are nested properly in time

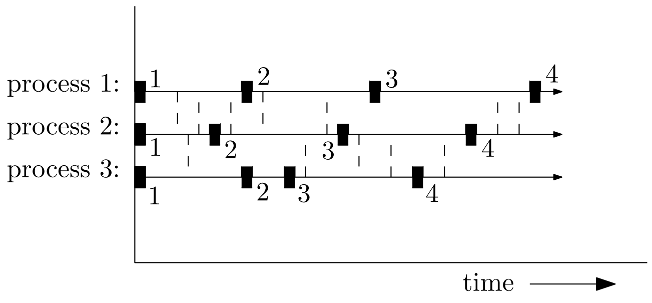
Both requirements are fulfilled automatically in high level programming languages, because the compiler takes responsibility for generating proper low level code.

Furthermore, considerable advantage can be taken of information which is provided indicating whether any given variable is local to particular operation. When an acceptance test is being evaluated, any non-local variables that have been modified must be available in their original as well as their modified form, because of the possible need to reset the system state. Note that the idea of saving the system state before entering a recovery block in order to restore this state in case the execution of a blocks fails, is equivalent to the patterns currently described in literature as creating checkpoints with the ability of rollback, see Appendix B.1 and Appendix B.2.

In order to allow restoring of the system state, Randell [30] proposes a recursive cache. This cache is divided in regions, in which each region is responsible for a nested recovery level. The entries in the cache contain the prior values of any variables that have been modified within the current recovery block. This provides the required information to roll-back the system state in case an error occurs.

Until now, only the progress of a single process composed of basic operations is considered. Example of basic operations are assignments and function calls, which can be a nested recovery block by itself. Extensions on the recovery block scheme described by Randell [30] are:

1. **Interacting processes:** When two or more processes start communicating with each other, all these processes must satisfy their respective acceptance tests and none may proceed until all have done so. If one process fails, all processes involved in the conversation must be backed up to the restoration point before the communication started. New communications can be initiated within a communication, however it is not allowed to have intersecting communications. This means that communication blocks must be strictly nested.



• *Figure 35: Domino effect caused by improper structuring of communication between processes. Communication between processes is indicated with dotted lines, whereas the black squares indicate restoration points. When process 2 fails, it is restored to recovery point 4. This means that process 1 needs to be restored to restoration point 3. Finally, all processes must be restored to its initial restoration point.*

   By requiring a structured way of communication between processes, the domino effect can be prevented. The domino effect occurs when a single process has to be restored due to a failure, causing other processes also to consume a restoration point. The worst-case scenario of the domino effect in which all restoration points are consumed is shown in Figure 35.
   In real-time streaming applications the domino effect is prevented, because these applications only perform local computations. When data is written to external memory, this is considered as a resource allocation. Resource allocation protocols prevent the domino effect by disallowing shared access to a resource.
2. **Multi-level systems:** Facilities for providing fault tolerance are also extended to deal with layered systems. In a layered system, each layer is an abstraction of the layer below it. Furthermore, each layer provides an interface to upper layers. In order to reduce complexity in handling fault tolerance, the understanding of the system designer is limited to a single layer. In this model, two types of faults must be dealt with:
   (a) **Errors above an interface:** Assume that an application layer, l_i, is implemented using recovery blocks. The layer above, l_{i-1} is responsible for all state changes made by layer l_i, as well as saving and restoring the effects caused by the progress of layer l_i. This assumes that layer l_{i-1} is fault free. Now it is investigated how to deal with errors below an layer-interface.
   (b) **Errors below an interface:** Again it is assumed that the layers are implemented using the recovery block scheme. Now assume that all alternatives in layer l_{i-1} fail to perform their operations on the layer below, l_i. This means that a recovery action must be performed in layer l_i. This is almost the same as the scenario described above. Although, there might be a scenario in which turning to the recovery point at level l_i fails. In general, this means that layer l_{i-1} has to abandon all operations of layer l_i. However, some errors occurring at layer l_i can be dealt with without abandoning this layer. After layer l_i has passed an acceptance test, but before all the information constituting its recovery point has been discarded, layer l_{i-1} might perform a check on the overall layer l_{i-1} acceptance.

Note that dealing with faults inside interfaces is an instantiation of providing fault tolerance for interacting processes.

C. SYSTEM BENCHMARKING ON X86 PLATFORMS

Measurement data of the video processing applications investigated in this report are obtained by collecting benchmarking results. These simulations run on general purpose X86 machines. Since we are interested in time measurements in the range of milliseconds for frame computation times and even micro seconds when measuring termination latencies, standard benchmarking methods using timer data are not sufficient. These timer data are provided by the operating system via system calls, which can consume milliseconds on itself.

With the introduction of the Pentium series, Intel specified a registered which counts the cycles consumed by the processor since it started [1]. This register, the *Time Stamp Counter* (TSC), is available on all X86 processors, accessible via an instruction called Read Time Stamp Counter (RDTSC). This register can be read via assembly instructions and therefore is a lot cheaper than regular operating system API calls. However the disadvantage is that the way to read this register can differ per platform. The operating system and the width of the registers, for example 32 bit versus 64 bit, influences the way of accessing the register. A library for measuring the consumed cycles in a specified interval is provided by Frigo [13]. This library implements an interface to overcome the platform dependent difficulties.

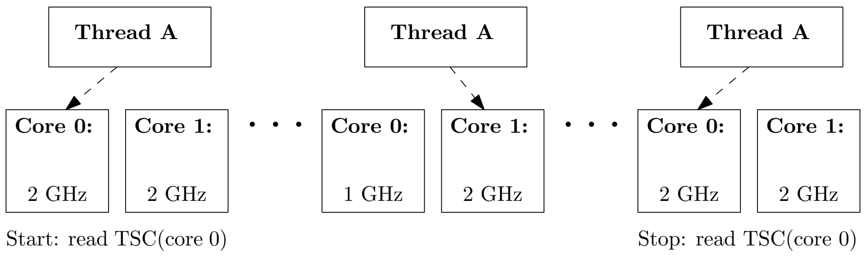
The TSC register contains a value representing all consumed CPU cycles since the start-up or reset of the core. Human readable measurements are preferably presented in time units in stead of consumed cycles. Meanwhile, the value stored in the Time Stap Counter can be affected by several mechanisms:

1. CPU frequency scaling: Todays processors provide the possibility to adapt the frequency to save power. When running at a lower speed, less cycles are consumed.
2. When the processor is turned into deep sleep mode, no cycles are consumed.
3. Different processor series update the time stap counter on different intervals. In early designs the counter is updated every clock cycle. Nowadays, it is tended to use a factor dependent on the front-side-bus and CPU speed.

Considering the above characteristics of the time stap counter, it is noticeable that in order to present measurements in time units, the average processor speed during the interval must be known. In case these values are known, the measured time interval can be calculated as follows:

$$t_i = \frac{T_{tsc}}{f_{avg}} \tag{15}$$

In Equation 15, t_i is the time interval in seconds, T_{tsc} is the difference of the TSC register value in the time interval and f_{avg} is the corresponding average frequency. Tian et al. [37] use similar methods to synchronize relative clocks over the network and claim to read values up to nanosecond precision. The synchronized, relative time values are precise in the order of 10 μs. It must be noted that the experimental test configuration of Tian et al. [37] contains fairly old hardware.



• *Figure 36: When a thread is assigned different cores during its life-span benchmarking results might be wrong, even if the start and stop values are gathered from the same core.*

A new problem arises with the *Symmetric Multi-core Processing (SMP)* systems. Every core has its own TSC register. Now consider the following scenario: When a thread is started the RDTSC instruction is executed. While running, the scheduler of the operating system can decide to stop a thread, t_A, temporarily in favor of an other thread. This leaves thread t_A in ready state, waiting for available processor time proceed processing. When a core is free, the scheduler can decide to run thread t_A, but not necessarily on the same core. So assume that the thread is assigned to another core. Now the thread is stopped and we read again the TSC register. Note that both counters are not necessarily synchronized, because both cores can be scaled separately. Therefore, the measured time in cycles does not represent a valid value. The measured time can even be negative.

Note that it is possible to detect on which core the thread runs, but this does not solve the problem. The operating system can decide to swap the thread to another core multiple times. While the begin and end times are measured on the same core, it is possible that this value is not valid, because the core can have an intermediate time of a down-scaled frequency or even sleep mode. This behavior is shown in Figure 36

In the priority processing application, only short and highly computational worker threads at high priority are used. Therefore, it is assumed that during this interval the processor runs at full speed. Operating systems often give the possibility to manually disable frequency scaling options, such that the frequency is fixed. Since the priority processing application uses only short worker thread, with a life-span of tens of milliseconds, it is assumed that the thread runs entirely on the same core. In practice this gives good results, however occasionally peaky time measurements can be observed which are not explainable. These can be relative high peaks, but also negative peaks. These peaky measurements can easily be filtered from the experimental data.

D. POSIX THREADS AND SIGNALS

The IEEE 1003.1 standards [20] defines a Portable Operating System Interface for Unix (POSIX), see [20]. In IEEE and Group [20, Section 2.4] signal concepts are defined. A signal is generated by an event, which can be a hardware fault, timer expiration or a software interrupt. A signal can be send to a thread, a process or a process group. Each thread can accept or block particular signals, whereas each signal has a default action. When a signal is received by a thread, it can take the default action, ignore the signal or perform a custom action which is done by a *signal handler*.

When a non-blocked signal is send to process while the process is executing a function, the signal can effect the behavior of this function. When the signal causes the process to terminate, the function will not return. When a signal handler is defined to take action upon receiving a signal, the interrupted function will continue after completion of the signal handler.

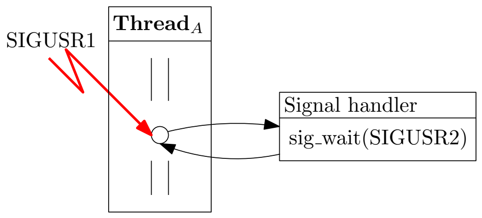
D.1 LINUX SIGNALS

Thangaraju [36] describes the usage of Linux threads from the perspective of the application programmer. Like described by McCracken [25], the Linux signal handling differs in some ways from the POSIX standards. The standards describe the handling of signals from the process perspective, whereas the Linux kernel provides signal handling per thread basis. It must be noted that all signals are first delivered to the process before actually being delivered to the correct thread.

Each thread inherits the signal mask from its parent process. The signal mask defines the set of blocked signals. This implies that once a signal is blocked by the process, it can not be caught by a child thread. When a signal can not be delivered to the correct thread it is handled by the corresponding Linux process, causing it to terminate as a default action. When sending signals at very high rates, like in multimedia applications, it will occasionally happen that a signal causes the main process to terminate unexpected. Since the standard Linux kernel is not a real-time kernel, it is not guaranteed when the signal is handled. This can cause unresponsiveness when a lot of signals are sent to a process.

D.2 SUSPENDING AND RESUMING POSIX THREADS

As a mechanism to divide the available resources among competing tasks, suspending and resuming of tasks have been descibed in Section 3.5.2. The POSIX API does not provide a standard way to allow suspending and resuming of threads (we assume implicitely a direct mapping of tasks to threads). Jenks et al. [21] describe a way to use Linux signals to allow suspending and resuming of threads. Figure 37 shows how the suspending and resuming of threads is achieved.



• *Figure 37: When a user defined signal $SIGUSR1$ arrives, the signal handler is executed which blocks the thread until a user defined resume signal $SIGUSR2$ arrives. The resume signal annulates the blocking state of the signal handler, such that the thread continues its operation.*

First, it is assumed that a thread is following its regular execution path. The thread is associated with a signal handler for a user defined signal, $SIGUSR1$. When this signal comes in, the signal handler is executed. This signal handler on its turn blocks until another user defined signal, $SIGUSR2$ arrives. Since after the successful return of the signal handler the thread will proceed its execution where it left before, this $SIGUSR2$ signal releases the thread from the blocked state, or in other words: resumes the thread.

The problem with applying this method in the field of video applications is the huge amount of signals that are needed to control the application. For example in the priority processing application, the control divides time-slices among competing algorithms. These time-slices are in the order of 1 ms. This means that every millisecond multiple signals must be handled by the thread, which causes to much overhead. As an alternative, the use of priority bands can be considered as described in Section 3.5.3.

E. PLATFORM SPECIFICATIONS

Measurement data for the priority processing applications are obtained by running simulations on the following platforms:

1. Gentoo GNU/Linux on x86_64
   - kernel 2.6.25-gentoo-r6
   - Intel Core 2 T7200, 2.00 GHz GenuineIntel
   - 2 GB RAM
   - The MathWorks Matlab 7.5.0.338 (R2007b), August 7^{th} 2007
   - Simulink 7.0 (R2007b), August 2^{nd} 2007
   - GNU Compiler Collection (GCC) 4.1.2 (Gentoo 4.1.2 p1.1)
2. Microsoft Windows XP Professional Service Pack 3
   - Intel Xeon E5345, 2.33 GHz GenuineIntel
   - 3 GB RAM
   - The MathWorks Matlab 7.5.0.338 (R2007b), August 7^{th} 2007
   - Simulink 7.0 (R2007b), August 2^{nd} 2007
   - Microsoft Visual Studio 2005 C++ Compiler