

Dynamic Resource Allocation for Competing Priority Processing Algorithms on a Cell/B.E. Platform

Coen Tempelaars, BSc

May 11, 2010

Master Thesis

Eindhoven University of Technology
Department of Mathematics and Computer Science
Chair of Systems Architecture and Networking

In cooperation with:
Brandenburg University of Technology Cottbus
Department of Mechanical, Electrical and Industrial Engineering
Chair of Media Technology

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

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

Tutor:
Dr. ir. B. Mesman
Researcher
b.mesman@tue.nl

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

Abstract

Scalable video algorithms (SVAs) allow a trade-off between the output quality and resource usage. The priority processing principle provides full utilization of available resources without violating real-time requirements. Even on platforms with limited system resources, this approach results in the best possible output quality given the resources used. This report shows how real-time performance for priority processing algorithms can be achieved on the Cell platform.

When several independent priority processing algorithms run on a shared platform, the algorithms start competing for resources. During earlier research, a decision scheduler has been developed to distribute the available resources among the priority processing algorithms. This is done by dividing the available time in fixed-size time slots, that are allocated to the algorithms. The goal of the decision scheduler is to maximize the overall output quality. Different strategies for reaching this goal have been implemented, including an adaptive strategy based on the concept of reinforcement learning.

During earlier research, three mechanisms for dynamic allocation of resources by the decision scheduler have been identified and different implementations of these mechanisms on a general purpose platform have been compared in a quantitative manner. These mechanisms are (1) preliminary termination, (2) resource allocation and (3) monitoring.

The Cell processor offers a general purpose core as well as several synergistic processing elements (SPEs), which are special purpose cores capable of performing single-instruction-multiple-data operations. Such a processor offers great advantages in various fields, one of them being signal processing. This report focuses on a setup in which the decision scheduler runs on the general purpose core and two priority processing algorithms share one of the SPEs. Using this setup, different implementations of the mechanisms for dynamic resource allocation are compared using quantitative methods. In this report, all comparisons focus on computational overhead and latency.

Preliminary termination is a mechanism used for preempting or terminating all algorithms that are executed in a video processing system. We compare cooperative termination with termination by means of signalling. Cooperative termination relies on a

mailbox that is periodically polled, whereas signalling relies on mechanisms for software interrupts, that are provided by the hardware platform or the operating system.

Resource allocation indicates the mechanism used for allocating a time slot to one of the competing algorithms, i.e. either allowing an algorithm to continue or performing a context switch. Since there is no operating system running on an SPE, typical mechanisms offered by an operating system, such as suspending/resuming a task or manipulating tasks' priorities cannot be used. This report describes one method to perform a context switch on an SPE and presents the progress achieved by two priority processing algorithms sharing one SPE.

The monitoring mechanism indicates how the achieved progress, as well as the used resources, for each of the competing priority processing algorithms are accumulated. This report presents one implementation of the monitoring mechanism on the Cell platform and analyses the accuracy of the implementation in a quantitative way.

Acknowledgements

This thesis is written as part of the master project that I have performed at the chair of Media Technology at Brandenburg University of Technology in Cottbus, Germany, in cooperation with the chair of System Architecture and Networking at Eindhoven University of Technology, the Netherlands. Upon successfully finishing the master project comes the conclusion of my study Computer Science and Engineering at Eindhoven University of Technology.

I have enjoyed working on this project, mainly because of the combination of implementing a new technology on an interesting platform and carrying out experiments upon the implementation. Therefore, I express my gratitude to Stefan Schiemenz and prof. Christian Hentschel for initiating this project and giving magnificent support along the route. Furthermore, I thank all the folks at the chairs of Media Technology and Communication Technology for the pleasant working environment, as well as social environment.

Second of all, I would like to thank Reinder Bril and Martijn van den Heuvel very much for supervising my work from Eindhoven, for providing great advice and giving numerous amounts of new insights. Furthermore, I would like to thank Bart Mesman, Asan Shabbir and Kris Hoogendoorn for their useful advice on programming the Cell processor. Finally, a big thanks goes to friend and fellow student Rom van Arendonk for proofreading my thesis.

As there is time for studying and working, a student should also spend time on social activities. I would like to thank my friends at fraternity 'Demos' for providing these kind of activities, unforgettable moments and a great environment for social development and I thank my friends at sailing union 'Boreas' for the nice moments of relaxation.

Last, but absolutely not the least, I would like to thank my family, parents and brothers for their great support and encouragement throughout my study years.

Contents

1	Introduction	1
1.1	Context and Background	1
1.2	Problem Description	3
1.3	Project Goals	3
1.4	Approach	4
1.5	Contributions	4
1.6	Overview	5
1.7	Glossary	6
2	Competing Priority Processing Algorithms	7
2.1	Scalable Video Algorithms	7
2.2	Priority Processing	8
2.3	Decision Scheduler	11
2.4	Dynamic Resource Allocation	12
2.4.1	Preliminary Termination	12
2.4.2	Resource Allocation	13
2.4.3	Monitoring	13
3	Overview of the Cell Processor	14
3.1	Main Constituents	14
3.2	Data Transfer	16
3.2.1	Direct Memory Access	16
3.2.2	Memory Access with Request Lists	17
3.2.3	Double Buffering	18
3.3	Communication Mechanisms	19
3.3.1	Mailboxes	19
3.3.2	Signal Notification Channels	20
3.4	Mechanisms for Measuring Time	20
4	Porting the Application	22
4.1	Assumptions and Boundary Conditions	22
4.2	Decomposition of the Scalable De-interlacer	23

4.2.1	Basic Function	24
4.2.2	Motion Detection	24
4.2.3	Noise Estimation and Block Ordering	25
4.2.4	Image Enhancement	25
4.3	Porting of the Scalable De-interlacer	25
4.3.1	Basic Function	25
4.3.2	Motion Detection	26
4.3.3	Noise Estimation and Block Ordering	26
4.3.4	Motion Adaptive Interpolation	27
4.3.5	Edge Dependent Motion Adaptive Interpolation	28
4.4	Efficient Implementation on the SPE	28
4.4.1	Single Instruction Multiple Data	28
4.4.2	Avoiding Branches	30
5	Implementation of Dynamic Resource Allocation Mechanisms	31
5.1	Implementation of Communication	31
5.1.1	Communication between PowerPC and SPE	32
5.1.2	Communication Safety	33
5.2	Preliminary Termination	33
5.2.1	Polling on the SPE	34
5.2.2	Interrupt Handling on the SPE	35
5.3	Resource Allocation	35
5.3.1	Context Switching	37
5.4	Monitoring	38
6	Evaluation	42
6.1	Research Questions	43
6.1.1	Preliminary Termination	43
6.1.2	Resource Allocation	43
6.2	Hypotheses	44
6.2.1	Preliminary Termination	44
6.2.2	Resource Allocation	44
6.3	Preliminary Termination Latency	45
6.3.1	Communication Latency	45
6.3.2	Termination Latency	47
6.4	Preliminary Termination Overhead	54
6.4.1	Absolute Overhead	54
6.4.2	Relative Overhead	54
6.5	Resource Allocation	61
6.5.1	Decision Scheduler Latency	61
6.5.2	Latency due to Context Switching	62
6.6	Monitoring	64
7	Conclusions	65

7.1	Future Work	66
7.1.1	Application Level	66
7.1.2	System Level	67
A	Commonly used SPE Instructions	72
A.1	The <i>spu_splats</i> Function	72
A.2	The <i>spu_sel</i> Instruction	72
B	Branch Intensive Code	73
C	Resource Management on the SPE	76
D	Latency due to Context Switching	78
E	The PlayStation 3	81

Chapter 1

Introduction

Priority processing algorithms guarantee real-time processing of video, while maximizing output quality given available resources. This report focuses on the implementation and control of such priority processing algorithms on the Cell Broadband Engine processor [1, 2]. This processor, called ‘the Cell’ for short, is a multi-core processor developed by an alliance of Sony, Toshiba and IBM. Different implementations for the mechanisms for dynamic resource allocation on the Cell platform are investigated, with the goal to maximize the overall output quality by several priority processing algorithms.

This chapter will start with a brief introduction into the domain of this thesis in Section 1.1. A concise description of the problem is stated in Section 1.2. Section 1.3 explains the goals of the project, whereas Section 1.4 states the approach taken towards meeting these goals. The contributions of the project are listed in Section 1.5. An overview over the entire report is given in Section 1.6 and a list of the abbreviations used throughout the report is given in Section 1.7.

1.1 Context and Background

Multimedia systems are systems that process audio, video and 3D streams in order to present the processed streams to the user. Rather than processing a video stream in hardware, a trend can be observed to process these streams in software. The main reasons for this shift towards processing in software are openness and flexibility [3]. A multimedia system should be flexible enough to display one video stream at a certain quality and to display multiple video streams at the same time. This flexibility can be achieved by reducing the quality when multiple video streams have to be processed. Multimedia systems based on dedicated hardware tend to be less flexible than systems that do video processing in software.

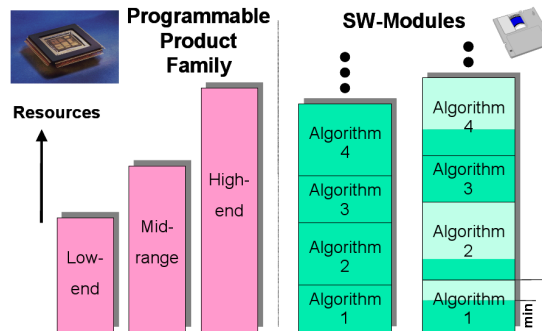


Figure 1.1: Schematic view on the amount of resources offered by a range of programmable products compared to the resource demands of fixed and scalable video algorithms. Dark green areas indicate non-scalable components, whereas light green areas indicate scalable components of an SVA. Source: [4]

Traditional video processing algorithms usually deliver output with a fixed quality level. An alternative approach for processing video streams in software is by using scalable video algorithms (SVAs). The differences in resource usage of these two approaches are shown schematically in the right part of Figure 1.1. Scalable video algorithms allow a trade-off between output quality and resource usage [5]. This results in various advantages, one of them being the possibility to deploy the same set of algorithms on different products that offer different amounts of resources.

Multimedia systems belong to the class of real-time systems, i.e. systems in which stringent timing constraints must be met [6]. In the case of video algorithms, the deadline is set at the end of a frame period, i.e. a frame should be delivered to the output device. Fluctuations in the rate at which frames are delivered are most likely observed by the viewer and result in a reduced quality of service. Meeting such a deadline can become problematic when several video processing algorithms are time-sharing the CPU.

Hentschel and Schiemenz [7] have proposed an approach, called *priority processing*, that guarantees full utilization of available resources by SVAs, without violating the real-time requirements of multimedia systems. The structure of a priority processing algorithm is as follows. At the start of a frame period, a non-scalable video algorithm computes a basic, low-quality output. This output is divided into blocks, which are analyzed and put into a priority-queue. The blocks in the queue are ordered by decreasing quality improvement. The contents of these blocks is then refined one block at a time, until the deadline is reached.

When multiple independent priority processing algorithms share a common platform with limited resources, these algorithms start to compete for resources. During earlier research, a decision scheduler (DS) [8] has been developed on a general purpose processor, with the task to distribute available resources over the competing priority processing

algorithms. This is done by dividing one frame period into fixed-size time slots and allocating every time slot to one algorithm, according to a predefined strategy. The goal of the decision scheduler is to maximize the overall quality of the output. The DS relies therefore on the priority processing algorithms to accurately report the achieved progress.

Van den Heuvel et al. [9] have identified three mechanisms for dynamic resource allocation to multiple priority processing algorithms, namely (1) *preliminary termination*, (2) *resource allocation* and (3) *monitoring*. For these three mechanisms, different implementations have been compared on a general purpose platform with respect to latency and computational overhead. The mechanisms will be covered in detail in the next chapter.

1.2 Problem Description

Currently, it is not possible to execute priority processing algorithms with real-time requirements on a general purpose platform. In this thesis, it is investigated whether priority processing algorithms can run on a Cell platform, while respecting real-time requirements.

The combination of a decision scheduler and priority processing algorithms shows a clear distinction between video processing and control. This combination suits itself very well to be mapped on a heterogenous platform. The Cell platform [1, 2] promises to be a good choice, because the Cell processor contains several synergistic processing elements (SPEs) besides its general purpose core. These SPEs are capable of performing single-instruction-multiple-data (SIMD) operations, hence it is possible to process multiple pixels, independent of each other, in parallel. This makes an SPE a good processor in the domain of video processing.

Since there is no operating system running on these SPEs, dynamic resource allocation mechanisms can only rely on mechanisms offered by the hardware platform, not on those that are typically offered by an operating system on a general purpose processor.

1.3 Project Goals

The goal of this report and the entire project is to show real-time performance of an application consisting of the decision scheduler and one or more priority processing algorithms on the Cell platform. The focus should be on the implementation of the mechanisms for dynamic resource allocation. Different implementations of these mechanisms should be compared using quantitative methods. All comparisons should focus on *latency* on the one hand and on *computational overhead* on the other hand.

By focussing on the implementation of mechanisms for dynamic resource allocation on the Cell platform, this report fits in a broader perspective concerning the implementation of priority processing algorithms with real-time requirements on a platform that is feasible for consumer electronics, of which the Cell is an example.

1.4 Approach

The approach towards achieving the goals as stated in the section before, can be subdivided into a number of steps that have to be executed. Before any analysis can be done, at least one scalable video algorithm that uses the priority processing principle has to be implemented on the Cell architecture. Upon successful porting of the algorithm, different strategies for the preliminary termination mechanism must be implemented and compared using quantitative analysis. These strategies are all based on efficient communication between the PowerPC and SPE cores.

The results from the analysis can be used in the second implementation step, in which a mechanism is implemented that allows for context switching of multiple SVAs on one SPE. The most efficient strategy for communication between the cores on the Cell will be chosen for initiating a context switch. Next, the decision scheduler can be implemented and the different strategies for resource allocation and monitoring must be identified and analyzed in a quantitative way.

1.5 Contributions

As a result of the implementation of the decision scheduler controlling several priority processing algorithms on the Cell, as well as the implementation and evaluation of the mechanisms for dynamic resource allocation, the following contributions have been made:

- Priority processing algorithms have shown to run with real-time requirements on the Cell platform.
- Different implementation variants for communication on the Cell platform between different cores have been compared. An implementation using software interrupts has shown to be both reliable and efficient in terms of latency and computational overhead.
- It has been shown that an efficient implementation for all three mechanisms for dynamic resource allocation rely on an efficient implementation communication between different cores on the Cell.

- A method to perform a context switch on a processor that does not run an operating system has been described and has been implemented on an SPE.
- A mechanism for allocation of time slots to priority processing algorithms has been described and analyzed in a quantitative way.
- A mechanism for monitoring of achieved progress and elapsed time has been described and analyzed, also in a quantitative way.

1.6 Overview

In Chapter 2, the different aspects in the domain of this thesis are explained. A lot of previous work has been done in the field of scalable video algorithms and priority processing, which is prerequisite knowledge for the remaining chapters.

Every platform comes with its advantages and disadvantages. The potential and the limits of the Cell platform will be clarified in Chapter 3.

Chapter 4 describes the source application on the one hand and implementation choices that have been made during the process of porting this application to the Cell on the other hand.

The mechanisms for dynamic resource allocation are the subject of this report. Chapter 5 describes the implementation of these mechanisms and presents a comparison of these implementations. The evaluation of the implementation of the mechanisms for dynamic resource allocation can be found in Chapter 6.

Finally, Chapter 7 presents the conclusions that are drawn from this work and presents possible future work.

1.7 Glossary

Throughout this document, several abbreviations are used. These are grouped below for convenience.

Notation	Description
Cell/B.E.	Cell Broadband Engine
CPU	Central Processing Unit
DMA	Direct Memory Access
DS	Decision Scheduler
EIB	Element Interconnect Bus
GPP	General Purpose Processor
ISR	Interrupt Service Routine
KiB	Kibibyte, i.e. 1024 bytes [10]
LCD	Liquid Crystal Display
LS	Local Store
MFC	Memory Flow Controller
PiP	Picture-in-Picture
PP	Priority Processing
PPE	PowerPC Processing Element
PPU	PowerPC Processing Unit
RTOS	Real-time Operating System
SIMD	Single-instruction-multiple-data
SPE	Synergistic Processing Element
SPU	Synergistic Processing Unit
SVA	Scalable Video Algorithm
VQEG	Video Quality Experts Group
WCET	Worst-case Execution Time

Chapter 2

Competing Priority Processing Algorithms

Chapter 1 has superficially described the scope of this thesis. In this chapter, the scope is discussed more extensively. The properties of scalable video algorithms are explained in Section 2.1. This is followed by Section 2.2 on the priority processing principle, which is a concept that shows to be useful in the field of scalable video algorithms. When multiple priority processing algorithms are implemented on a shared platform, these algorithms start competing for resources. Section 2.3 explains a decision scheduler, which is responsible for allocating resources to the competing algorithms. This chapter concludes with a description of mechanisms for dynamic resource allocation in Section 2.4.

2.1 Scalable Video Algorithms

Signal processing algorithms, of which video processing algorithms are a subset, usually have real-time requirements. This report focuses on video processing algorithms for which both input and output are uncompressed video sequences consisting of consecutive frames. The processing time per frame is usually content dependent and the amount is usually limited, since the frames should be delivered to the output device at a predefined frame-rate.

If the worst-case processing time per frame is known for the combination of video processing algorithms, one could guarantee the desired frame rate by implementing the algorithms on a platform that always offers enough resources. But since the workload for video processing algorithms heavily depends on the contents of the input, this generally is an inefficient use of resources.

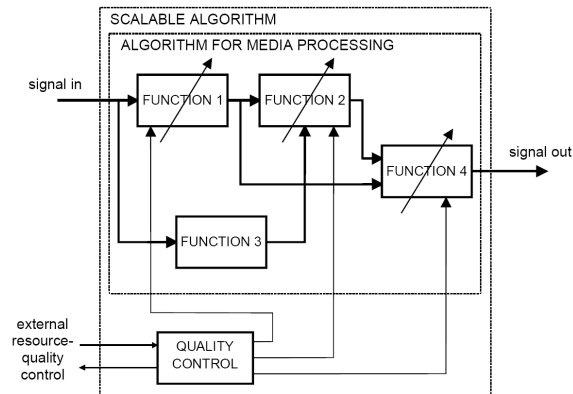


Figure 2.1: The basic structure of a scalable video algorithm. All components process video streams (thick lines), the scalable components also offer an interface for quality control (accessed through thin lines). Source: [3]

Furthermore, resource availability might change over time, because resources are usually shared between several algorithms. Fluctuations in the number of executing algorithms are generally caused by the environment, for example when the user enables picture-in-picture, or the video sequence has to be encoded and stored besides being displayed.

Scalable video algorithms have been introduced [5] to cope with these problems, because these SVAs have the property of being able to trade output quality for resource usage. The basic structure of such an algorithm is shown in Figure 2.1. Hentschel et al. [3] define a scalable video algorithm to have the following properties:

- It allows the dynamic adaptation of output quality versus resource usage on a given platform.
- It supports different platforms / product families for media processing.
- It is easily controllable by a control device for several predefined settings.

2.2 Priority Processing

In video processing systems, video frames should be delivered to the output device at some predefined frame-rate. This results in a fixed deadline per frame that has to be obeyed by the video processing algorithm. One way to ensure that a complete frame has been processed before the deadline has been reached, is to make sure that the worst-case execution time (WCET) for the video processing algorithm(s) is strictly less than one frame-period. This however, results in under-utilization of resources.

One method that allows video processing with limited resources is using a work-preserving approach, as described by Wüst et al [11]. Using this approach, the processing of the current frame is completed, even if the deadline has been exceeded. If a deadline miss occurs, the previous frame is repeated by the output device and processing of the next frame is skipped. Wüst et al. have implemented the quality control component to use a self-learning strategy, so that a scalable video algorithm produces output of maximal quality, whilst keeping the number of deadline misses to a minimum. A disadvantage of such a work-preserving approach is that there is a need for an extra buffer such that the previous frame is stored at any time. Introducing an extra buffer contributes to larger delays in the system. The control of a scalable video algorithm as described by Wüst et al. just reasons over one SVA, in contrast with the decision scheduler as described below.

A different method is to use the *milestone method* [12], which is an approach based on the *imprecise computation* technique [13]. Using this technique means decomposing a time-critical task into a mandatory subtask, which is executed first, and an optional subtask. The mandatory subtask consists of a non-scalable video algorithm that computes some low-quality output, ready to be transferred to the output device when necessary. During the optional subtask, the output is enhanced by a scalable video algorithm. This enhancement is done in a step-by-step fashion, with monotonically increasing quality, where the refined output is stored in the output buffer every time a predefined milestone is met. The number of pixels that can be enhanced depends on the contents of the frame and of the amount of available time left. As a result, the available resources are fully utilized, unless the entire frame can be enhanced before the end of the frame-period is reached.

Hentschel and Schiemenz [7] present an extension to this method, called *Priority Processing*. In their approach, the contents of the frame is analyzed between the two phases. During this analysis phase, the frame is divided in fixed-size blocks. For every block in the frame, a content-dependent priority is determined, which indicates how much output quality can be gained by enhancing this block. Subsequently, the blocks are ordered from highest quality increase to lowest. During the enhancement phase, the contents of as many blocks as possible is being refined, following the order as derived in the analysis phase. Both the analysis phase and the enhancement phase can be preliminary terminated when the deadline is reached.

The advantages of using the principle of priority processing are twofold [7]. On the one hand the available resources are always fully utilized, without violating real-time requirements. On the other hand, the output will be of the best possible quality given the resources used and the available processing time. When priority processing is compared to the work-preserving approach, a difference is that priority processing operates on a single output buffer, whereas there is a need for an extra buffer with the work-preserving approach.

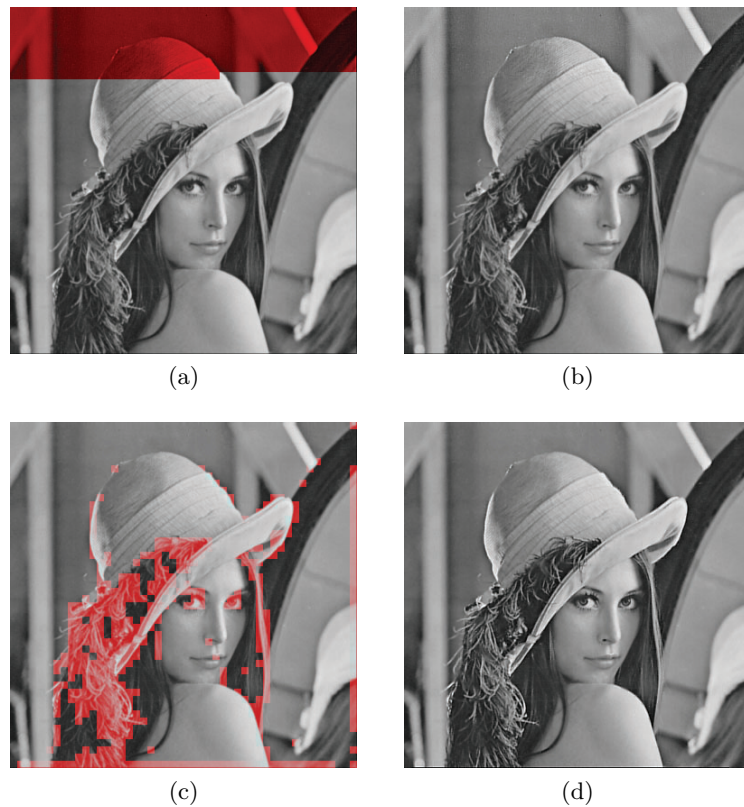


Figure 2.2: The output of a sharpness enhancement filter applied to a single frame. Figure (a) shows which blocks have been processed by the filter operating in a sequential way and (b) shows the corresponding output. Figure (c) shows the blocks that have been processed by a filter using a content-dependent priority order and (d) presents the corresponding output. Source: [7]

The result of using the priority processing principle has been visualized by applying a sharpness enhancement filter to a single frame in Figure 2.2. This figure shows twice the same frame in which 20% of the contents has been sharpened. The top-left frame shows which pixels have been processed in red for a sequential approach. The result of sequential sharpness enhancement is shown in the top-right frame. The bottom-left frame shows which pixels are processed with the priority processing approach, after a content-dependent priority order has been determined. The bottom-right frame shows the result of a sharpness enhancement filter with priority processing.

The progress of processing one frame over time shows a typical shape when priority processing is used. This shape is outlined in a simplified way in Figure 2.3. Notice how the three phases of execution are clearly visible. Throughout the remainder of this report, these phases will be called:

1. Basic output phase

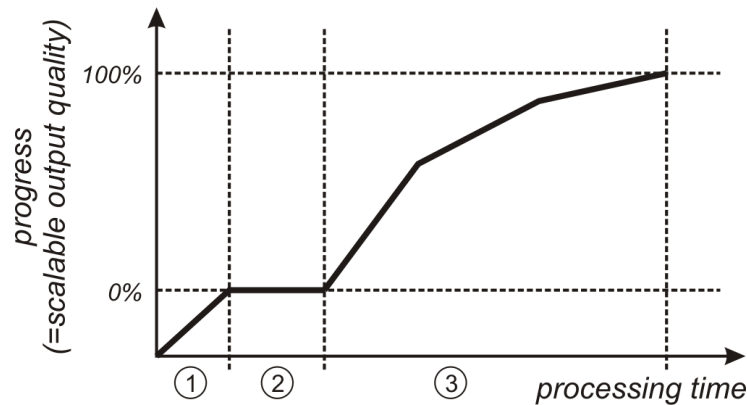


Figure 2.3: A general view upon the progress of a priority processing algorithm over time. The numbered phases are (1) basic output phase, (2) content-analysis phase and (3) image-enhancement phase. Source: [8]

2. Content-analysis phase

3. Image-enhancement phase

In Figure 2.3, the progress of the non-scalable basic function is defined to provide an output quality of 0%. Note that output quality is defined as the output quality as perceived by the user, hence it depends on the number of blocks processed. An output quality of 100% is reached when the entire frame has been enhanced by the scalable video algorithm.

2.3 Decision Scheduler

Whenever multiple priority processing algorithms share a common platform with limited resources, these algorithms start competing for resources. In earlier research, a decision scheduler has been created on a general purpose platform in a Matlab / Simulink environment. The purpose of the DS is to allocate the available amount of processing time within one frame period to the algorithms in such a way that the overall output quality is maximized. Therefore, a frame period is divided into a fixed number of equally sized time slots. The primary tasks of the decision scheduler are to keep track of elapsed time, to notify the priority processing algorithm(s) of reached time slot boundaries and frame deadlines and to decide which algorithm to allocate the next time slot to.

The DS decides which algorithm to allocate a time slot to, based upon the achieved progress per algorithm and on a certain strategy. Schiemenz [8] has implemented three different strategies, being (1) *round-robin*, (2) *minimum progress* and (3) *reinforcement learning*.

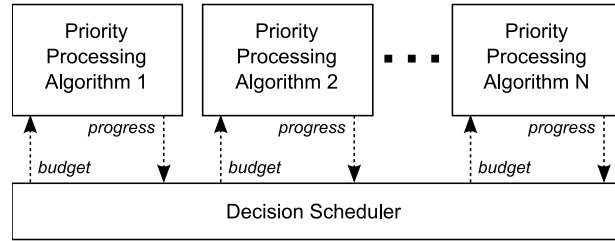


Figure 2.4: The interaction between the decision scheduler and the algorithms it controls. The DS grants a budget (resources) to the algorithms, whereas the algorithms are responsible for reporting the achieved progress to the DS. Source: [8]

If *round-robin* scheduling is used, the algorithms are assigned time slots in a cyclic manner, so the decision is made independent from the achieved progress by the SVAs. With the *minimum progress* strategy, the algorithm that has achieved the least total progress is allocated the next time slot.

The final strategy is based on Sutton and Barto’s concept of *reinforcement learning* [14]. With this strategy, the decision scheduler keeps track of the achieved progress, as reported by a priority processing algorithm at the end of a time slot. The goal of maximizing the overall output quality is pursued by maintaining a history of decisions made in combination with the resulting progress. Based on a probabilistic value, the DS takes either a decision based upon this table, or it takes an exploration step with the purpose to improve the history table.

2.4 Dynamic Resource Allocation

Van den Heuvel et al. [9] have identified three platform independent mechanisms that support the allocation of system resources whilst optimizing resource utilization. An efficient implementation of these mechanisms is necessary in a setup of a decision scheduler controlling several priority processing algorithms. The three mechanisms are described in the following subsections.

2.4.1 Preliminary Termination

To ensure that all running algorithms abort processing and give up the resources they have in use, the decision scheduler must be able to enforce preliminary termination of the algorithms it controls. There are two main approaches for preliminary termination, namely cooperative termination, by means of periodically polling a shared resource (e.g. a global variable or a dedicated communications channel) and termination by means of signalling.

2.4.2 Resource Allocation

The decision scheduler acts as an arbiter that allocates a time slot to one of the algorithms that are competing for resources. This allocation can either be done using mechanisms offered by the operating system (e.g. suspending and resuming a task or manipulating task priorities) or in a cooperative way if there is no operating system or these mechanisms are not provided. This report focuses on the latter, since there is no operating system running on an SPE. Therefore a mechanism for performing a context switch has to be written from scratch, which can only make use of mechanisms provided by the (hardware) platform itself.

2.4.3 Monitoring

The decision scheduler bases its decision to grant a time slot to a certain algorithm on the progress made by the different algorithms and the time slots that have already been granted to the algorithms. At the end of every time slot, the currently running algorithm has to measure its progress and the decision scheduler needs to be activated again. Accurate time slots are therefore required, as is a low-latency communication mechanism between the decision scheduler and the priority processing algorithms.

Chapter 3

Overview of the Cell Processor

Section 3.1 gives an overview of the Cell processor and presents the main components of the Cell with their specific properties. This report differentiates bus usage for data transfer (between the SPE's local store and main memory) and bus usage for communication between the PowerPC and the SPE. Mechanisms for efficient transfer of data are given in Section 3.2 and a description of the communication mechanisms offered by the Cell is given in Section 3.3. The chapter concludes with Section 3.4, which explains how measuring time on the Cell is done.

3.1 Main Constituents

The Cell processor contains several cores, a high-level overview of this processor is shown in Figure 3.1. One of the cores is a general purpose core, called the PowerPC Processor Element (PPE). This core is mainly used for the bookkeeping of the cores that do the real work on a Cell. The PPE is a 64-bit general purpose processor that conforms, as its name suggests, to the PowerPC architecture. The other cores on the Cell are called Synergistic Processor Elements (SPEs). One Cell processor contains up to eight SPEs. Every SPE consists of a processing unit (SPU) that is capable of performing high-speed SIMD instructions on 128-bit vectors [2].

Every SPE is only able to operate on data that resides in its own local store (LS), which has a size of 256 KiB for every single SPE. The local store is not a cache, the SPE has to request data transfers between the local store and main memory explicitly. Such a data transfer is always delegated by the processing unit to a dedicated memory flow controller (MFC), which is part of the SPE. The memory flow controller can also be invoked by other SPEs or the PowerPC.

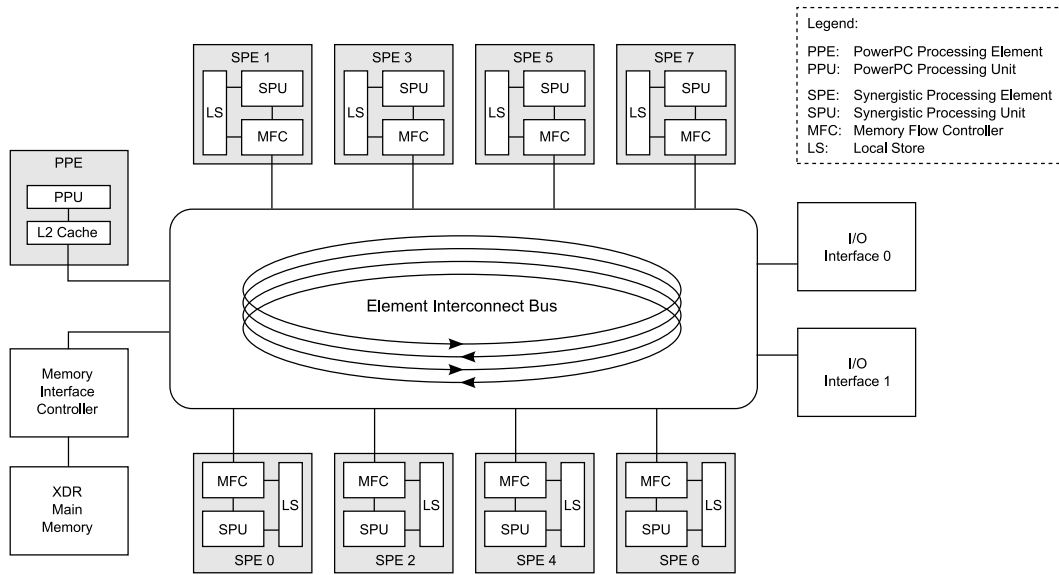


Figure 3.1: High-level overview of the components in the Cell processor. Source: [2]

As a result of the decoupling of data transfer and processing on one SPE, both operations can be done in parallel when an alternating double buffer is used. This technique will be described in Section 3.2.3 in detail. The data transfers are done using DMA over a dedicated bus, consisting of four rings. These memory transfers only succeed if strict rules are obeyed, as will be explained in Section 3.2.1.

All memory transfers between different components of the Cell processor make use of the same bus, which connects all SPEs, the PowerPC, main memory and external I/O devices with each other. The bus consists of four rings, two transferring the data clockwise and two that carry data in the counterclockwise direction. Every ring has a bandwidth of 128 bits and operates at half the speed of the processor [15], but can take up to three concurrent transfers, as long as their paths do not overlap.

In case the entire bus is available to one SPE, it can use two of the four rings for transferring data to and from main memory. This is because a ring can not be used if a transfer would cross more than halfway around the ring. Depending on the distance between the SPE and main memory, the maximal throughput could be 16 bytes per cycle. However, this throughput will not be reached, because of the maximal rate of 8 bytes per cycle at which the MFC operates [15].

3.2 Data Transfer

Mechanisms for efficient transfer of data between the local store and main memory are desired, because the 256 KiB of local storage on one SPE is usually not sufficient. For example, it is not large enough to hold the contents of one entire frame. Therefore, parts of a frame are being transferred from and to main memory, while another part of the frame in the local store is being processed.

In this section, a technique for requesting data transfers at the memory flow controller is discussed, followed by an explanation of a technique called *double buffering*, which allows for computation and data transfer to be executed in parallel.

3.2.1 Direct Memory Access

A data transfer between global memory and the local store of a specific SPE is done using the DMA mechanism and can either be initiated by the PowerPC or by the SPE itself. The latter choice is common practice, because there is a short communication path between an SPE and its memory flow controller [2], while the PowerPC communicating with an MFC is done using the bus. Furthermore, direct memory transfers between the local stores of two SPEs are also supported.

A chunk of data in main memory can be transferred to a local store and stored as 128-bit vectors, if its size is a multiple of 128 bits. The transfer is performed the most efficient, if the data is aligned on a 128-*byte* boundary and if the data size is a multiple of 128 *bytes*. The maximal data size for one DMA request is 16 KiB, and a maximum of 16 DMA requests can be queued at a time. The maximum size of outstanding DMA requests for one SPE is therefore $16 \times 16 = 256$ KiB, which equals the size of its local storage area.

IBM's Cell programming tutorial [16] describes the performance of DMA in a qualitative manner: "The performance of a DMA data transfer is best when the source and destination addresses are aligned on a cache line boundary and are at least a cache line sized (i.e. 128 bytes)." Dou et al. [17] have measured the performance of direct memory access on the Cell and observed that the DMA bandwidth can be up to a factor 2 as large when the memory size is a multiple of the cache line size, compared to different memory sizes. Furthermore they show that the optimal bandwidth is not achieved for memory sizes smaller than 2 KiB.

Once a DMA transfer is requested by the SPU, it is handled completely by the MFC. As a consequence, the SPU can neither influence the order nor the speed in which pending DMA requests on the Cell processor are handled. However, there are a few things the SPE can do.

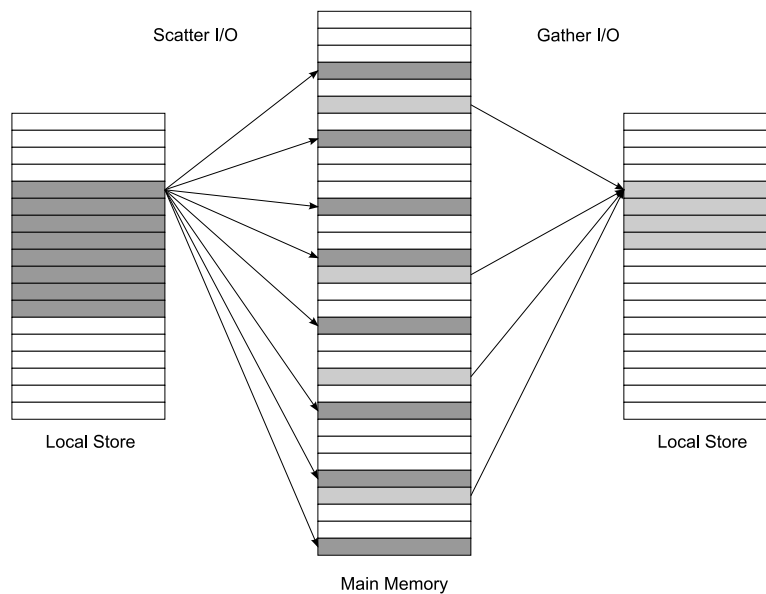


Figure 3.2: With scatter I/O it is possible to transfer a number of equally sized, consecutive portions of data from the local store to arbitrary locations in main memory. With the gather I/O technique, equally sized portions of data at arbitrary locations in main memory can be transferred to consecutive locations in the local store. Source: [2]

Every DMA request that is sent to the memory flow controller contains a 5-bit tag. There is a command available for the SPU to check the status of all memory transfers with a specific tag and a command to wait for completion of all pending DMA requests with a specific tag. A memory transfer can also be requested with the *fence* or with the *barrier* option. In the first case, the MFC will ensure that all DMA transfers that are requested before the fenced request, will be performed before it. The barrier option extends the fence option with the insurance of the MFC that also all DMA requests done after it, will actually be performed after it.

3.2.2 Memory Access with Request Lists

The memory flow controller in the SPE accepts not only single DMA requests, it can also accept a request that carries a pointer to a list and perform so-called vectorized I/O, or scatter/gather I/O. The argument for using this technique is that it enables the SPU to order up to 2048 DMA transfers with a single request to the MFC. This is useful, because the MFC can queue only a maximum of 16 requests [2]. If the SPU tries to add a DMA request to a full DMA queue, the SPU blocks, possibly leading to unpredictable response-times.

There are a few drawbacks to the scatter/gather I/O technique though. First of all,

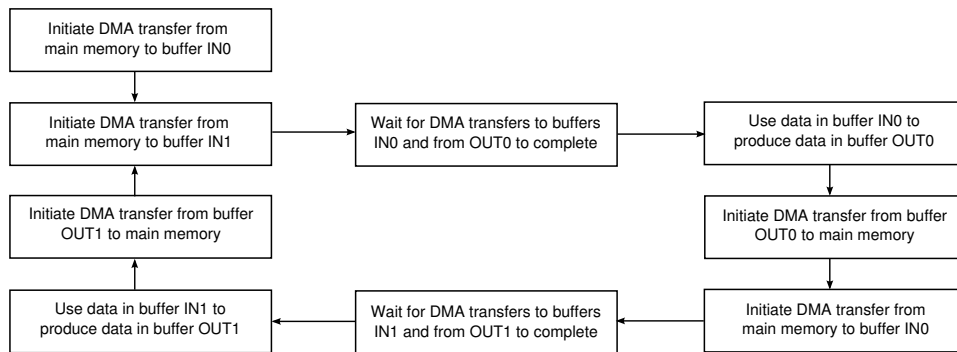


Figure 3.3: The concept of double buffering. Rectangles indicate actions, arrows indicate transitions that can be taken after an action has completed. Source: [16]

the list must be constructed entirely before the actual request is done. Furthermore, the transfers in a DMA list can access arbitrary locations in main memory, but must sequentially go to (or come from) a single location in the local store. This behavior is shown schematically in Figure 3.2. Finally, all transfers in a single DMA request list must either go from the local store towards main memory (scatter), or vice versa (gather).

3.2.3 Double Buffering

Due to the decoupling of a memory flow controller and a processing unit on one SPE, computation and data transfer can be done in parallel. This can be done reliably using a technique called *double buffering*.

In order to allow for double buffering, both the input- and the output-buffer are divided into two distinct parts. All parts are large enough to contain either input or output values for the processing of one block of pixels. At any point in time, one of the two input buffers contains data for the current block to be processed. The other input buffer is filled with data for processing of the following block. After the processing of the current block has been finished, the two buffers exchange their tasks. For the output buffer, the process is almost the same. One part of the buffer is used for the output of the previous block, which is being transferred to main memory. The other part of the buffer is used for the output of the current block. These two buffers also exchange their tasks after the processing of the current block has been finished.

The entire process is outlined in Figure 3.3. This figure is an extension to Figure 26 in the IBM Cell/B.E. Programming Tutorial [16].

3.3 Communication Mechanisms

The Cell processor provides two mechanisms for communication between the PowerPC and the SPE, that can be used for the PowerPC to delegate work to the SPE. These mechanisms are communication using mailboxes and communication over signal notification channels [2].

Every SPE supplies one mailbox for incoming messages, two mailboxes for outgoing messages and two signal notification channels for incoming signals. All these communication mechanisms use messages of exactly 32 bits in size and all messages propagate over the element interconnect bus, the same bus used for memory transfer between main memory and the local store of an SPE. The communication mechanisms outlined in the coming sections are summarized in Table 3.1.

3.3.1 Mailboxes

The incoming mailbox (where ‘incoming’ is from the SPE’s perspective) has a capacity of four messages and follows the first-in-first-out principle. The SPE can read messages from this mailbox, the PowerPC and other SPEs can put messages into the mailbox.

It is possible to check the capacity of a mailbox before reading or writing to it. However, if the SPE tries to read from an empty mailbox, processing on the SPE will block until there is a message to read. Likewise, if an external party tries to send to a full mailbox, the sending entity blocks.

The two outgoing mailboxes operate likewise, although both have a capacity of one instead of four and the PowerPC is the only entity capable of reading from these mailboxes. One of the two mailboxes is able to raise an event at the PowerPC upon receiving a message and is therefore called the outgoing *interrupt* mailbox.

The term interrupt is a little misleading here, because it is not possible to raise a software interrupt on the PowerPC upon receiving a message. In reality, what happens is that

Entity	Type	Capacity
Mailboxes	Input	4
	Output	1
	Output with interrupt	1
Signal channel 1	Input	1
Signal channel 2	Input	1

Table 3.1: Summary of all communication mechanisms offered by one SPE. ‘Input’ means that the PowerPC and other SPEs can write into this entity, ‘output’ works vice versa.

an event is added to the PowerPC's event-queue when the outgoing *interrupt* mailbox contains a message.

A choice can be made between using the outgoing *interrupt* mailbox, while the PowerPC polls its own event-queue and using a normal outgoing mailbox, while the PowerPC periodically polls the status of this mailbox. In theory, the first method should result in lower latency, because it implies uni-directional communication over the bus, whereas the second method implies bi-directional communication.

The latencies of the two methods have been compared in a quantitative manner (see Section 6.3.1). Surprisingly, the second method results in lower latency than the first method, so the second method is used throughout this report.

3.3.2 Signal Notification Channels

Besides communication using mailboxes, the Cell also provides a signalling mechanism with signal notification channels. These channels support only communication towards an SPE, coming from either the PowerPC or another SPE. Communication towards the PowerPC is not possible using signalling.

Unlike the mailbox communication mechanism, writing to a signal notification channel never results in blocking. Sending a signal into a full channel either results in the signal being lost or results in the contents of the channel being logically OR-ed with the incoming signal. This depends on the channel-mode, which can be set during initialization of the SPE.

The SPE can enable interrupts, so that it is interrupted every time an incoming message is received and/or an incoming signal is pending. This mechanism is explained in more detail in Section 5.2.2 on the different implementations of the preliminary termination mechanism. Scarpino's book [2] notes that enabling interrupts on the SPE requires additional processing time, but this amount is not quantified.

Signals and messages that are exchanged between different components of the Cell processor propagate over the bus. The signals and messages can be delayed if the bus is highly used by other components, but every signal or message is guaranteed to be delivered at some point in time [2].

3.4 Mechanisms for Measuring Time

The implementation of the decision scheduler, as well as several measurements presented in the remainder of this report, rely on accurate monitoring of time on the Cell processor.

Both the PowerPC and every SPE have their own time base registers that are updated with a certain frequency, regardless of whether the Cell is in its power-saving state or not. When the Cell is in the power-saving state, the Cell reduces its clock frequency [18].

The update frequency is equal for all time base registers on all cores. The update frequency for the time base register on the PowerPC can be inspected by the operating system. On our platform (see Appendix E) this frequency is 79.8 MHz, so the smallest period of time measurable is 12.5 ns.

On the PowerPC, the time base register is a 64-bit register that is reset at the power-on of the system. Therefore, it is impossible for this time base register to overflow.

On the SPEs, the time base register is just a 32-bit register. On our platform, this timer has a lifetime of 53 seconds. It is possible though to reset the SPE's time base register at any point in time. So, if we measure time intervals on the SPE that are guaranteed to be smaller than 53 seconds, we can measure them using the decrementer with a precision of 12.5 ns.

Chapter 4

Porting the Application

This chapter describes the implementation of the decision scheduler and a scalable video algorithm on the Cell processor. It explains the choices (either caused by the target platform, caused by the source implementation on the general purpose platform or arbitrary choices) that have been made during the implementation phase.

The chapter starts in Section 4.1 with a listing of assumptions that have been made and boundary conditions that have been identified.

Section 4.2 presents a decomposition of the scalable de-interlacer, which is a priority processing algorithm already existing on a general purpose platform, that is ported to the Cell platform into components and describes the properties of every component. Section 4.3 addresses the decisions made when porting the scalable de-interlacer to the Cell platform.

Finally, in Section 4.4 some details for an efficient implementation of the scalable de-interlacer on the Cell platform are explained.

4.1 Assumptions and Boundary Conditions

Before starting an attempt towards a mapping of the priority processing application on the Cell platform, a number of assumptions and boundary conditions has been identified.

Assumptions:

- A1. The decision scheduler is assumed to have the entire PowerPC at its disposal.
- A2. Only one SPE is used for the priority processing algorithm(s), this SPE is assumed

to be available at any time.

- A3. A fixed memory partitioning on the SPE is assumed.
- A4. It is assumed that the local storage on the SPE is sufficient for both algorithms (code, stack, global variables) and buffers.

Boundary conditions:

- B1. There is no operating system running on the SPE.
- B2. The operating system running on the PowerPC is not an RTOS, but a Linux-based general purpose OS. Therefore, the OS might not offer high-resolution timers and predictable response times for applications running on top of it.

The priority processing application, as it already exists on a general purpose platform, contains two scalable video algorithms, being a scalable de-interlacer and a scalable sharpness enhancer.

For investigation of different preliminary termination mechanisms, it suffices to have only one SVA running on the Cell processor. The scalable de-interlacer was chosen to be mapped on the Cell processor. This was an arbitrary choice, the scalable sharpness enhancer could have been chosen equally well. For investigation of the other two mechanisms, we need to run two scalable video algorithms on the Cell. For our research though, it suffices to have the implementations of the same SVA twice, working on two independent input streams, running on the Cell processor.

4.2 Decomposition of the Scalable De-interlacer

The task of a de-interlacer is defined by Schiemenz [19] as to “fill in the missing lines of interlaced transmitted or stored video signals for representation on progressive pixel-oriented displays, such as plasma or LCDs.”

The scalable de-interlacer belongs to the class of scalable video algorithms and has therefore a typical structure as outlined in Figure 2.1. A decomposition of the scalable de-interlacer into components is drawn in Figure 4.1. The different functions this SVA is composed of, are explained in the sections below, except for the ‘control’ part, which is not implemented on the Cell platform.

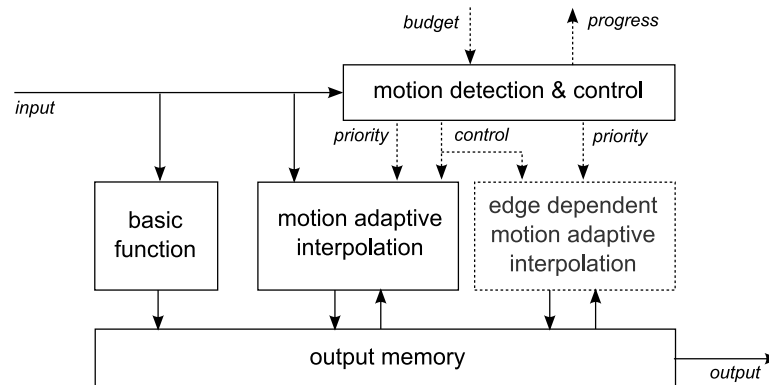


Figure 4.1: A decomposition of the scalable de-interlacer into components. The blocks indicate functions, with the exception of the *output memory* block, which indicates a buffer. The solid arrows indicate data flow, the dashed arrows indicate control flow. The *edge dependent motion adaptive interpolation* function is an optional function, without this function, the algorithm is still a priority processing algorithm. Source: [19]

4.2.1 Basic Function

The input for the scalable de-interlacer is an interlaced video stream stored on disk. Every frame in the input stream consists of lines belonging to two fields; the current field and the field next in time. For every field, the lines are alternating either present or missing. In odd fields, the odd lines are present and in even fields the even lines are present.

The non-scalable basic function of the scalable de-interlacer uses linear spatial interpolation to fill in the missing lines. The interpolation is done on both luminance and chroma values of pixels in the line exactly above and in the line exactly below the missing line, which results in a low-quality output. The basic function of the scalable de-interlacer is not interruptible, since it is a mandatory component of the priority processing algorithm.

4.2.2 Motion Detection

After the basic low-quality output has been computed, the contents of the field can be analyzed. During this phase and the next phase, the functions operate on luminance values of pixels only, as opposed to the non-scalable basic function. The first part of the field-analysis is the detection of motion in the missing lines, using a technique as described by Hentschel in [20]. The input for this function consists of six different lines and the output is one integer for every pixel.

4.2.3 Noise Estimation and Block Ordering

The motion detection function is followed by a function that estimates the noise-level in the current field. This value is based on the luminance values of 500 pixels taken at random positions from the current frame. The computed noise value will serve as an input-parameter for motion detection of the next field.

After the noise-level has been computed, a priority queue is constructed based on the computed motion values. This involves an implementation of a sorting algorithm. Both the noise estimation and block ordering functions are part of the ‘control’ component in Figure 4.1.

4.2.4 Image Enhancement

After the analysis-phase (motion detection, noise estimation and block ordering) has been completed, any remaining time is available for image enhancement. This is performed by the motion adaptive interpolation function. If this function runs to completion and there is still time remaining, the image quality is enhanced by the edge dependent motion adaptive interpolation function. Both functions operate on pixel-blocks of a predefined size and are computational intensive.

4.3 Porting of the Scalable De-interlacer

The components of the scalable de-interlacer, as explained in the previous section, have been implemented on the Cell platform. This section presents the decisions that have been made during the implementation phase.

The behavior of the scalable de-interlacer on the Cell over time is outlined in Figure 4.2 at the end of this section.

4.3.1 Basic Function

The non-scalable basic function operates on pixels in isolation and is therefore well parallelizable. Since the function also consists of arithmetic operations only, it was decided to port this function to the SPE. An alternating double buffer is used to parallelize computation and data transfer on the SPE. In order to maximize the size of the DMA requests from and to main memory, it was decided to let the basic function operate on row-basis, since frame content is stored in main memory in row-major order. That means that all buffers are capable of containing one row of pixels.

The buffer-size for this function is the size needed to store the pixels in five rows, two being for current input, one for future input, one for current output and one for previous output. This buffer-size follows from the buffer-size used in the prototype algorithm running on a general purpose platform.

4.3.2 Motion Detection

For the same reasons as the non-scalable basic function, it was decided to port the motion detection function to the SPE. Again, the double buffering technique from Section 3.2.3 was used in the implementation. Despite the fact that the motion detection function and the non-scalable basic function operate in largely the same way, they have not been combined into one algorithm, because the motion detection function can be preliminary terminated, whereas the basic function can not.

In order to maximize the size of DMA requests, this algorithm was also designed to operate on row-basis. The size of the input buffer is large enough to contain the luminance values of nine rows and the size of the output buffer is equal to the size of two rows of meta-data (i.e. motion values). Would there not be enough memory, then it would of course suffice to let the motion detection function operate on parts of a row at a time. As long as the computation time is greater than the data transfer time, which is the case in the current implementation, the DMA size is not an issue. If the DMA size happens to be an issue, the guidelines in Section 3.2.1 and the measurements from Dou et al. [17] can be used to optimize the DMA throughput.

4.3.3 Noise Estimation and Block Ordering

It was decided to port the noise computation function to the PowerPC, because the SPE lacks functionality for generation of (pseudo-)random numbers and because accessing single pixels in main memory by the SPE implies a high DMA overhead. Dependent on the position of the pixel in main memory, the DMA call could involve 16 times more pixels than necessary.

The construction and sorting of a priority queue is also implemented on the PowerPC, since sorting algorithms are difficult to implement on an SPE using vector operations. Sharma et al. [21] have shown that it is possible to implement a sorting algorithm using vector operations.

The decision to implement these two functions on the PowerPC rather than on the SPE has two unforeseen disadvantages. First of all, this approach introduces more communication between the PowerPC and the SPE. The time needed for communication between these cores is not negligible, as is shown in Chapter 6.

The second disadvantage is the implication that the PowerPC will be unavailable for some time, while it is working on these functions. This contradicts assumption A1: “The decision scheduler is assumed to have the entire PowerPC at its disposal.” If our model is extended so that more SPEs are controlled by the PowerPC, the PowerPC should be available the majority of the time. If the PowerPC is working on a function for one SPE, this could mean that up to five SPEs are waiting for an instruction from the PowerPC.

4.3.4 Motion Adaptive Interpolation

The motion adaptive interpolation function is not only a computational intensive function, it also needs a relatively high amount of input data. To enhance the quality of pixels in x lines, this function needs $4x + 8$ lines of pixels plus x lines of motion values as input. These amounts are the same amounts as used in the prototype scalable de-interlacer implemented on a general purpose platform. Since these values reside at various positions in main memory, filling the buffers for the advanced de-interlacing function results in $5x + 8$ distinct DMA calls.

Despite this high amount of DMA calls, it was decided to port the motion adaptive interpolation function to the SPE, because of the large amount of arithmetic operations involved in this algorithm.

The number of actual DMA calls has been reduced drastically by using DMA request lists, which is explained in Section 3.2.2, for memory transfers from main memory towards the local store of the SPE. By making use of DMA lists, the number of DMA requests decreases, hence there will be less blocking of the SPU. This is at the cost of an increased computational overhead, because besides a DMA request, a list has to be constructed. Measurements have shown that the increase in overhead is relatively small. In the implementation, DMA lists are not used for transfers from the local store towards main memory in the current implementation, because it would imply overhead and the number of DMA requests in this direction is, with at most eight per block, relatively small.

The advanced de-interlacing function as mapped on the Cell platform operates on 128-bit vectors. Therefore, the size of every line transferred from and to the SPE should be a multiple of 128 bits [2]. In the current implementation, one pixel is either stored as 3 consecutive bytes in one buffer or as 3 integers in different buffers. This combination results in the minimal size for one line being 16 pixels. Therefore, the block size on which the advanced de-interlace function operates is decided to be 16×16 pixels, although 8×8 pixels is preferred from a quality-of-service point of view [7].

4.3.5 Edge Dependent Motion Adaptive Interpolation

The implementation of the edge dependent motion adaptive interpolation function has not been finished on the SPE, due to a lack of time. The fact that this function has not been implemented is no problem for the evaluation of mechanisms for dynamic resource allocation in the context of priority processing algorithms, since the scalable de-interlacer is a priority processing algorithm without having this function implemented.

The motion adaptive interpolation function is defined to achieve an output quality between 0% – 50% and the edge dependent motion adaptive interpolation function between 50% – 100%. Therefore, it has been decided that the maximal achievable progress or output quality for the ported scalable de-interlacer is 50%.

4.4 Efficient Implementation on the SPE

The scalable de-interlacer as presented in Section 4.2 has been implemented on the Cell platform. Some choices have been made in order to get a satisfactory implementation. These choices are explained in detail in this section.

4.4.1 Single Instruction Multiple Data

The processing unit within the SPE has been designed to handle data in vectors of 128 bits using SIMD operations. The instruction set of the SPEs [2, 22] is rather limited, but many common instructions are available for vectors of integers, floats and doubles. A subset of the instruction set also operates on vectors of long integers, short integers and bytes.

Processing scalar code on an SPE has a downside. A scalar has to be cast into a vector before applying the appropriate instruction and cast back into a scalar afterwards, causing a serious increase in computation time. Scarpino calls using scalars rather than vectors “wasteful – both in processing time and memory usage.” [2]

The algorithms that have been ported to the SPE rather than the PowerPC, have therefore been rewritten in vector code. The vectorization of the non-scalable basic function and the motion detection function has been performed in a quite straightforward manner. Both functions perform a set of arithmetic functions on every pixel in the frame. These functions are the same for every pixel, no case distinction is made. This has resulted in two vectorized algorithms in which rows of four consecutive pixels are processed in parallel.

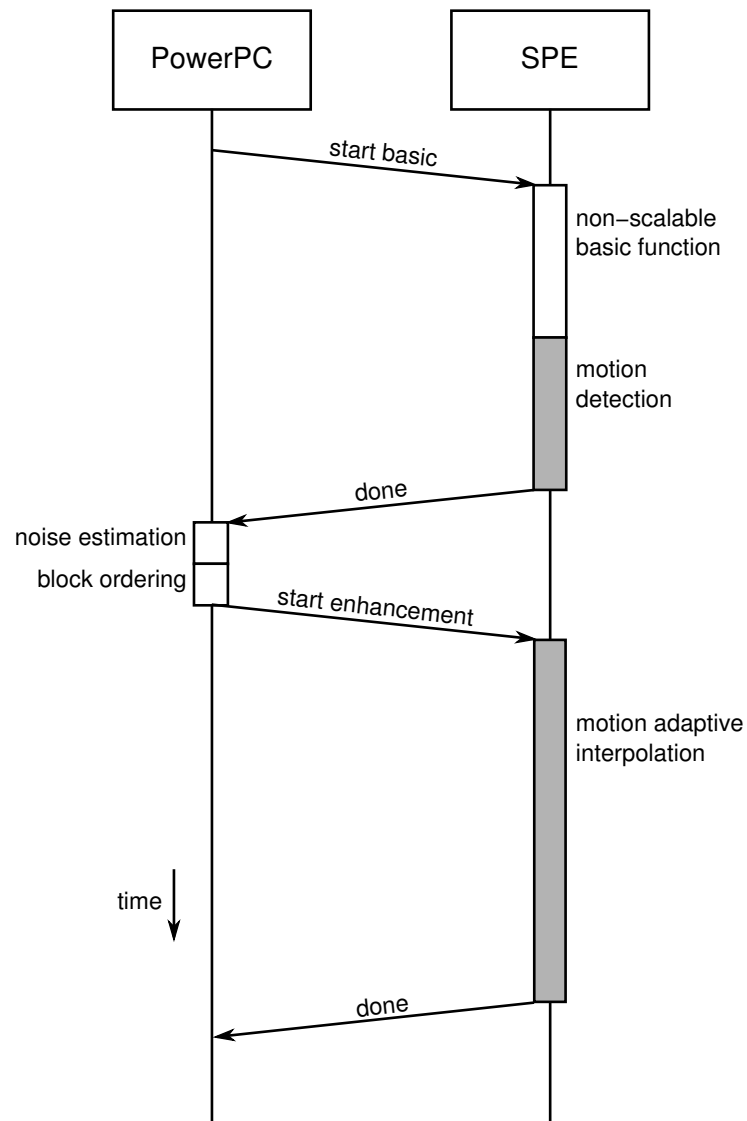


Figure 4.2: Time-line that shows which components of the scalable de-interlacer run on which part of the Cell during the processing of one frame. The shaded parts denote algorithms that can be preliminary terminated.

The vectorization of such algorithms is a time-consuming and tedious effort. Vectorization has been performed onto the scalable de-interlacer by translating every line of scalar code into vector code, every line has been treated in isolation. The process of vectorization has resulted visibly in increased performance. However, this increase has not been quantified. Possibly, more performance can be gained by rewriting the vectorized code in a way that fits the SPE best.

4.4.2 Avoiding Branches

The motion adaptive interpolation function is, as its name implies, not a function in which the same operations are performed on all pixels in the frame. First of all, a large part of the function is skipped if the motion value for a certain pixel, is not within a certain range. If the motion value does lie within the range, the pixel is analyzed for having line-flicker and/or serration properties (the latter is also known as mouse's teeth) with respect to its surrounding pixels, in the spatial as well as in the temporal domain [20]. The interpolation algorithm that enhances the pixel based on the pixel-analysis is a case-distinction over four distinct interpolation functions. Three of these functions are considerably small in size and run in a constant amount of time. However, one of the functions is of considerable size and contains an iteration, which causes it to run in linear time. A high-level overview of this function is shown in Code Listing B.1 (Appendix B). The code structure as presented in Listing B.1 has not been mapped in a one-to-one fashion onto the SPE. First of all, this branch intensive code will perform poorly on the Cell processor, because the Cell lacks a dynamic branch predictor.

The main reason, however, for choosing a different structure is the fact that four pixels are processed in parallel. Despite the four pixels being consecutively aligned in one row, it is unlikely that only one interpolation function applies to all four pixels. Therefore, it was chosen to apply all four interpolation functions on all pixels that are processed in parallel, resulting in four sets of intermediate results. These intermediate results are filtered using the motion values to get the final results of the interpolation function. Only in the case that all four motion values are outside of the range of interest (0 – 200), a branch is taken so that the large computation part is avoided. The new structure in vectorized code is shown in Code Listing B.2 in Appendix B.

If for some reason a branch in the vectorized code can not be avoided, it can be valuable to hint the SPU's static branch predictor at compile-time of the expected value of the guard of a conditional statement. Hinting the SPU can be done with the `__builtin_expect` compiler directive [2]. In case the SPU is not hinted, it will always assume the guard of a conditional statement and the guard of an iteration to be true. In case the static branch predictor happens to be wrong, the pipeline needs to be flushed, causing an 18-cycle penalty, because the wrong instructions have been prefetched.

Chapter 5

Implementation of Dynamic Resource Allocation Mechanisms

This chapter describes the different implementations of the mechanisms for dynamic resource allocation that have been identified. Since the implementation of the first two mechanisms relies on communication mechanisms, communication is treated first in Section 5.1.

Section 5.2 describes the different implementations of mechanisms for preliminary termination. Section 5.3 describes how context switching, i.e. a prerequisite for resource allocation, is done on an SPE. Finally, Section 5.4 describes the implementation of the monitoring mechanism.

5.1 Implementation of Communication

The decision scheduler, running on the PowerPC, is responsible for the allocation of resources to one or more SVAs running on the SPE. More specifically, the tasks of the decision scheduler are (1) to keep track of elapsed time, (2) to notify all priority processing algorithm(s) of frame deadlines and (3) to decide which algorithm to allocate the next time slot to. Figure 5.1 shows the mapping of the decision scheduler controlling multiple SVAs on the Cell platform.

From this figure, it shows that mechanisms for dynamic resource allocation rely on reliable and efficient communication between the PowerPC and SPE. The implementation of communication between these cores is done using the communication mechanisms offered by the Cell platform (Section 3.3) and is explained in this section.

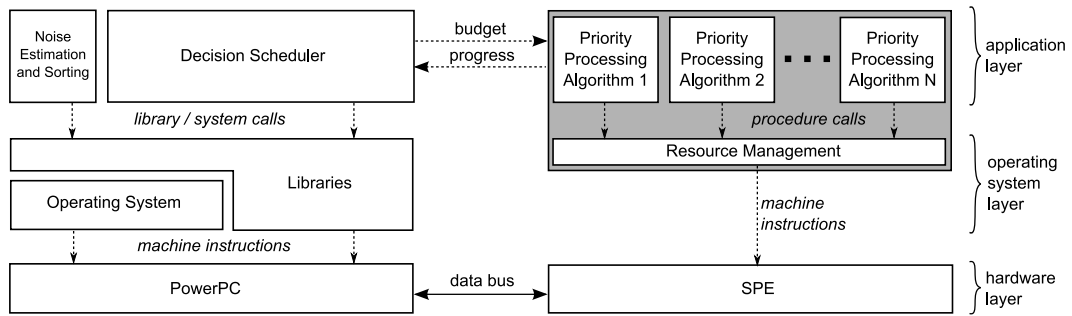


Figure 5.1: The structure of the decision scheduler controlling multiple priority processing algorithms, as implemented on the Cell platform. Boxes indicate hardware or software components. Dotted arrows indicate a logical connection, solid arrows indicate a physical connection.

5.1.1 Communication between PowerPC and SPE

The PowerPC is responsible for controlling which algorithm runs on the SPE. Therefore, the SPE is programmed to wait for the arrival of a signal in the first signal notification channel during initialization of the SPE and after a running SVA has been preempted. The SPE could also have been programmed to wait for a message, but since messages and signals are implemented in a similar way in the hardware [2], the arbitrary choice has been made to wait for a signal.

The first four bits of such a signal specify the task to be done by the SPE. The tasks are: (1) start processing a new frame, (2) resume processing of a frame, (3) preempt the running algorithm and (4) abort all algorithms. There are 12 more tasks possible to implement in the future. The fifth bit indicates which of the double alternating buffers the algorithm should use. The sixth bit indicates which of the two SVAs the message is for. The remaining 26 bits are for SVA specific parameters, e.g. noise value. If these 26 bits are insufficient in a future implementation, the message could either be followed by another message or signal, or the parameters could be transferred to the SPE from some pre-determined location in main memory.

After either completing the work or preliminary termination of the algorithm running on the SPE, the SPE notifies the PowerPC either by putting a message into its outgoing mailbox, which is checked periodically by the PowerPC or by putting a message into its outgoing interrupt mailbox, which raises an event, for which the PowerPC checks periodically. These two approaches are compared in Section 6.3.1, the approach performing the most efficient will be used in the implementation of the mechanisms for dynamic resource allocation that rely on communication.

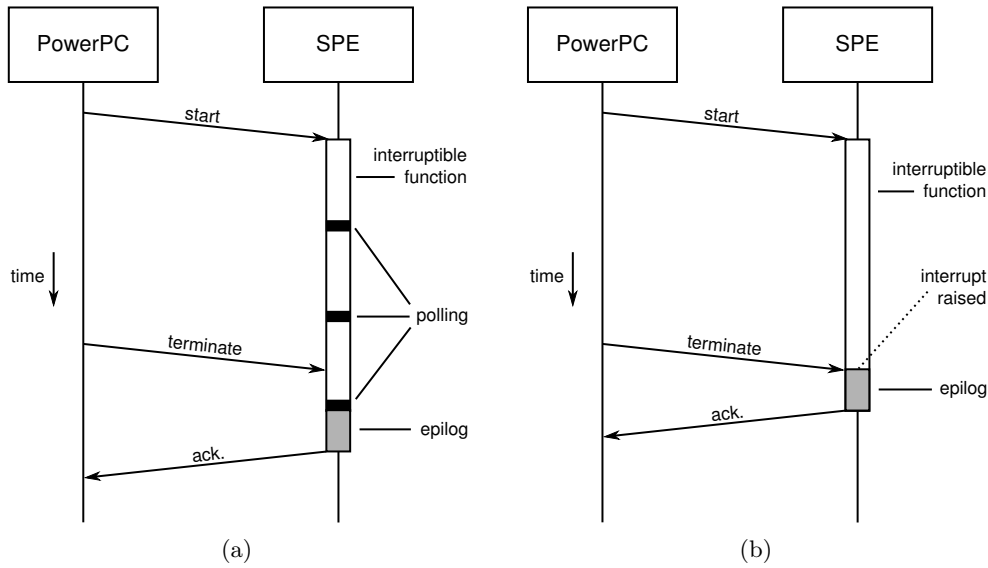


Figure 5.2: These figures show the time-line in case the decision scheduler on the PowerPC preliminary terminates a task running on the SPE. In (a), polling is used to detect the message and in (b) this is done using software interrupts.

5.1.2 Communication Safety

Although the messages and signals used for communication between the PowerPC and SPE do not carry any identifier and could theoretically introduce a possible race condition, the communication scheme between the PowerPC and SPE is safe. This holds, because on the Cell, signals and messages are guaranteed to be delivered and because at all times and because in our implementation only one message of a certain type or with a certain task is sent.

5.2 Preliminary Termination

The priority processing algorithms, controlled by the decision scheduler, behave like periodic tasks in a real-time system. Therefore, it is assumed that the total worst-case computation time for all mandatory components processing one frame in the system is strictly smaller than one frame period. The tasks are allocated a certain amount of processing time and when a predefined deadline is reached, the task should terminate itself or should be terminated within a certain limited amount of time.

Since there is no operating system scheduler on the SPE, it is preferred to use cooperative termination for the SVAs. The communication mechanisms on the Cell are suitable for

```

1 while (!all_blocks_processed && running)
2 {
3     while (!entire_block_processed && running)
4     {
5         Process 4 pixels in parallel...
6
7         poll_signal_channel1(); //Present in case of pixel-polling
8     }
9     poll_signal_channel1(); //Present in case of block-polling
10 }

```

Listing 5.1: Outline of an SVA with polling statements.

cooperative termination, but the SPE needs some mechanism for detecting incoming messages within a reasonable amount of time. There are two main possibilities for detecting the arrival of a message, either by periodically polling the incoming signal channel or by enabling a software interrupt to be raised upon delivery of a message. These two possibilities are explained in the following sections and are shown graphically in Figure 5.2.

5.2.1 Polling on the SPE

If the polling mechanism is used for preliminary termination of the SVA, the computational part of an SVA implemented to run on one of the SPEs is generally outlined like the code in Listing 5.1. Van den Heuvel et al. [9] identified two positions in the code to be well-suited for polling the incoming signal channel. In case polling is done after the parallel processing of every four pixels, the method is called ‘per pixel polling’. If the channel is polled after one block of $16 \times 16 = 256$ pixels has been processed, the method is called ‘per block polling’.

One would expect polling to be a costly operation on a vector processor like the SPU, because it lacks a dynamic branch-predictor. However, using the branch hinting technique as explained in Section 4.4.2, the costs can be limited. The SPU can safely be hinted at every time it polls for an incoming signal, no signal has been received. As a result, only after receiving a signal, a penalty must be taken. But since the SVA has to jump out of the pixel processing loop(s) in Code Listing 5.1, the pipeline needs to be flushed anyway.

5.2.2 Interrupt Handling on the SPE

The procedure necessary for registering the SPE to listen for software interrupts and handling a single interrupt is explained in detail in Section 13.2 of Scarpino's book [2]. It basically boils down to the following steps:

1. Select the event(s) of interest;
2. Create an interrupt service routine (ISR);
3. Enable checking for interrupts.

If one decides to write a global variable in the ISR, any other writes to this variable during regular execution become *critical sections*. Checking for interrupts should be disabled upon entering any critical section, and can be enabled after exiting it.

Scarpino [2] describes that part of the ISR should be acknowledging the event that caused the interrupt, because a pending event causes software interrupts to be generated again and again. What Scarpino does not describe, is that the interrupt return address might not be immediately available after an software interrupt has been raised and that the acknowledgement channel might be written with some delay. Therefore it is recommended to wait for all pending write actions to channels to complete, using the `spu_sync_c()` instruction, before finishing the ISR [18].

In our priority processing application, the desired behavior of the interrupt service routine is that it terminates the algorithm that is executed on the SPE, possibly after executing some epilog phase, and notifies the application running on the PowerPC. This has been implemented by doing a non-local jump, by making use of the `setjmp()` and `longjmp()` commands. Code Listing 5.2 presents an outline of such a structure.

5.3 Resource Allocation

An implementation consisting of a decision scheduler controlling multiple priority processing algorithms is a necessary prerequisite for analyzing an implementation of the resource allocation mechanism and the monitoring mechanism. It was chosen to implement the scalable de-interlacer twice on one SPE, with both SVAs operating on independent video streams.

Due to the limited amount of local storage at the SPE side, either the buffers used by one scalable de-interlacer now have to be shared by both SVAs, or the sizes of the buffers have to be reduced. Since the first option implies that a buffer has to be refilled after a context switch occurs, the latter option has been chosen. It was unsure whether a

```

1 #include <setjmp.h>
2 jmp_buf jump_buffer;
3
4 void spe_receive_job (int job)
5 {
6     if (setjmp(jump_buffer)) { goto epilog; }
7     execute_job(job);
8
9     epilog:
10    send_done_message();
11 }
12
13 void interrupt_service_routine (void)
14 {
15     acknowledge_event();
16     empty_signal_channel();
17
18     spu_sync_c();
19     longjmp(jump_buffer, 1);
20 }

```

Listing 5.2: Implementation of preliminary termination using interrupts.

reduction of the buffer size is possible without influencing the performance in a negative way. Measurements have been performed to clarify whether performance has been lost due to this decision or not.

The implementation of the non-scalable basic function and motion detection function makes use of several fixed-size single-place buffers. A reduction of the sizes of these buffers by the same factor implies that the number of DMA requests increases, whereas the amount of data transferred per request decreases. This results in larger data transfer times [17]. If the processing time per block is larger than the data transfer time per block, a reduction in buffer size has no negative effect on the overall performance of the scalable de-interlacer. Since data transfer times can not be measured on the Cell, the total waiting times for DMA requests to complete are compared between the implementation with the original buffer size (720 pixels) and with the reduced buffer size (240 pixels).

It is expected that reducing the buffer size will not influence the average processing time

Buffer size	Basic function	Motion detection
720 pixels	96.046 μ s \pm 13.363	196.007 μ s \pm 4.382
240 pixels	110.591 μ s \pm 5.027	194.149 μ s \pm 1.211

Table 5.1: Average waiting times for DMA transfers to complete per frame for different buffer sizes and different functions of the scalable de-interlacer.

Algorithm	VQEG 3	VQEG 5	VQEG 6
Basic de-interlacing	4.068 ms \pm 0.078	4.075 ms \pm 0.063	4.055 ms \pm 0.035
Motion detection	8.328 ms \pm 0.027	8.344 ms \pm 0.034	8.355 ms \pm 0.033

Table 5.2: Average execution times for two time-consuming, content independent parts of the priority processing algorithm, operating on buffers of a reduced size.

of the motion detection function, since this is a computation-intensive function, hence the time needed for computation outnumbers the time needed for data transfer. However for the basic function, it is expected that the processing time will increase as a result of the smaller buffer size. The results of the comparison are outlined in Table 5.1 and confirm the expectations. Note that from this table, it seems that the motion detection function performs better for smaller buffer sizes. However, this can not be claimed, since the confidence intervals overlap.

Reducing the buffer size for the non-scalable basic function and for the motion detection function has resulted in a more efficient implementation of the scalable de-interlacer, because during the adaption of the non-scalable basic function, a bug in a DMA transfer request had been fixed. Since a lot of measurements concerning the evaluation of different implementations of the preliminary termination mechanism had already been performed, a choice had to be made whether to fix this bug or to ignore it. It was chosen to fix the bug, without redoing any measurements. The argument for this choice is that the bug occurred in the non-scalable basic function, which can not be preliminary terminated by definition. This means that the values measured for preliminary termination should not have been affected by the bug.

The average computation times for the old versions of the algorithms, operating on entire rows, are presented in Table 6.5 (Section 6.4.2). The average computation times for the improved algorithms are presented in Table 5.2.

5.3.1 Context Switching

A mechanism for context switching is needed on the SPE, so that it can be shared by multiple priority processing algorithms. The implementation of a mechanism for context switching is given in Code Listing C.1 in Appendix C. For a successful implementation of a mechanism for context switching, a mechanism is needed so that an algorithm can resume its work after it has been preempted earlier, hence it is needed to accurately record the state of the priority processing algorithms at any time. The state of a priority processing algorithm can be divided into (1) the current function that is running, (2) the block or row this function is operating on and (3) the pixel this function is currently processing.

Both scalable de-interlacers write their states using global variables, the values are updated after a successful memory transfer towards main memory. For reasons of efficiency, it was chosen to keep track of the current function (1) and the block or row it is operating on (2), but not to keep track of the pixel that is being processed. This is done in order to limit the complexity of the code where DMA request lists are generated. This approach means that after a context switch, some work is re-done on a small amount of pixels.

Any interruptible function of the scalable de-interlacer is allowed to run either to completion or to the end of a time slot. When the end of a time slot has been reached, such a function is preempted by the PowerPC, using one of the communication mechanisms for preliminary termination. It was chosen to use software interrupts for preempting the scalable de-interlacer, based on the results in Chapter 6.

Whenever an interruptible function on the SPE is preempted, it stops processing pixels and waits for all pending DMA requests to finish. After this has happened, it returns an acknowledgement towards the PowerPC. The decision scheduler running on the PowerPC then decides which SVA to allocate the next time slot to. This information is packed into a message and sent to the SPE. The SVA that is granted the time slot then continues processing pixels, based on the state that has been recorded in global variables. Figure 5.3 shows a typical sequence of actions for the decision scheduler allocating resources to two SVAs.

An interruptible function of an SVA that is allowed to continue its work, first initializes DMA requests for both its buffers, before it starts processing pixels in parallel with data transfers between the local store and main memory. This approach has been chosen to keep a simple implementation, where every component of the SVA always executes following the same sequence, i.e. the double buffering scheme as in Figure 3.3.

In retrospect, some improvements for the implementation of the resource allocation mechanism have been identified. These are explained in detail in Section 7.1 ‘Future Work.’ The main improvement concerns a modification of the decision scheduler and priority processing algorithms so that these execute in parallel. The sequence of actions concerning this improvement is shown in Figure 5.4.

5.4 Monitoring

As explained in Section 2.4, the implementation of the monitoring mechanism should provide accurate accounting of time slots, as well as accurate reporting of the achieved progress by an SVA towards the decision scheduler.

The elapsed time is monitored by the decision scheduler using a busy-wait loop, in which

the DS repeatedly inspects the value of the time base register on the PowerPC. The time base register has an accuracy of 12.5 ns. The achieved progress is reported by an SVA using the acknowledgement (ack) message that is returned to the PowerPC after an SVA has been preempted, as is shown in Figure 5.3.

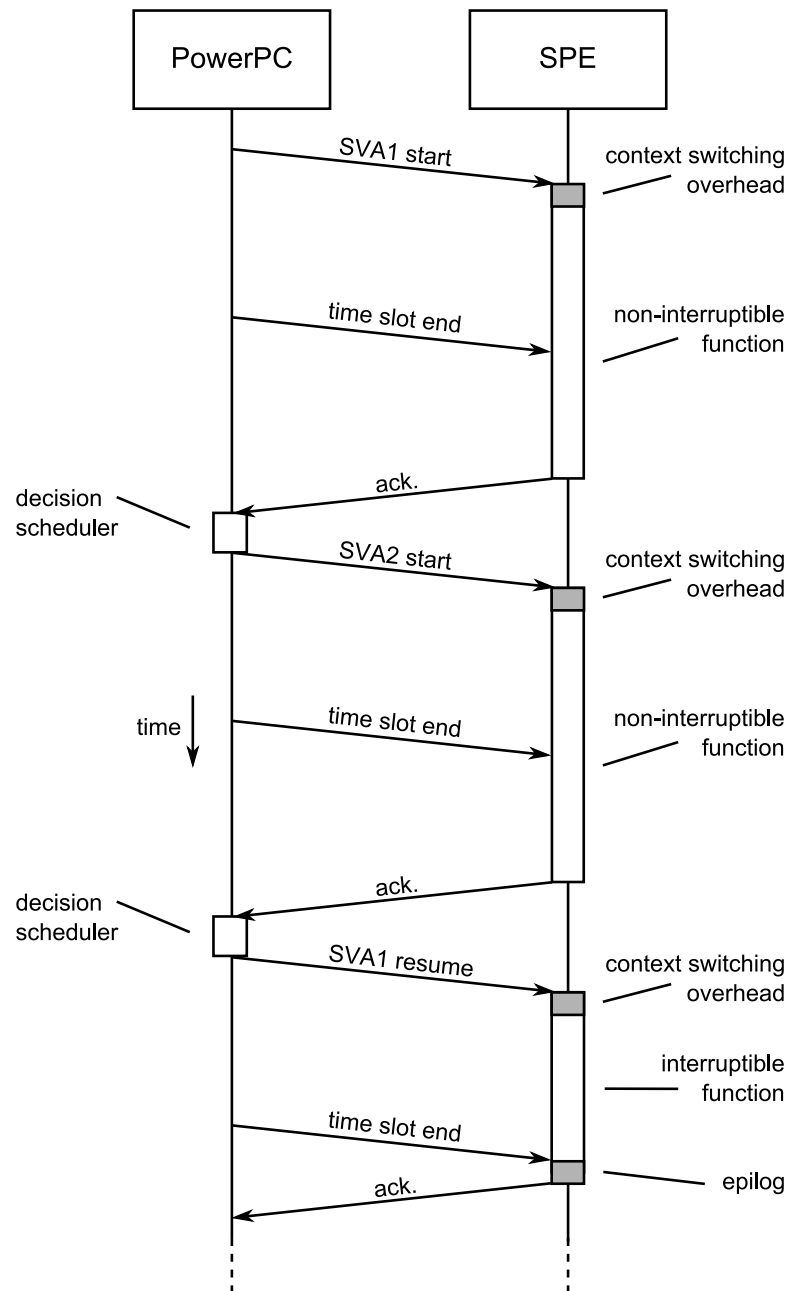


Figure 5.3: This time line shows a typical sequence of actions for a decision scheduler allocating time slots to scalable video algorithms. Arrows from left to right indicate communication between the cores. The application distinguishes mandatory (non-interruptible) and optional (interruptible) functions. For both functions, there is context switching overhead, i.e. time needed for computing the locations of buffers in main memory. Interruptible functions need some time after preliminary termination or a context switch to wait for all pending memory transfers to complete, this is called the 'epilog' phase.

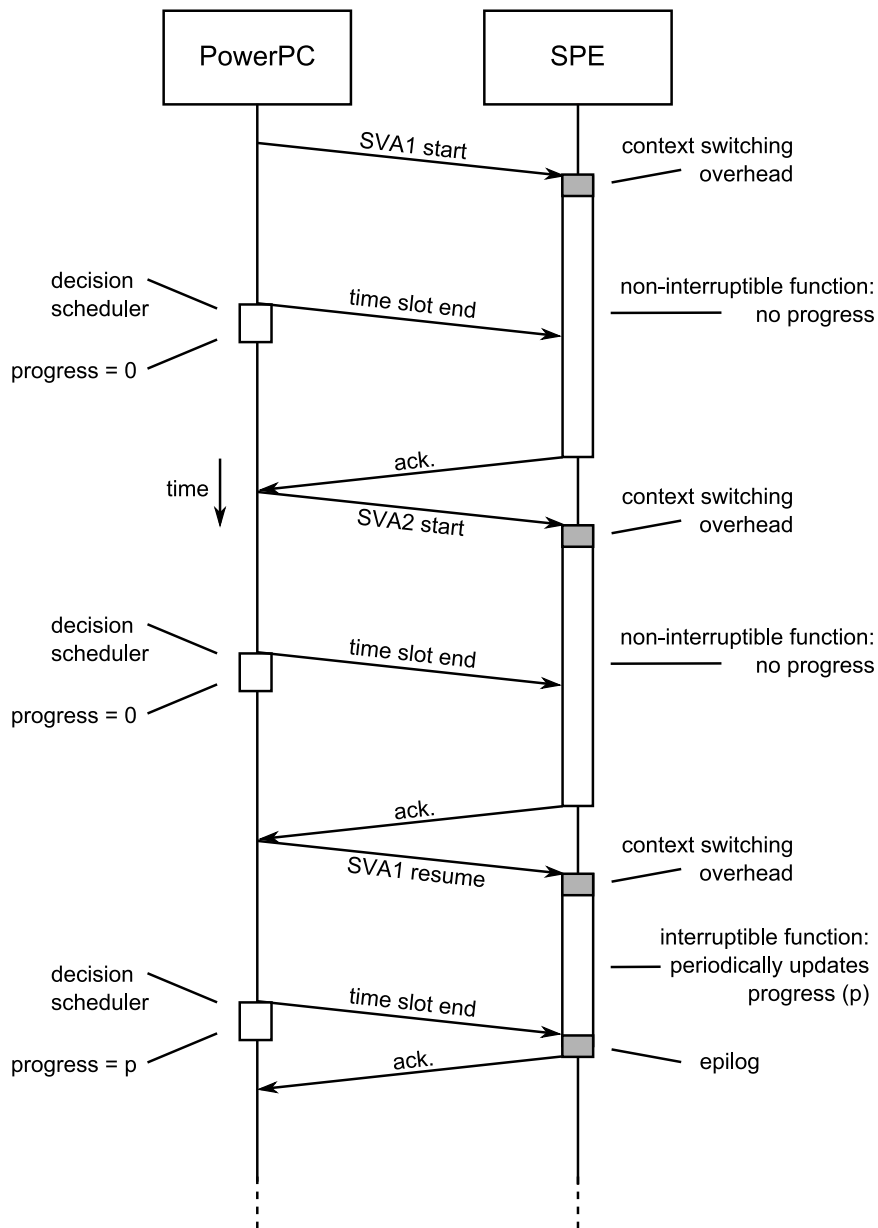


Figure 5.4: The monitoring mechanism has been implemented as shown in Figure 5.3. This time line shows that progress does not have to be reported with an acknowledgement message, but can also be periodically updated in a shared variable in main memory. Subsequently, the decision scheduler could be modified to execute in parallel with the priority processing algorithms. This would mean trading accuracy of the reporting of achieved progress for higher performance of the entire application. This modification is explained in detail in Section 7.1.

Chapter 6

Evaluation

This chapter presents an evaluation of the implementation of the three mechanisms for dynamic resource allocation. Results of experiments regarding different implementation variants are compared with each other. All experiments are done using slightly modified versions¹ of the interlaced, uncompressed video sequences from the Video Quality Experts Group (VQEG) [23], numbers 3, 5 and 6. These sequences will be referred to as VQEG, video or source 3, 5 and 6 in the following sections. All these sequences contain 400 frames of uncompressed video in standard-definition format.

A lot of experiments involve measurements of elapsed time, in the order of milliseconds or even microseconds. All time measurements on the SPE are done using the decremter and on the PowerPC using its time base register, both components have been explained in Section 3.4.

For some measurements, processing times and latencies for the initial two video frames are extraordinary high compared to those for other frames. These peaks in the timings are suspected to be the result of some caching behavior within either the memory interface controller or the main memory. Since these peaks in the results were reproducible and were consistently only for the first two frames of a video sequence, these measurements are left out of the results, without further notice.

This chapter starts with a concise description of the research questions posed in Section 6.1 and their corresponding hypotheses in Section 6.2. In the subsequent sections, an evaluation of the three mechanisms for dynamic resource allocation is given. Section 6.3 presents the latencies measured for different implementations of the preliminary termination mechanism. The absolute, as well as relative, overhead implied by these implementations are described in Section 6.4. In Section 6.5, the implementation of the

¹The video sequences from the VQEG have been converted from RGB to YCbCr color encoding and treat the upper-left pixel as a binary counter, indicating whether the frame is an even or an odd frame.

resource allocation mechanism on the Cell is compared to the general purpose platform. Finally, our implementation of the monitoring mechanism is evaluated in Section 6.6.

6.1 Research Questions

The different implementation variants as described in the previous sections give rise to a number of research questions regarding the preliminary termination and resource allocation mechanisms. No research questions have been posed with respect to the monitoring mechanism, since the implementation for this mechanism is believed to be both efficient and reliable. The research questions are stated below, all of them are answered in the remainder of this chapter.

6.1.1 Preliminary Termination

The goal is to minimize the average termination time needed for preliminary terminating a video processing algorithm running on one SPE, while maintaining a *reasonable* processing time, i.e. limiting the computational overhead. Unfortunately, it is not possible to quantify ‘reasonable processing time’ in advance, not even by a domain expert. This goal results in the following research questions:

1. What are lower bounds, upper bounds and means for communication latency and for termination latency per frame, both for a 95% confidence interval?
2. What are absolute and relative increases in processing time (i.e. computational overhead) after enabling preliminary termination mechanisms?

6.1.2 Resource Allocation

The goals are: (a) to find the relation between the number of time slots per frame period and the time needed for the DS to make a decision; and (b) to find the overhead for two SVAs controlled by the decision scheduler, compared to one SVA running in isolation. These goals result in the following research questions:

1. What is the relation between the average time needed for the DS to make a decision and the number of time slots per frame period?
2. What is the overhead (as a result of communication, context switching and processing time for the DS) for two SVAs controlled by the DS in terms of expected, but not achieved progress?

6.2 Hypotheses

The research questions as described in the previous section are expected to show the following outcomes.

6.2.1 Preliminary Termination

Preliminary termination results in overhead and latency. The termination latency can be split into communication latency and SPE-side termination latency.

1. The communication latency is expected to be larger when the PowerPC periodically polls a mailbox on the SPE, compared to when an interrupt is raised on the PowerPC upon arrival of a new message. The termination latency is expected to be smaller when software interrupts are used on the SPE, compared to periodically polling a communications channel.
2. The overhead introduced by the polling mechanism is expected to be negligible, since the *branch hinting* technique (Section 4.4.2) is used. On the other hand, the overhead introduced by enabling software interrupts is expected to be significant, since Scarpino [2] explicitly states that enabling software interrupts costs additional processing time.

6.2.2 Resource Allocation

On the boundaries of time slots, the decision scheduler needs time to decide which algorithm to allocate the next time slot to.

1. The average time needed for the decision scheduler per time slot is expected to be a continuous amount per time slot, however this amount is expected to be larger when there are more time slots per frame period.
2. The overhead due to communication, context switching and processing time for the DS² is expected to be significant. Therefore, the overall progress achieved for two SVAs controlled by the DS is expected to be less than half of the average output quality achieved by one SVA in isolation.

²The processing time for the DS matters in the current implementation, since the DS and the priority processing algorithms *alternate* rather than execute in parallel.

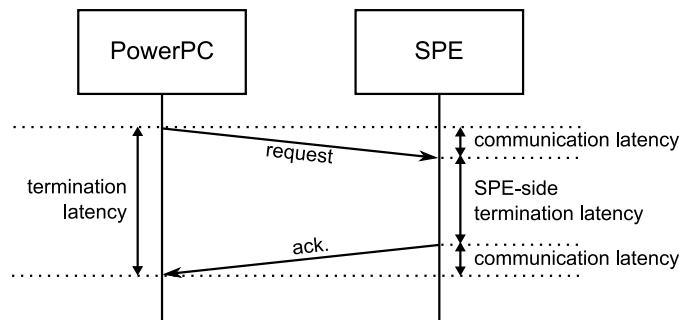


Figure 6.1: Graphical representation of the subdivision of termination latency into communication latency and SPE-side termination latency.

6.3 Preliminary Termination Latency

The preliminary termination mechanism and the different implementation strategies have already been discussed in the sections before. The goal of this section is to evaluate the average latency for the three different implementations of this mechanism. This latency is called the *termination latency* and is defined as the time that elapses at the PowerPC between the moment a preliminary terminate message is sent and the moment that an acknowledgement of termination has been received.

Termination latency can be subdivided into two components, being *communication latency* and *SPE-side termination latency*, as shown in Figure 6.1. The communication latency is explained in the next section. This section is not succeeded by a section on SPE-side termination latency, since this is not measurable. Instead, it is succeeded by a section on the total termination latency. This section is divided into two parts that are based on which phase the SVA is in, when it is preliminary terminated.

6.3.1 Communication Latency

The main difference between preliminary termination on a general purpose platform and on the Cell platform, is the notion of *communication latency*. Communication latency is defined as the time that elapses between the moment a ‘ping’ message is sent from the PowerPC to the SPE and the moment that an acknowledgement for this ‘ping’ has been received (see also Figure 6.1). In other words, communication latency is the time needed for a message to traverse from the PowerPC to the SPE and vice versa. The SPE should be listening continuously for incoming messages and acknowledge these as quickly as possible, to allow for accurate measurement of the mean communication latency. On a general purpose platform, the time needed for a signal to be raised or a global variable to be set is negligible. On a multi-core platform such as the Cell, this is certainly not the case.

Strategy	Latency
1. Polling the outgoing mailbox	49.3 $\mu\text{s} \pm 0.193$
2. Polling the event-queue	78.4 $\mu\text{s} \pm 0.432$

Table 6.1: Mean communication latencies for two different implementations and a 95% confidence interval.

In Section 5.1.1, two approaches for the PowerPC to detect an acknowledgement message have been described. These are: (1) the PowerPC periodically checks the status of the SPE's outgoing mailbox and (2) a message in the SPE's outgoing *interrupt* mailbox causes an event to be raised on the PowerPC, for which it checks periodically. The latency for the second approach is expected to be smaller than the latency for the first, because the second approach needs just uni-directional communication. A message from the SPE to the PowerPC causes an event to be queued on the PowerPC. Then it suffices to poll for events using a busy-waiting loop, which is expected to introduce a negligible latency.

Besides a larger latency, the first strategy is also expected to result in a larger overhead on the bus than the second strategy. This is again expected since the first strategy needs bi-directional communication, whereas uni-directional communication suffices for the second strategy. However, the difference in communication latency has not been measured, since it falls outside of the scope of this report.

For both approaches, a frame period of $T = 60$ ms is used, i.e. a preliminary termination signal is sent exactly 60 ms after the message to start de-interlacing a new frame has been sent. The mean latencies, with a 95% confidence interval, for both implementations are presented in Table 6.1.

The expectation that the second possibility would show a smaller average latency than the first one has not come true. To the contrary, the average latency happens to be over a factor 0.5 larger. The reason for this behavior is probably that the instruction on the PowerPC for checking the event-queue has a processing delay of more than 28 μs . If this is in fact the case, though, a good reason for this behavior of the PowerPC has not been found.

The first strategy has been chosen as the strategy for detecting whether a message has been posted in a SPE's outgoing mailbox. The reason for this choice is that the SPE is the only user of the bus towards main memory. Since the bus is used only a fraction of the time when an SVA is running on the SPE, a slight increase in bus-usage is justified, since it accounts for a serious advantage in communication latency.

As a result of this choice, the lower bound on termination latency will be assumed to be $49.3 - 0.193 = 49.1$ μs in the following section. This lower bound indicates that when using the Cell platform, the possibilities to reduce the latency are bounded by the

communication latency of at least 49.1 μ s.

6.3.2 Termination Latency

An algorithm that is running on one of the SPEs can be preliminary terminated by sending a signal from the PowerPC to the SPE. The detection of such an incoming signal can be done in two different ways. One way is that the SPE could be programmed to poll for incoming signals periodically. A shorter polling period is expected to result in a shorter SPE-side termination latency, but in a larger computational overhead. The other way is to program the SPE that it receives a software interrupt upon arrival of a new signal. A dedicated interrupt service routine should be programmed in such a way, that it handles all preliminary termination prerequisites.

As explained above and shown in Figure 6.1, the total latency for preliminary terminating an algorithm that is running on the SPE, is the sum of two latencies, being the communication latency and the SPE-side termination latency. The SPE-side termination latency is defined as the amount of time that elapses between the SPE receiving a message to preliminary terminate its work and the moment of sending a confirmation message to the PowerPC. The combination of both latencies is outlined in Figure 6.1.

It is impossible to measure the actual SPE-side termination latency, because the moment at which the preliminary terminate message has been received, is unknown to the SPE. Would it be known, then the SPE could be programmed to terminate its work immediately. This behavior is desired, but technically infeasible.

Since the average communication latency has been measured in the previous section, we can measure the total termination latency and subtract the average communication latency from it to give an estimation of the SPE-side termination latency.

The overhead introduced by polling for unread signals is expected to be negligible, as explained in Hypothesis 2 of Section 6.2.1. The overhead introduced by per pixel polling is expected to be 32 times larger than the overhead caused by per block polling. This is because one block of pixels contains $16 \times 16 = 256$ pixels, but only half of the block is processed (128 pixels) by the scalable de-interlacer and every four pixels are processed in parallel, so polling is done once after every four pixels are processed ($128/4 = 32$).

The overhead introduced by interrupt detection, however is expected to be significant. Scarpino [2] writes that enabling interrupt detection on the SPE requires additional processing time. Unfortunately, he does not quantify this.

The SPE-side termination latency for all mechanisms is expected to be smaller than the communication latency in case polling is used, since communication actually takes some time to complete. However, it is impossible to estimate how much larger. The latency

Variant	Latency VQEG 3	Latency VQEG 6
1. Block polling	1.600 $\mu\text{s} \pm 0.186$	1.685 $\mu\text{s} \pm 0.073$
2. Pixel polling	0.831 $\mu\text{s} \pm 0.514$	0.567 $\mu\text{s} \pm 0.058$
3. Interrupts enabled	0.279 $\mu\text{s} \pm 0.096$	0.818 $\mu\text{s} \pm 0.062$

Table 6.2: Average SPE-side termination latencies for different implementation variants. The frame period is set to 40 ms.

for block polling is expected to be larger than for pixel polling, which is in turn expected to be larger than the termination latency when software interrupts are enabled. This is because polling a message with a low frequency generally gives rise to a larger latency, than polling a message with a high frequency, which in turn results in a larger latency than using interrupts.

Termination Latency during the Enhancement Phase

The average SPE-side termination latency is, due to high fluctuations, only measurable if a large number of samples is taken. Therefore, the algorithm has been executed 20 times on VQEG sequence 6. For every single run, the median total termination latency per frame has been taken from that particular run, the run before and the run after. This has been done to filter out the large fluctuations caused by communication latency.

The frame period has been set to 40 ms, which causes the preliminary termination message to be sent during the last phase of the SVA, before the entire frame has been processed.

For every of the 400 frames, the mean termination latency (including communication latency) for a 95% confidence interval has been calculated, which can be found in Figure 6.3. The calculated average SPE-side termination latency for block polling, pixel polling and software interrupts is given in Table 6.2. These values are obtained by subtracting the timing values for one implementation variant by a busy-waiting variant.

From the values in Table 6.2, we can conclude that the SPE-side latency is very difficult to measure, due to the high fluctuations in the communication latency. We can say with 95% confidence that the SPE-side termination latency using interrupts is less than 1 μs for the sequences considered.

In the table, one can also see that the SPE-side latency for block polling is just 2 or 3 times larger when compared to pixel polling. This can be explained by the fact that the SPE has to wait for all pending data transfers before sending an acknowledgement message to the PowerPC. This waiting time clearly dominates the time needed for the SPE to detect a preliminary termination message and take the appropriate actions.

Variant	Latency VQEG 3	Latency VQEG 6
1. Block polling	$2.878 \mu\text{s} \pm 0.104$	$2.865 \mu\text{s} \pm 0.082$
2. Pixel polling	$0.323 \mu\text{s} \pm 0.101$	$0.286 \mu\text{s} \pm 0.071$
3. Interrupts enabled	$-0.103 \mu\text{s} \pm 0.116$	$-0.208 \mu\text{s} \pm 0.064$

Table 6.3: Average SPE-side termination latencies for different implementation variants. The frame deadline is set to 10 ms and the results from the first 40 frames are ignored.

Termination Latency during the Analysis Phase

In the previous section, the average termination latencies have been measured with the frame deadline set to 40 ms, in order to preliminary terminate the priority processing algorithm during the enhancement phase. In this section, we observe the behavior of the priority processing algorithm when it is preliminary terminated after 10 ms, i.e. during the analysis phase.

The results are expected to show a slightly larger latency, due to the fact that the DMA grain size is larger during the analysis phase. Larger memory transfers result in a larger time the SPE has to wait for the data transfers to complete, before it can send an acknowledgement message to the PowerPC.

The results, however, show some unexpected behavior. All results from measurements on the three different implementation variants on two different input sequences showed extraordinary high fluctuations in the latencies for the first 40 frames. For the subsequent frames, these fluctuations suddenly disappear. This behavior has not been observed with measurements performed for previous sections and is shown in Figure 6.2.

Extensive tests have shown that this behavior is not caused by specific properties of the priority processing algorithm, nor is it caused by content dependent properties of the algorithm. The highly fluctuating latencies are expected to be caused by the *bus arbiter*. Chen et al. [15] describe that this entity controls access to the bus. Unfortunately they do not describe the policies used by the arbiter to grant or deny access to the data bus.

It is suspected that this bus arbiter has some learning capabilities for allocating the bus for either data transfer between the SPE and main memory or communication between the SPE and the PowerPC. If this is in fact the case, the arbiter is notified after about 10 frames that the latency for polling the SPE's outgoing mailbox is too high. It adapts its policy and we see in Figure 6.2 a drop in the latency to approximately $80 \mu\text{s}$. This is repeated several times, until the latency reaches a steady value of approximately $50 \mu\text{s}$.

As a result, the first 40 frames are consistently ignored for calculating the average SPE-side latency, when the algorithm is terminated during the analysis phase (see Figure 6.4 for the measured latencies). The calculated latencies are given in Table 6.3. The pixel polling variant shows a smaller latency compared to the latency with a frame deadline of

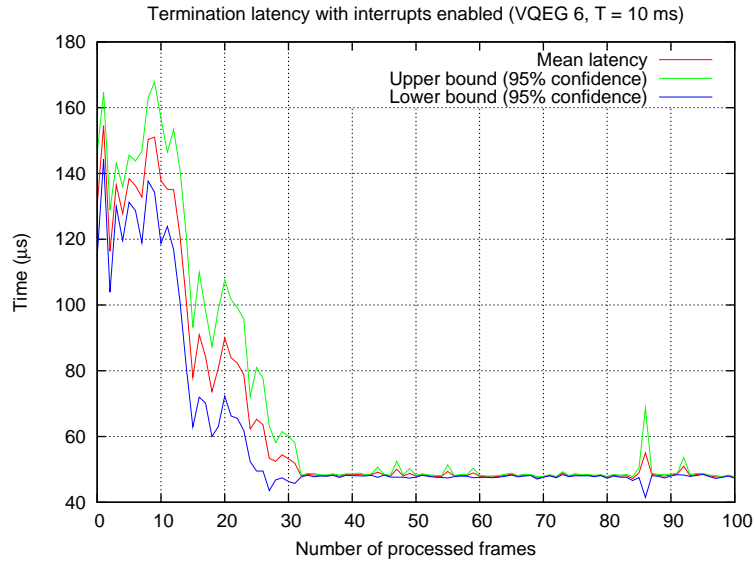


Figure 6.2: Preliminary termination latency per frame with software interrupts enabled and the frame deadline set to 10 ms. The latencies for the first 40 frames are consistently extraordinary high and are therefore ignored for calculating the average latency per frame.

40 ms. This is not as expected, so the larger DMA grain size does not affect the latency much. The smaller latency is probably caused by the fact that the computation time for processing four pixels in parallel during the analysis phase is a lot shorter than during the enhancement phase.

One can observe that when block polling is used, the SPE-side termination latency is larger when processing is preliminary terminated during the analysis phase, compared to when the termination signal is received during the enhancement phase. This can be explained by the fact that the motion detection function, which is part of the analysis phase, operates on row basis (see also Section 4.3). Since this function processes every second row, a ‘block’ contains in this case up to $2 \times 720 = 1440$ pixels, whereas a block during the enhancement phase contains just $16 \times 16 = 256$ pixels.

Finally we see in Table 6.3 that the variant with software interrupts shows a negative latency, when the total termination latency for software interrupts is subtracted from the calculated communication latency. In theory, such a negative latency is impossible. There are several possible reasons for measuring negative latency; the assumption that the SPE immediately returns upon reception of a ‘ping’ message could be wrong. Another reason could be that the communication latency has not been measured accurately enough. However, one can conclude that the variant with software interrupts enabled results in most cases in the smallest latency, being with 95% confidence at most $55 \mu\text{s}$, as well as a reasonably small overhead.

In Figures 6.3d and 6.4d, it shows that the termination latency for software interrupts is larger when interrupted during the enhancement phase compared to the analysis phase. This is probably caused by the waiting time for memory transfers to complete. During the enhancement phase, there are several DMA transfers regarding smaller blocks of data, whereas during the analysis phase, there are less DMA requests, transferring larger data blocks. Probably, the SPE needs almost $1 \mu\text{s}$ more to wait for the smaller requests to complete.

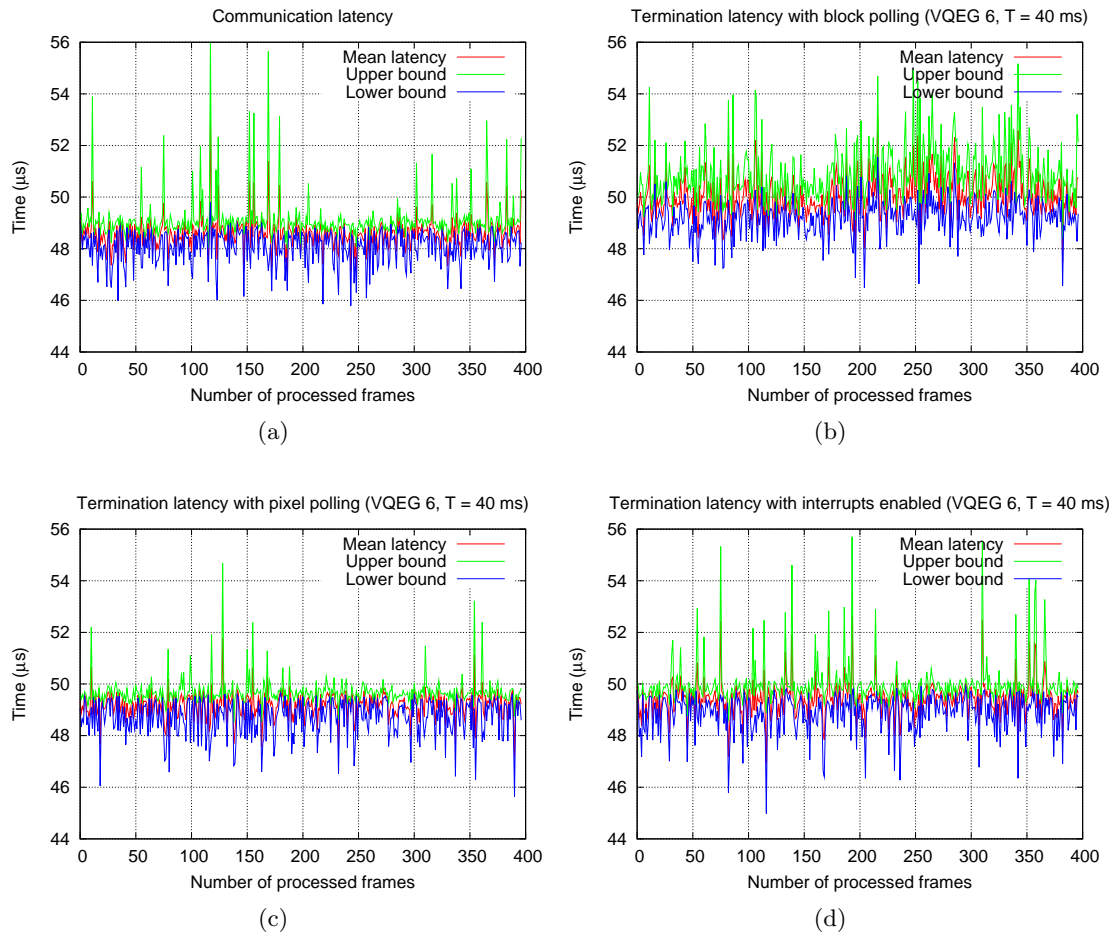


Figure 6.3: Figure (a) shows the average communication latency per frame measured over 20 separate runs, including a 95% confidence interval. Figure (b) shows the average termination latency when block polling is used, in (c) pixel polling is used and in (d) software interrupts are enabled. The frame deadline has been set to 40 ms, so the SVA is terminated during its final phase. Notice that the fluctuations caused by communication latency are generally larger than the relative differences between the results.

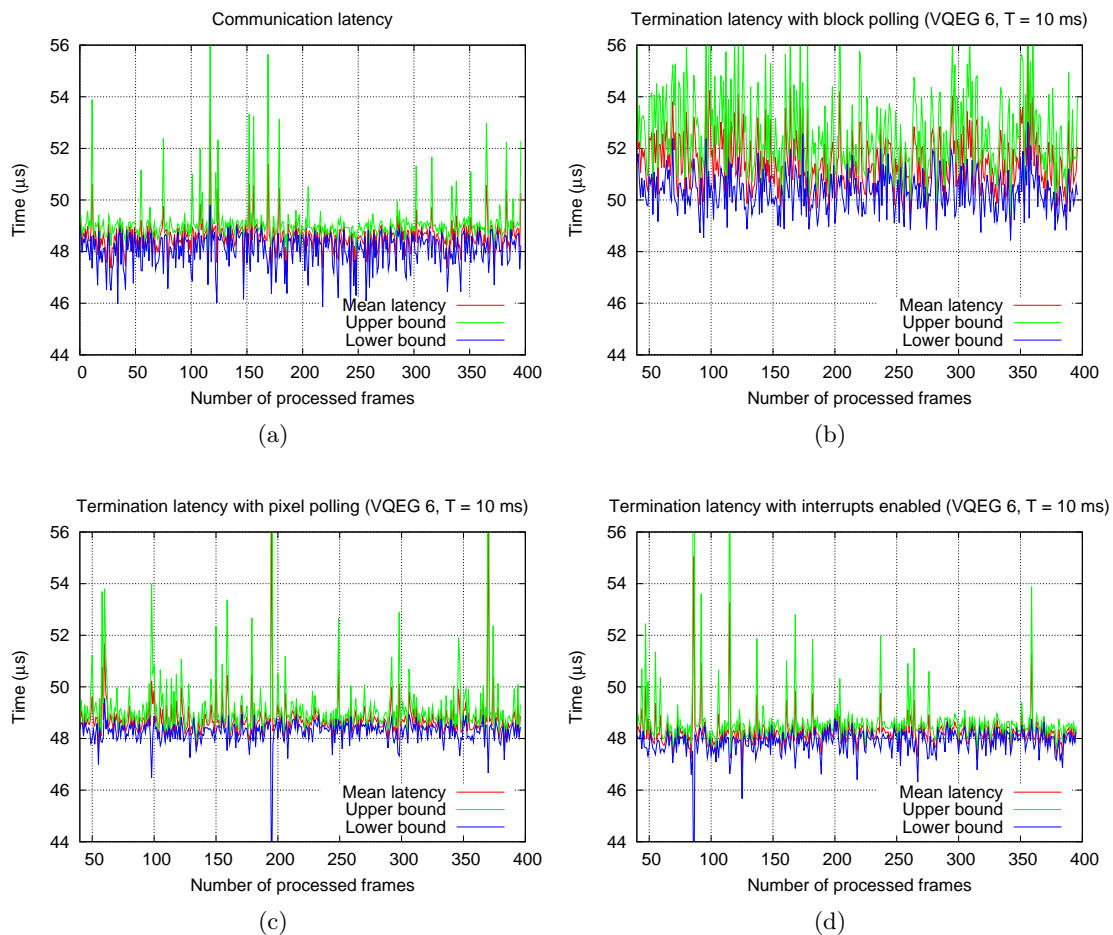


Figure 6.4: These four figures are obtained identically to Figures 6.3(a-d), with the exception that the frame deadline is set to 10 ms. This causes the SVA to get the preliminary terminate message during the analysis phase. Since the communication latency (a) is independent from the frame deadline, this figure is identical to Figure 6.3a.

6.4 Preliminary Termination Overhead

Periodically polling an incoming signal channel, as well as enabling software interrupts, result in computational overhead. In Section 6.4.1, the absolute overhead is presented for all three implementation variants. The absolute overhead is compared to the computation time of the scalable de-interlacer to compute the relative overhead in Section 6.4.2.

6.4.1 Absolute Overhead

The three variants for preliminary termination, as explained in Section 5.2, have been compared to a variant without polling and software interrupts, in order to quantify the absolute overhead introduced by every one of the three variants.

Although the processing time for a frame is content-dependent, the absolute overhead for polling is not. Therefore the measurements for overhead have only been done with video sequence 6 of the VQEG [23]. Every measurement has been executed 20 times, from which the 16 median values per frame have been extracted. From these 16 values, the mean has been computed. This approach has been taken to filter out any external influences on the processing time, which cause high fluctuations in the results.

The results can be found in Figures 6.5a-c. These results show that the overhead for pixel polling is on average between 1.2 and 1.3 ms. We expected the absolute overhead for block polling to be 32 times smaller than the overhead introduced by pixel polling, which would be about $1.25/32 = 0.04$ ms. In Figure 6.5 we can see that this is almost the case. The average overhead introduced by block polling turns out to be approximately 0.06 ms. The absolute overhead as a result of enabling software interrupts has been measured to be approximately 0.15 ms per frame. The mean absolute overhead per frame for all three variants has been summarized in Table 6.4.

Figures 6.5a-c show fluctuations that are relatively low compared to typical fluctuations in (content-dependent) video processing times. The fluctuations are expected to be a combination of measurement inaccuracies and, in the case of polling, the number of polling statements executed. The number of polling statements are expected to fluctuate, because the time needed for processing pixels or a block generally shows fluctuations.

6.4.2 Relative Overhead

Now that the absolute overhead for different implementations of the scalable de-interlacer has been measured, it makes sense to measure the worst-case execution time (WCET)

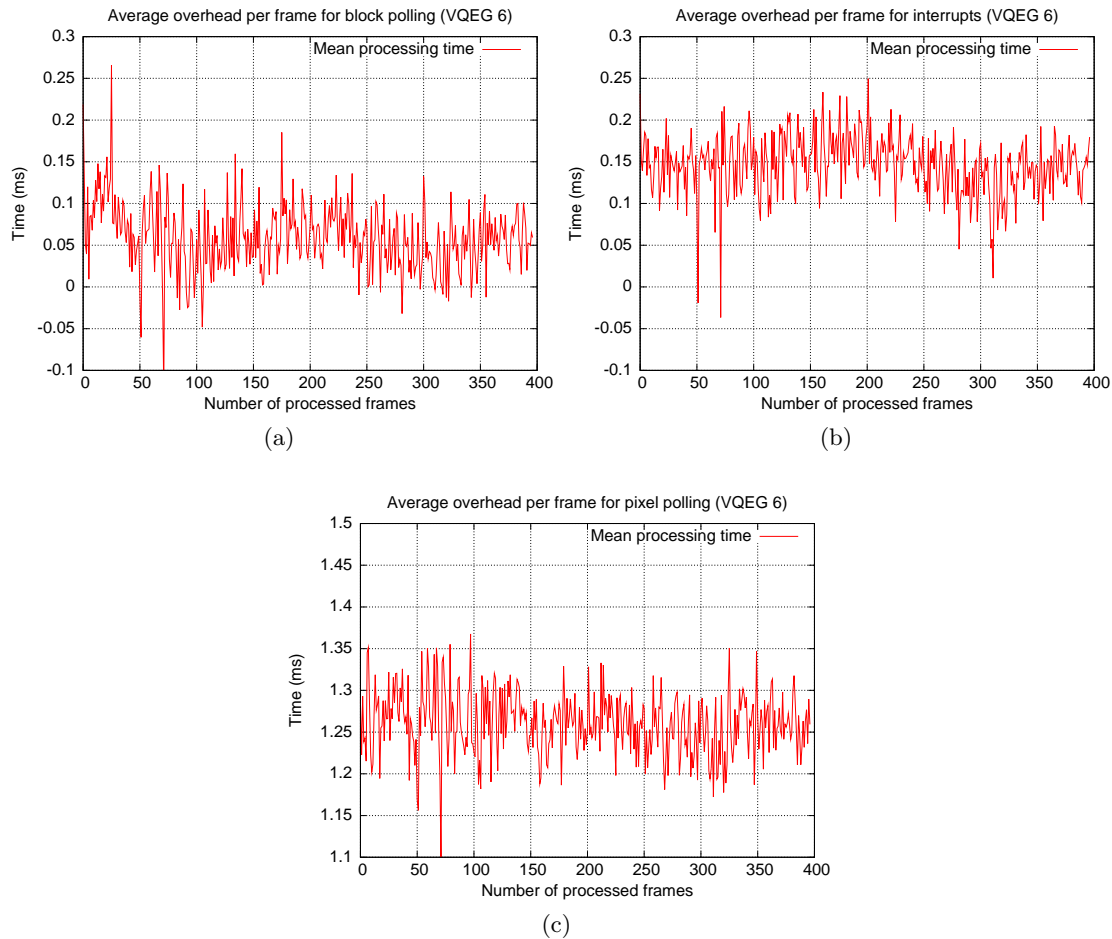


Figure 6.5: Polling a communication channel and enabling interrupts on an SPE causes overhead. Figure (a) shows the overhead for block polling, which is comparable to the overhead for enabling interrupts (b). Figure (c) shows the overhead for polling after processing four pixels in parallel. All measurements have been obtained with an instrumental approach, performed 20 times on VQEG 6. For every frame, the average has been computed using the median 16 values. Note the different scopes of the y-axes.

of the scalable de-interlacer, so that the relative overhead can be calculated. For video algorithms, it is generally not feasible to measure or calculate the worst-case execution time, due to the data-dependent nature of the algorithms. However, a fair indicator for the WCET can be obtained by measuring the execution time per frame for different video sources several times and by comparing the medians of different sources.

The computation time per frame of the scalable de-interlacer has been measured for VQEG sequences 3, 5 and 6, by taking the median of three consecutive runs. The results are plotted as red dots in Figures 6.7a-c. From these measurements, we can conclude that the time needed for the scalable de-interlacer to run to completion, approximates 50 ms per frame.

Using these results and the absolute overhead for the different implementations (Table 6.4), it is possible to show the relative overhead in relation to the processing time of the scalable de-interlacer. This relation is plotted in Figure 6.6. Since the processing time per frame approximates 50 ms, the relative overhead is less than 1% for both block polling and interrupts.

Figure 6.6 clearly shows that the approach with software interrupts results in a significant lower latency than with block polling, hence the solution with interrupts is the preferred one.

The motion adaptive interpolation function is the only component of the scalable de-interlacer for which the computation time is content dependent. To find out whether this function is responsible for the fluctuations visible in Figures 6.7a-c, we have measured the computation time for the motion adaptive interpolation function *independent* from the computation time for the entire scalable de-interlacer. The results have been plotted as green dots in Figures 6.7a-c. These figures show clearly that the motion adaptive interpolation function is the only content dependent component of the scalable de-interlacer and that this function is responsible for the majority of the time needed to run to completion.

The computation times for two content independent components, being the non-scalable basic function and the motion detection function, have also been measured. The results are presented in Table 6.5.

Variant	Absolute overhead
1. Block polling	0.059 ms \pm 0.004
2. Pixel polling	1.260 ms \pm 0.004
3. Interrupts enabled	0.146 ms \pm 0.004

Table 6.4: Average absolute overhead per frame, introduced by different implementation variants for preliminary termination, with a 95% confidence interval (VQEG sequence 6).

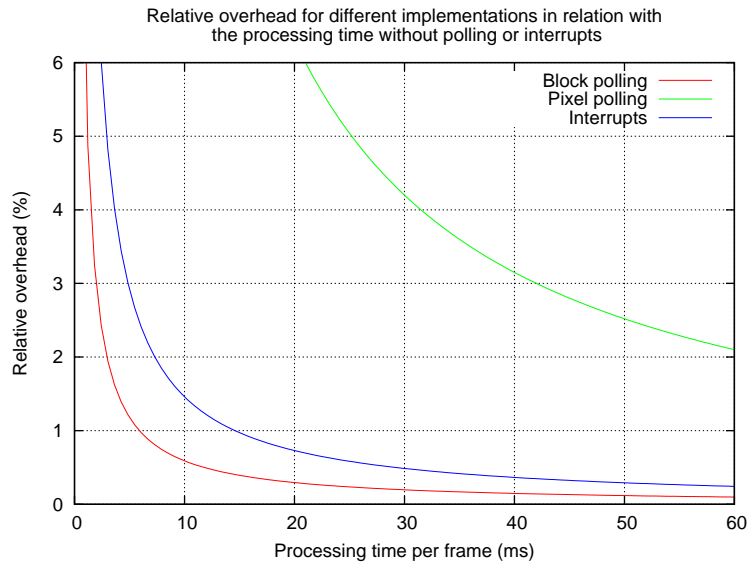


Figure 6.6: Relative overhead introduced by different implementation variants for preliminary termination as a function of the computation time granted to an SVA or the processing time needed by an SVA.

Computation Times per Block

Van den Heuvel [24] observed that there is a relation between the WCET for a block of 8×8 pixels and the position of the block in the priority queue on the Windows platform. The WCET for the first block in the queue approximates $20 \mu\text{s}$. The computation time per block decreases almost linearly to approximately $0 \mu\text{s}$ for the block in the end of the priority queue.

On the Cell platform, the WCET per block according to its position in the queue is expected to be more or less constant, for reasons explained in Section 4.4.2. Notice that on the Cell, blocks of 16×16 pixels are used. These blocks are comparable to the blocks used on the Windows platform, though, because the Cell processes four pixels in parallel.

The WCET per block, relative to its position in the queue, has been plotted in Figure 6.8. The WCET shows to be fairly constant for the first 1400 blocks and a slight dip can

Algorithm	VQEG 3	VQEG 5	VQEG 6
Basic de-interlacing	7.533 ms \pm 0.014	7.524 ms \pm 0.011	7.550 ms \pm 0.021
Motion detection	8.340 ms \pm 0.007	8.347 ms \pm 0.007	8.345 ms \pm 0.006

Table 6.5: Average execution times for two time-consuming, content independent parts of the priority processing algorithm, with a 95% confidence interval.

be observed for the final 220 blocks. This dip can be explained by the fact that in the last 220 blocks, the probability that all four pixels that are processed in parallel can be skipped is a little higher than usual. The blocks in the end of the queue are placed there, because a pretty low amount of motion is observed in these blocks. Since the processing time per block shows to be more or less constant, the processing time per frame as a function of the desired progress has a more or less linear shape. This function is plotted in Figure 6.9.

The observed worst-case execution time can either be caused by the processing time needed during the enhancement phase, or by the time needed for data transfers, because both computation and data transfer are performed in parallel. The total time the SPE has to wait for pending data transfers to complete during the enhancement phase has been measured. Since this total waiting time approximates zero, we can conclude that the observed WCET during the enhancement phase is caused by processing time.

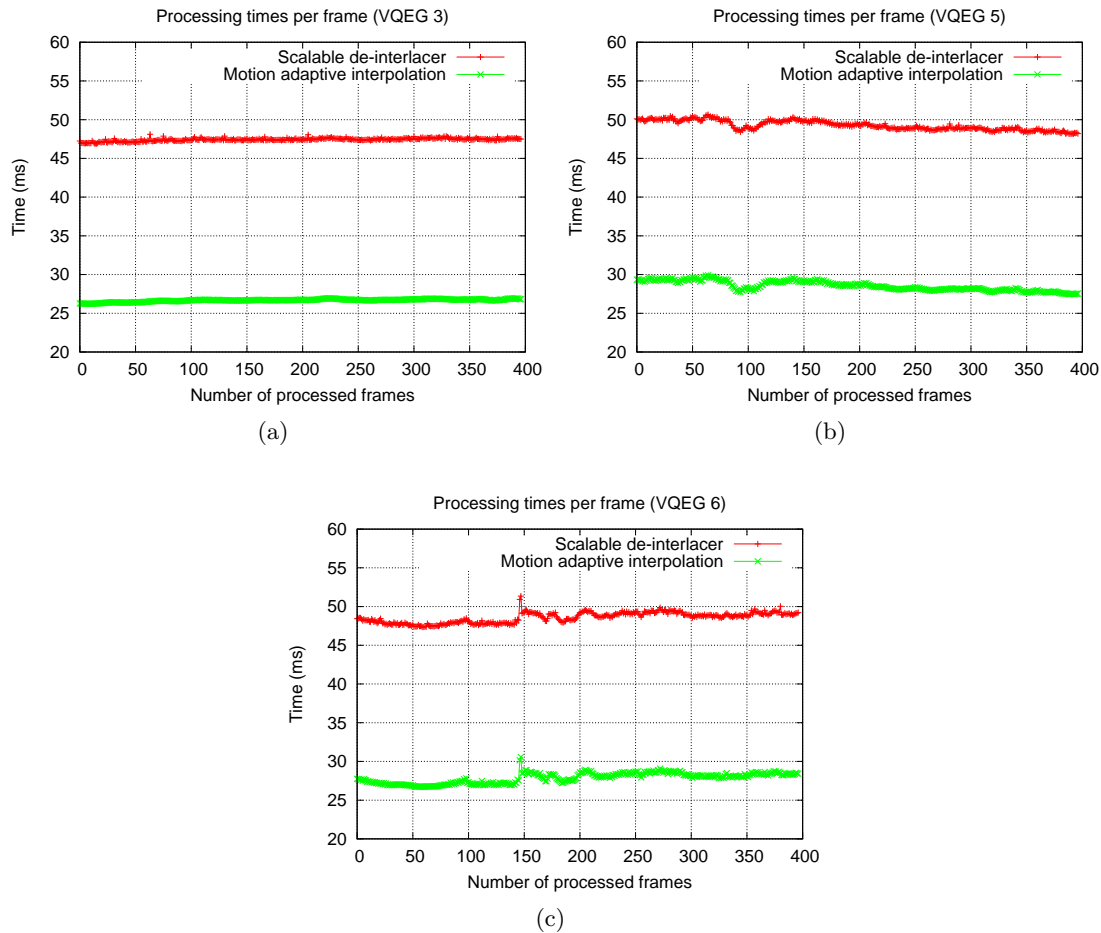


Figure 6.7: The total processing time needed by the scalable de-interlacer per frame versus the processing time needed by the motion adaptive interpolation function. The different graphs concern different input sequences. These graphs clearly show that the motion adaptive interpolation function is the only content dependent component of the scalable de-interlacer and that this function is responsible for the majority ($> 50\%$) of the processing time needed for the scalable de-interlacer to run to completion.

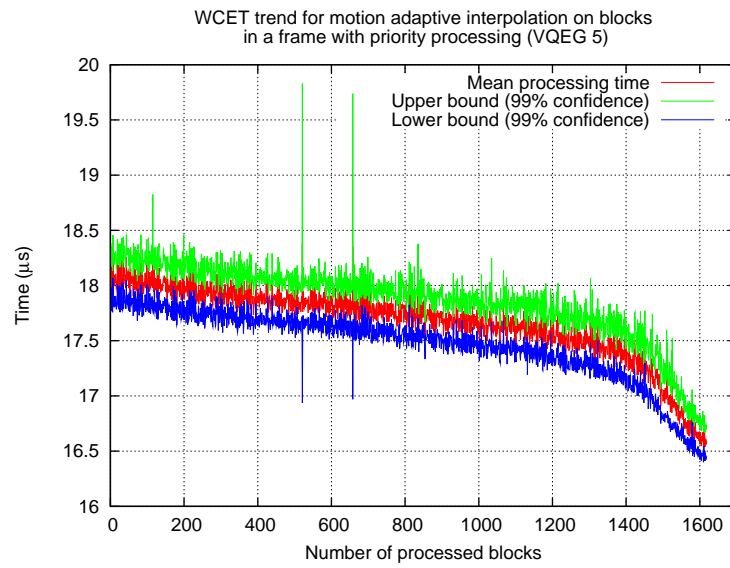


Figure 6.8: Worst-case execution time per block, relative to its position in the priority queue. Video sequence 5 is used as input.

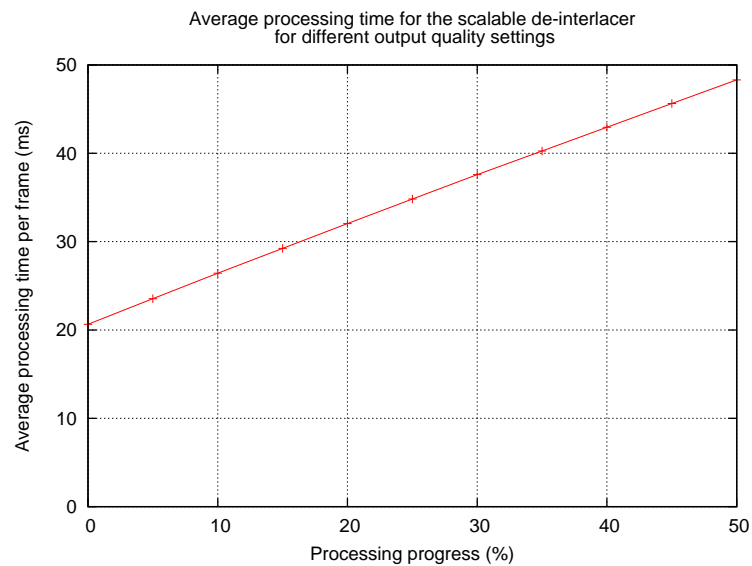


Figure 6.9: The average processing time per frame as a function of the desired processing progress.

6.5 Resource Allocation

On the general purpose platform, Van den Heuvel et al. [9] identified a couple of different implementation variants for resource allocation. These mechanisms all rely on the existence of operating system primitives, such as the ability to suspend or resume a task and the possibility to manipulate tasks' priorities. On the Cell platform, there is no operating system running on the SPE, but the SPE still has to be shared by two SVAs. Therefore, one mechanism, inspired by the way an operating system handles a context switch, has been conceived and designed.

The implementation of the resource allocation mechanism is explained in Section 5.3 and is visualized in Figure 5.3. In this figure, one can observe three different reasons that cause a delay in resuming the work on the SPE: (1) communication latency, (2) latency caused by the decision scheduler and (3) latency due to context switching. The communication latency is already covered in the section on preliminary termination. In this section, we focus on the latency caused by the decision scheduler and the latency due to context switching.

6.5.1 Decision Scheduler Latency

The average latency caused by the decision scheduler has been measured by Schiemenz on a general purpose platform using dummy SVAs [8]. In this report, we measure the average DS latency on the Cell processor for the decision scheduler controlling two scalable de-interlacers. The decision scheduler is set to use the reinforcement learning strategy, because the other two strategies result in a constant and negligible latency. This latency is expected to be more or less constant over time. The average DS latency is expected to be larger when the number of time slots per frame period increases, as a result of a growing history table used by the reinforcement learning policy.

Figure 6.10 show the decision scheduler latencies per time slot, for different amounts of time slots per frame period. Figure 6.10a shows the latency for a frame period consisting of 10 time slots of 4 ms each, Figure 6.10b shows this for 20 time slots of 2 ms each and Figure 6.10c shows it for 40 time slots of 1 ms each.

First of all, one can observe that the latency for the first decision of every frame is always about a factor two larger than the average latency for the other time slots. Schiemenz [8] explains that this behavior is likely caused by initialization of the reinforcement learning scheme.

Secondly, an interesting property is the peaking latency for the decision after four (Figure 6.10a), six (Figure 6.10b) or ten (Figure 6.10c) time slots. This behavior is due to the following. The decision scheduler generally allocates the first time slot to one of the SVAs

and the following consecutive x time slots to the other SVA, where x is the number of time slots needed for the motion detection function to complete. After this function has completed, the DS faces a decision that takes some more time than usual. The time needed for the motion detection function to complete is between 8.3 and 8.4 ms (Table 5.2). Therefore, in Figure 6.10a, the motion detector needs three time slots of 4 ms each (i.e. 8 – 12 ms) to complete. In Figure 6.10b, it completes after five time slots of 2 ms each (i.e. 8 – 10 ms) and in Figure 6.10c after nine time slots of 1 ms each (i.e. 8 – 9 ms).

Finally, one can observe that the average latency is not exactly constant, but for a larger number of time slots, a more constant latency can be observed. One can also observe that the average DS latency is higher when the number of time slots per frame period increases.

6.5.2 Latency due to Context Switching

The latency due to context switching has not been measured in this report, because it was thought to be impossible to measure. In retrospect, this latency can probably be quantified by measuring the elapsed time between the reception of a ‘resume’ message on the SPE and the start of processing the first pixel. Instead of measuring the latency due to context switching, a comparison is made between the overall progress that is achieved by one SVA running in isolation versus the overall progress achieved by two SVAs controlled by the decision scheduler. This comparison has not been performed in an extensive and exact way, therefore the results are not presented in this section. Rather, the results give an indication on the average latency due to context switching and are therefore presented in Appendix D.

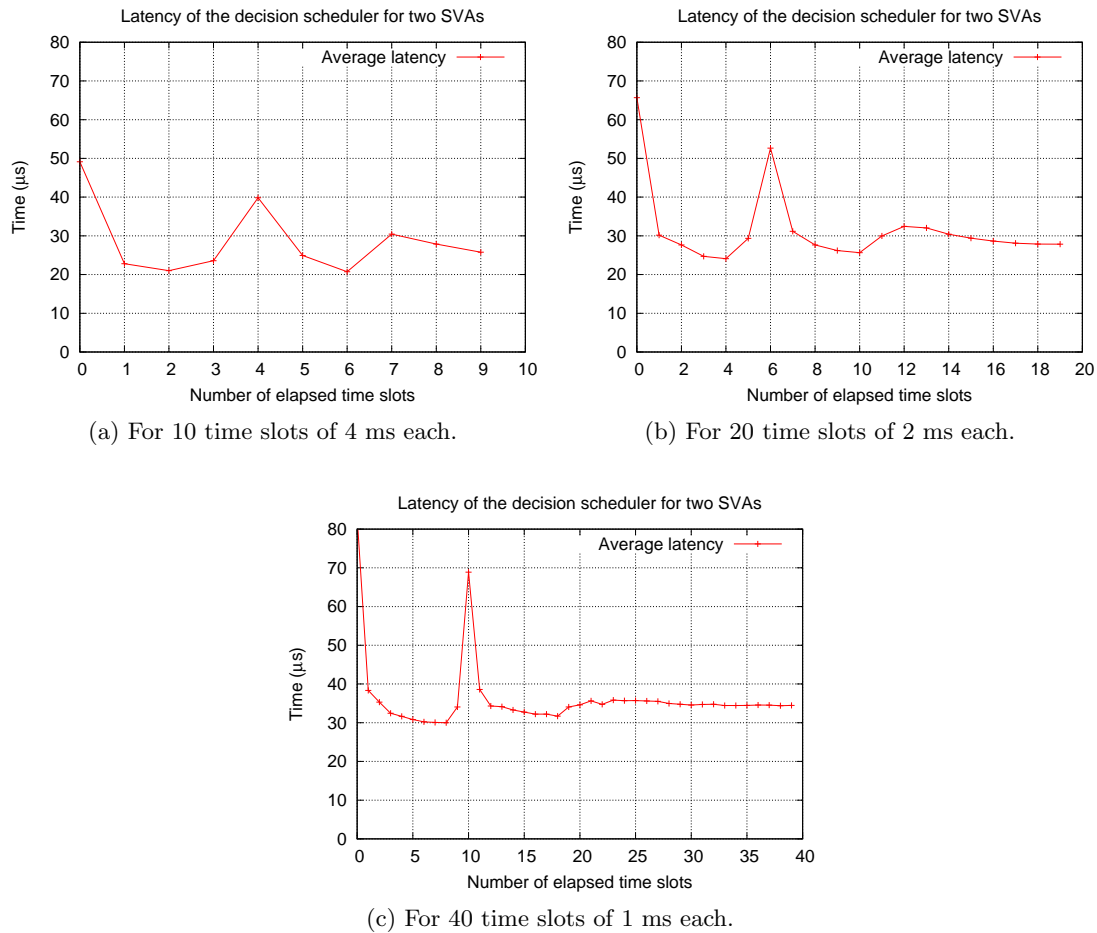


Figure 6.10: These graphs show the mean time needed for the decision scheduler to decide which SVA to allocate the time slot to, for different amounts of time slots per frame period. The mean latency per time slot is calculated from median latencies, which are obtained by taking per time slot the median of the latency for the current time slot, the same time slot one run later and the same time slot one run before, for 20 runs. The system consists of two scalable de-interlacers, operating on distinct but identical input streams (VQEG 6).

6.6 Monitoring

The decision scheduler implementing the reinforcement learning mechanism relies on the accurate allocation of time slots to the priority processing algorithms and relies on the accurate reporting of progress the algorithms have achieved. An implementation for the monitoring mechanism is easily created using the measurement results for preliminary termination and the mechanisms as provided by the Cell.

Whenever the decision scheduler sends a message towards the SPE to activate one of the SVAs, it starts inspecting its time base register (see Section 3.4 for details) with a busy-waiting loop. Whenever a predefined amount of time has elapsed, a preliminary terminate message is sent to the SPE. The SPE acknowledges this message and puts the progress value into this acknowledgement message.

This mechanism ensures an allocation of time slots as accurately as possible on the Cell. Therefore, there are no research questions formulated for the monitoring mechanism, neither is any quantitative analysis done.

Chapter 7

Conclusions

Scalable video algorithms using the principle of priority processing guarantee real-time performance on programmable platforms, even with limited resources [9]. In this thesis, a single priority processing algorithm has been shown to achieve real-time performance on the Cell platform, a platform suitable for consumer electronics. When multiple priority processing algorithms share a certain platform with limited resources, these algorithms start to compete for resources. A decision scheduler (DS) has been created to distribute the available resources over competing priority processing algorithms [8]. Preliminary measurements (see Appendix D) show that real-time performance has been achieved for two scalable de-interlacers sharing one signal processing core (SPE) on the Cell. The implementation of an application consisting of a decision scheduler and multiple priority processing algorithms is based on three mechanisms for dynamic resource allocation, identified by Van den Heuvel et al. [9]. The implementation variants for all three mechanisms for dynamic resource allocation rely on one shared principle, being efficient and reliable communication.

We have identified the most efficient implementation for communication as one in which an incoming signal raises a software interrupt at the SPE and in which the PowerPC periodically polls for the presence of an outgoing message at the outgoing mailbox of the SPE. Since messages and signals are guaranteed to be delivered on the Cell platform and since raised interrupts are always handled by an interrupt service routine, this implementation has shown to be a reliable implementation for communication as well.

Regarding the **preliminary termination** mechanism, the implementation variant relies on the chosen principle for efficient and reliable communication solely. The latency for preliminary termination using this principle has shown to be approximately 50 μ s and the computational overhead is less than 1%.

The **resource allocation** mechanism relies on three principles, one of these is the

principle for communication, that has already been described. The other two are a principle for performing a context switch on a core that does not run an operating system and a principle for making an accurate decision for allocating resources to algorithms using a minimal amount of time. The latter is handled by the decision scheduler and is subject of research by Schiemenz [8]. Although it has not been a core subject of this report, the time needed for making a decision has been investigated and is found to be in the order of $35 \mu\text{s}$, for time slots with a fixed size of 1 ms.

A lightweight method for performing a context switch without making use of an operating system has been designed and has been implemented on one SPE. Unfortunately, no concrete measurements regarding the time needed to perform such a context switch have been performed. An indication for the total overhead for a system consisting of a decision scheduler and two priority processing algorithms is given (Appendix D). This is done by comparing the achieved progress by two SVAs controlled by the decision scheduler, with the expected progress as a result of one SVA running in isolation. It has been calculated that in the current implementation, approximately $390 \mu\text{s}$ per time slot of 1 ms is consumed by the context switching mechanism and probably by unknown factors. Since this value is expected to be in the order of $10 \mu\text{s}$, future research can be done towards an implementation for context switching that has better performance.

With respect to the **monitoring** mechanism, a straightforward implementation for allocation of time slots with an accuracy of 12.5 ns and an implementation for accurate reporting of progress by the SVAs to the decision scheduler has been described.

7.1 Future Work

The identified future work is subdivided into the application level and the system level for clarity.

7.1.1 Application Level

On the application level, two improvements regarding performance have been identified. These concern (1) the mapping of components of a scalable video algorithm on certain cores on a multi-core platform and (2) combining the basic and analysis phases in a priority processing algorithm.

Mapping of Components onto Cores

At the time of implementing the scalable de-interlacer on the Cell platform, it was decided to implement certain components (being a noise-estimation function and a sorting algorithm) on the PowerPC rather than on the SPE. Although this decision has been made from a performance point-of-view, because these components probably have better performance on a general purpose core, the decision could have been an unfortunate one since it introduces extra communication between the cores. The noise-estimation function can probably be mapped onto the SPE without much effort. Sharma et al. [21] have suggestions towards sorting algorithms on a Cell processor.

Combine the Basic and Analysis Phases

The second suggestion towards future work is to combine the computational parts of the non-scalable basic function and the motion detection function. The direct advantage of this approach is that the number of data transfers between main memory and the local storage drastically decreases, which has a positive effect on the overall performance of the priority processing application.

Another advantage of using this technique is that the function of output quality in terms of processing time (like Figure 6.9), behaves more like a linear function. This results in smaller fluctuations in output quality over time, which results in a better overall perceived quality.

A point for discussion is that the motion detection function, which is part of the optional functionality of a priority processing algorithm, is shifted towards the mandatory part of the algorithm. This contradicts with a property of the priority processing principle, stating that the mandatory part of the algorithm should always be as small as possible. On the other hand, Figure D.1 (Section 6.5) shows that the amount of available time is on the Cell platform always sufficient for the non-scalable basic function and motion detection function to run to completion.

7.1.2 System Level

On the system level, performance can be gained by modifying the DS and priority processing algorithms to run in parallel rather than in an alternating way and by making better use of the buffers that are needed by the scalable de-interlacer. Finally, a different direction can be taken by using multiple SPEs on the Cell platform.

Run the DS and algorithms in parallel

The decision scheduler should be programmed to work in parallel with the priority processing algorithm, rather than in an alternating way. The DS should decide which algorithm to allocate a time slot to, before it sends a preemption signal to the running SVA. For this, the strategy of reporting progress to the DS needs to be changed, by e.g. programming the SVA to periodically post the achieved progress into a shared variable in main memory. Afterwards, the SPE could be programmed such that a priority processing algorithm is not preempted if it is granted the next time slot, which allows the use of gain-time.

Improve the Buffer Usage

Two more improvements can be made concerning the implementation of the principle for context switching. The first improvement is that if a buffer is known to contain data that is ready to be processed, such a buffer does not need to be refilled after the algorithm is reactivated due to a context switch. Second of all, the current implementation uses two buffers per algorithm to enable double buffering. This can be improved by using one buffer per algorithm, plus one buffer that is shared among the algorithms.

Towards Using Multiple SPEs

Finally, research can be done towards using more of the available SPEs to run scalable video algorithms. In that case, a decision needs to be made either to run the same algorithm(s) on all available SPEs, with every SVA operating on parts of the video frame, or to pipeline the entire frame through the set of available SPEs, where each SPE is responsible for a specific task.

References

- [1] IBM, “Power architecture.” URL, Accessed March 2010. <http://www.ibm.com/technology/power/>.
- [2] M. Scarpino, *Programming the Cell Processor: For Games, Graphics, and Computation*. Prentice-Hall, 2008.
- [3] C. Hentschel, R. J. Bril, Y. Chen, R. Braspenning, and T.-H. Lan, “Video quality-of-service for consumer terminals – a novel system for programmable components,” in *IEEE Transactions on Consumer Electronics (TCE)*, vol. 49, pp. 1367 – 1377, November 2003.
- [4] C. Hentschel, R. J. Bril, and Y. Chen, “Video quality-of-service for multimedia consumer terminals – an open and flexible system approach,” in *IEEE 2002 Int. Conf. on Comm., Circuits and Systems and West Sino Expositions*, pp. 659 – 663, July 2002.
- [5] C. Hentschel, M. Gabrani, K. Van Zon, R. J. Bril, and L. Steffens, “Scalable video algorithms and quality-of-service resource management for consumer terminals,” in *Proc. Int. Conf. on Consumer Electronics (ICCE)*, pp. 338 – 339, June 2001.
- [6] G. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2005.
- [7] C. Hentschel and S. Schiemenz, “Priority-processing for optimized real-time performance with limited processing resources,” in *Proc. Int. Conf. on Consumer Electronics (ICCE)*, pp. 1 – 2, January 2008.
- [8] S. Schiemenz, “Echtzeitsteuerung von skalierbaren priority-processing algorithmen,” in *Tagungsband ITG Fachtagung – Elektronische Medien*, pp. 108 – 113, March 2009.
- [9] M. M. H. P. van den Heuvel, R. J. Bril, S. Schiemenz, and C. Hentschel, “Dynamic resource allocation for real-time priority processing applications,” in *Proc. Int. Conf.*

- on *Consumer Electronics (ICCE)*, pp. 67 – 68, January 2010.
- [10] National Institute of Standards and Technology, “Prefixes for binary multiples.” URL, Accessed March 2010. <http://physics.nist.gov/cuu/Units/binary.html>.
- [11] C. C. Wüst, L. Steffens, W. F. J. Verhaegh, R. J. Bril, and C. Hentschel, “QoS control strategies for high-quality video processing,” *Real-Time Systems*, vol. 30, pp. 7 – 29, May 2005.
- [12] N. Audsley, A. Burns, R. Davis, and A. Wellings, *Integrating unbounded software components into hard real-time systems*, ch. 5, pp. 63 – 86. Kluwer Academic, 1995.
- [13] J. Liu, K.-J. Lin, W.-K. Shih, A. C. Yu, J.-Y. Chung, and W. Zhao, “Algorithms for scheduling imprecise computations,” *Computer*, vol. 24, pp. 58 – 68, 1991.
- [14] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.
- [15] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, “Cell broadband engine architecture and its first implementation: a performance view,” *IBM J. Res. Dev.*, vol. 51, pp. 559–572, 2007.
- [16] IBM, “Cell broadband engine programming tutorial,” 2008. Version 3.1.
- [17] Y. Dou, L. Deng, J. Xu, and Y. Zheng, “DMA performance analysis and multi-core memory optimization for SWIM benchmark on the cell processor,” in *Proc. IEEE Int. Symp. on Parallel and Distributed Processing with Applications*, pp. 170 – 179, December 2008.
- [18] IBM, “Cell broadband engine programming handbook,” 2009. Version 1.12.
- [19] S. Schiemenz and C. Hentschel, “De-interlacing using priority processing for guaranteed real time performance with limited processing resources,” in *Proc. Int. Conf. on Consumer Electronics (ICCE)*, pp. 1 – 2, January 2009.
- [20] C. Hentschel, “High quality noise insensitive motion detector using one field memory,” in *IEEE Transactions on Consumer Electronics (TCE)*, vol. 42, pp. 696 – 704, August 1996.
- [21] D. Sharma, V. Thapar, R. A. Ammar, S. Rajasekaran, and M. Ahmed, “Efficient sorting algorithms for the cell broadband engine,” in *Proc. IEEE Symp. on Computers and Communications*, pp. 736 – 741, July 2008.
- [22] IBM, “C/C++ language extensions for cell broadband engine architecture,” 2006. Version 2.2.1.

- [23] Video Quality Experts Group, “Test sequences.” URL, Accessed January 2010. ftp://vqeg.its.bldrdoc.gov/SDTV/VQEG_PhaseI/TestSequences/.
- [24] M. M. H. P. van den Heuvel, “Dynamic resource allocation in multimedia applications,” Master’s thesis, Eindhoven University of Technology, July 2009.
- [25] Mentor Graphics, “Nucleus RTOS.” URL, Accessed October 2009. http://www.mentor.com/products/embedded_software/nucleus_rtos/.

Appendix A

Commonly used SPE Instructions

The SPE offers a range of functions that operate on 128-bit vectors. These are all explained in great detail in IBM's Programming Handbook [18] and to some extent in Scarpino's book on programming the Cell processor [2]. The purpose of most SPE functions should speak for itself, however, a few of them might be somewhat cryptic.

In this appendix, a few functions that are used throughout the code listings in this report and that are not self-explaining will be explained.

A.1 The *spu_splats* Function

This function takes a scalar as input and replicates it into a 128-bit vector, which it returns. The input can be of any type of which vectors can be handled by the SPE, including integers, floats and characters.

A.2 The *spu_sel* Instruction

This function takes two vectors v_1 and v_2 of the same type and one 'filter' vector that should be of an unsigned type. It returns a vector of the type equal to the first two input vectors. The function observes all 128 bits in the input vectors in isolation. If bit i ($0 \leq i < 128$) of the filter vector equals 0, bit i in vector v_1 serves as bit i in the output. Analogous, if bit i of the filter vector equals 1, bit i of vector v_2 is copied into bit i of the vector that is returned.

Appendix B

Branch Intensive Code

This appendix presents the outline of the motion adaptive interpolation function, as used in the scalable de-interlacer, twice. Code Listing B.1 presents this function in scalar code as implemented on a general purpose processor, Listing B.2 shows the function as implemented on the Cell in vector code.

The vector code makes high usage of the `spu_sel()` instruction, an explanation of this operation is given in Appendix A.

```

1 int interpolate_pixel (int y, int x)
2 {
3     if ((0 < motion[y][x]) && (motion[y][x] < 200)) {
4         // Compute vp1, indicating presence or absence of line-flicker
5         // Compute vp2, indicating presence or absence of serration
6
7         if ((vp1 <= 0) && (vp2 <= 0)) {
8             // Relative small interpolation function
9         }
10        if ((vp1 <= 0) && (vp2 > 0)) {
11            // Relative small interpolation function
12        }
13        if ((vp1 > 0) && (vp2 <= 0)) {
14            // Relative small interpolation function
15        }
16        if ((vp1 > 0) && (vp2 > 0)) {
17            //
18            // Relative large interpolation function,
19            // which includes an iteration
20            //
21        }
22    }
23    else {
24        // Relative small function
25    }
26
27    // Variable 'result' contains the value of the interpolated pixel
28    return result;
29 }

```

Listing B.1: The outline of the motion adaptive interpolation function in scalar code.

```

1 vector signed int interpolate_pixels (vector signed int in_vec)
2 {
3     vector signed int p_final , p_intermediate;
4     vector unsigned int filter;
5     p_final = spu_splats(0);
6
7     if (there exists component z in in_vec for which:  $0 < z < 200$ ) {
8         // Relative small interpolation function
9         p_final = spu_sel (p_final , p_intermediate , filter);
10
11        // Relative small interpolation function
12        p_final = spu_sel (p_final , p_intermediate , filter);
13
14        // Relative small interpolation function
15        p_final = spu_sel (p_final , p_intermediate , filter);
16
17        //
18        // Relative large interpolation function ,
19        // which includes an iteration
20        //
21        p_final = spu_sel (p_final , p_intermediate , filter);
22    }
23    else {
24        //Relative small function that computes p_final
25    }
26    return p_final;
27 }

```

Listing B.2: The outline of the motion adaptive interpolation function in vector code.

Appendix C

Resource Management on the SPE

Code Listing C.1 presents the implementation of the main routine running on the SPE, which is responsible for allocating the SPE to one of the SVAs.

Allocating the SPE to one of the SVAs is done after receiving a signal, which contains the SVA to run as one of the parameters, by calling the appropriate subroutine. Every subroutine is responsible for storing the block it has processed last in the global `block` variable. Preempting a running SVA is done using software interrupts, that invoke the `interrupt_service` routine, in which a non-local jump is taken to jump out of the subroutine.

```

1 jmp_buf jump_buffer;
2 int running, function [NUMSVAS], block [NUMSVAS];
3
4 int spe_main (unsigned long long argp)
5 {
6     get_buffer_addresses(argp);
7     read_signal(&parameters);
8
9     while (parameters.task != ABORT_ALL_SVAS)
10    {
11        running = 1;
12        if (setjmp (jump_buffer)) { goto epilog; }
13
14        if (function [parameters.sva] == BASIC_FUNCTION) {
15            basic_function ();
16        }
17        if ((function [parameters.sva] == MOTION_DETECTION) && running) {
18            spu_ienable ();           // enable interrupts
19            motion_detection ();
20            spu_idisable ();          // disable interrupts
21        }
22        if ((function [parameters.sva] == IMAGE_ENHANCEMENT) && running) {
23            spu_ienable ();
24            motion_adaptive_interpolation ();
25            spu_idisable ();
26        }
27
28        epilog:
29            wait_for_pending_dmas_to_complete ();
30            send_acknowledgement ();
31            read_signal(&parameters);
32        }
33
34        wait_for_pending_dmas_to_complete ();
35        send_acknowledgement ();
36        return 0;
37    }
38
39 void interrupt_service (void)
40 {
41     acknowledge_event ();
42     empty_signal_channel ();
43     running = 0;
44
45     spu_sync_c ();
46     longjmp(jump_buffer, 1);
47 }

```

Listing C.1: The main routine that runs on the SPE, which is responsible for allocating the SPE to one of the SVAs.

Appendix D

Latency due to Context Switching

Preliminary measurements have been performed to find an indication of the delay caused by performing a context switch on the SPE. These measurements have not been performed in an extensive way and rely on an assumption that may not be true. It has been assumed that the achieved progress over time by the scalable de-interlacer is approximately the same for different video sequences. Therefore, the measurements are presented in this appendix, rather than in the chapter on evaluation of the mechanisms.

First, the overall performance for a single component setup, i.e. one priority processing algorithm running in isolation, has been measured. The frame period has been set to 40 ms, because this causes the SVA to be preliminary terminated during the final phase. From Table 6.5, it is known that a single SVA does not achieve any progress for approximately the first 16 ms, since this time is needed for the non-scalable basic function and content-analysis function. From Figure D.1a, one can observe that during the remaining 24 ms, a progress of approximately 35% is achieved by a single SVA. Furthermore, from Figure 6.9 we know that there is a linear relation between the achieved progress and the processing time.

For two SVAs that are controlled by the decision scheduler, the time needed for the basic and content-analysis functions is known (Table 5.2) to be approximately 25 ms. This excludes the time needed for communication between the SPE and PowerPC and the time needed for the DS to take a decision at the end of every time slot. As a result, 15 ms remain for the two SVAs to enhance the output. The expected overall progress that can be achieved during 15 ms is approximately and at most $35\% \times 15/24 \text{ ms} = 22\%$.

The actual overall progress for two SVAs has been measured with two different video streams. As a result, the comparison between one SVA and two SVAs is not a fair comparison. But since the progress achieved for sequence 5 is assumed to approximate the progress for sequence 6, it provides an indication. Every frame period is of length

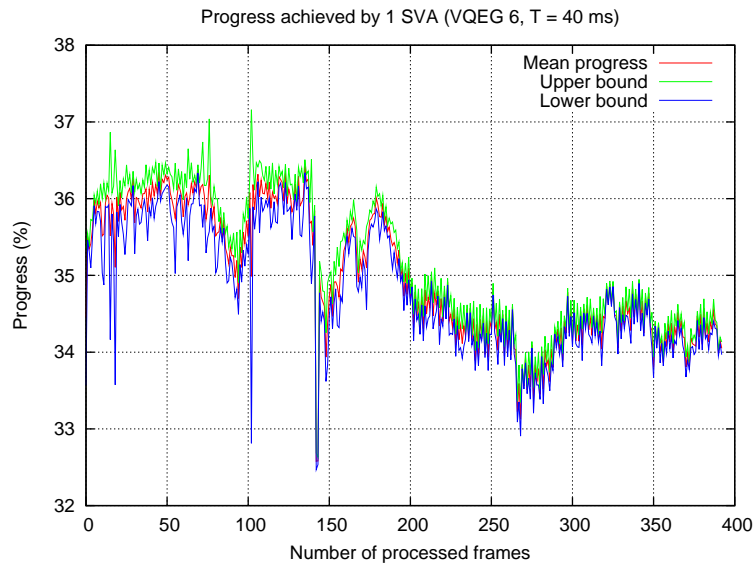
40 ms and has been divided into 40 time slots of 1 ms each. The results are plotted in Figure D.1b. For these measurements, the policy of the DS has been set to round-robin. This is done because round-robin guarantees a similar allocation of time slots to algorithms for different runs, hence it allows for taking the average progress per frame over a number of runs.

The results in Figure D.1b show that the total overall progress is about 11%, which is only half the expected amount. So for every time slot of 1 ms, approximately 500 μs are wasted on overhead. The overhead is a combination of (1) time needed for communication, (2) the latency of the decision scheduler and (3) overhead because of context switching.

The time needed for communication is the time needed for preliminary terminating an algorithm plus the time needed for sending the new decision to the SPE, i.e. the time needed for one round-trip communication between the cores and one single communication towards the SPE. This time is expected to be approximately $50 \mu\text{s} + 0.5 \times 50 \mu\text{s} = 75 \mu\text{s}$ (Section 6.3). The DS latency is approximately 35 μs per time slot (Figure 6.10).

This leaves $500 - 75 - 35 = 390 \mu\text{s}$ being consumed for performing the actual context switch, i.e. changing the running SVA and reloading buffers. Here is currently a huge amount of time wasted, since one would assume the time needed for a context switch to be around the time to process one block, i.e. approximately 18 μs .

Although the plots in Figure D.1 can not be compared directly from a resource allocation point-of-view (one SVA running with half the frame period and all SVAs operating on identical video sequences would probably have been better to support a comparison), these plots have been created out of curiosity, with the desire to give an indication of the achieved progress over time for two algorithms with different loads.



(a)



(b)

Figure D.1: Figure (a) shows the average progress achieved by one SVA running in isolation, measured over 20 runs. The bounds are for a 95% confidence interval. Figure (b) shows the average progress achieved by two SVAs, controlled by the DS with a round-robin policy. The SVAs operate on different video streams (videos 5 and 6 of the VQEG), hence the differences in achieved progress. Again, the values have been averaged over 20 runs.

Appendix E

The PlayStation 3

There exist two ranges of systems based on the Cell architecture. One range is the IBM BladeCenter of which the products cost several thousands of US dollars. The other range is the Sony PlayStation 3, that is widely available for just a couple of hundred US dollars. The IBM BladeCenter gives access to two Cell processors, resulting in a total of 16 SPEs. The PlayStation 3 carries one Cell processor, with seven active SPEs, of which one is reserved for the operating system. The PlayStation 3 is perfectly capable of running a distribution of GNU/Linux, namely Yellow Dog Linux.

Yellow Dog Linux is not a real-time operating system (RTOS) so there are no guarantees for response times of processes. There is one RTOS tailored for the Cell platform, namely Mentor Nucleus [25]. Currently it is unknown whether this RTOS runs on the PlayStation 3, because it is developed especially for the IBM BladeCenter range of products. Since this thesis is about porting an application with soft real-time requirements to the Cell platform, it was decided to use Yellow Dog Linux as an operating system.

The specifics of the PlayStation 3 that has been used, are summarized in the table below.

CPU frequency	3192 MHz
Main memory	210 MiB
Operating system	Yellow Dog Linux 6.1
Kernel	2.6.28-0.ydl61.1
GCC version	4.1.1

Table E.1: Specifics for the PlayStation 3 that has been used.