Dynamic resource allocation for real-time priority processing applications

Dynamic Resource Allocation for Real-Time Priority Processing Applications

Martijn M.H.P. van den Heuvel, Reinder J. Bril, Stefan Schiemenz and Christian Hentschel, *Member IEEE*

Abstract — *Flexible signal processing on programmable platforms are increasingly important for consumer electronic applications and others. Scalable video algorithms (SVAs) using novel priority processing can guarantee real-time performance on programmable platforms even with limited resources. Dynamic resource allocation is required to maximize the overall output quality of independent, competing priority processing algorithms that are executed on a shared platform. In this paper, we describe basic mechanisms for dynamic resource allocation and compare the performance of different implementations on a general purpose platform*[1].

Index Terms — priority processing, scalable video algorithms, dynamic resource allocation, preliminary termination, monitoring.

I. INTRODUCTION

Consumer electronic products are increasingly becoming more open and flexible, which is achieved by replacing dedicated single-function hardware components by software components running on programmable platforms. This trend aims at versatile, future proof, upgradable products and reduced time-to-market for new features. To enable cost-effective media processing in software, scalable video algorithms (SVAs) have been conceived which allow trading resource usage against output quality at the level of individual frames, and complemented with dynamic resource management techniques [1]. The additional advantage of the latter approach is the ability to add functionality on already fully loaded platforms, albeit at lower quality of some functionalities, and to reuse hardware and software modules to support product families [2].

The novel principle of priority processing provides optimal real-time performance for SVAs on programmable platforms even with limited system resources [3]. According to this principle, SVAs provide their output strictly periodically and processing of images follows a priority order. Hence, important image parts are processed first and less important parts are subsequently processed in a decreasing order of importance. After creation of an initial output by a basic function, processing can be terminated at an arbitrary moment in time, yielding the best output for given resources. This characteristic behavior of priority processing is shown in Figure 1.

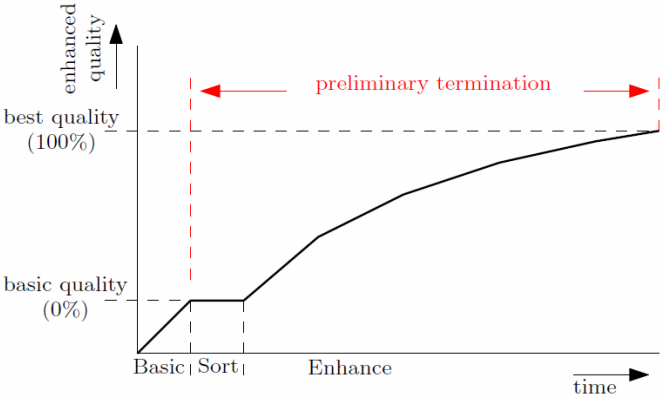

Fig. 1. Priority processing, as a function of time consumption versus output quality, can be divided in three phases: 1) produce a basic output at the lowest quality level; 2) order image content according to decreasing importance; 3) enhance the output quality by processing the content in the order of importance.

To distribute the available resources, i.e. CPU-time, among competing, independent priority processing algorithms constituting the functionality of an application, a decision scheduler has been developed [4]. The decision scheduler aims at maximizing the overall output quality of the application by maximizing the total relative progress of the algorithms on a frame-basis. The relative progress of an algorithm is defined as the fraction of the number of already processed blocks and the total number of blocks to be processed in a frame. The decision scheduler divides the available resources (processing time) within a period into fixed-sized quanta, termed *time-slots*, and dynamically allocates these time-slots to the algorithms. The activated SVA has access to all system resources for the duration of this time-slot. Strategies for allocating time-slots have also been addressed in [4], including round robin (RR), where time-slots are allocated in a strictly alternating way, and a strategy based on *reinforcement learning* (RL), where time-slots are allocated based upon a self-learning algorithm.

A. Problem Description

We consider multimedia applications that are composed of a set of competing, independent priority processing algorithms and a decision scheduler, where the algorithms are running on a single processor and have the entire processor at their disposal. To maximize the overall output quality of such an

[1] Martijn M.H.P. van den Heuvel and Reinder J. Bril are with the Department of Mathematics and Computer Science, Eindhoven University of Technology, den Dolech 2, 5612 AZ Eindhoven, The Netherlands (email: m.m.h.p.v.d.heuvel@tue.nl; r.j.bril@tue.nl)

Stefan Schiemenz and Christian Hentschel are with the Department of Media Technology, Brandenburg University of Technology, Konrad-Wachsmann-Allee 1, 03046 Cottbus, Germany (email: stefan.schiemenz@tu-cottbus.de; christian.hentschel@tu-cottbus.de)

Manuscript received February 16, 2010
Current version published 06 29 2010;
Electronic version published 07 06 2010.

application, we need support for the control strategies of its decision scheduler through (*i*) mechanisms for dynamic resource allocation to its algorithms and (*ii*) efficient implementations of these mechanisms. To address this problem, we consider a basic setting of a multimedia system with an application containing two priority processing algorithms for deinterlacing [5] and sharpness enhancement [3] that share a general purpose platform.

B. Contributions

Based on described control strategies in [4], we identify three basic mechanisms for dynamic resource allocation, i.e. *preliminary termination*, *resource allocation* and *monitoring*. We compare the performance of different implementations for each of the identified dynamic resource allocation mechanisms on a general purpose platform.

Furthermore, we identify an application model for the priority processing application. The generic application model is mapped on a process model using a concrete general purpose platform.

C. Overview

The outline of the remainder of this paper is as follows. First, Section II discusses related work. Section III presents an abstract application model, describing the distinctive responsibilities between system support and the priority processing application. The required mechanisms for dynamic resource allocation are explained in more detail in Section IV. Section V describes the mapping of the application model to a concrete general purpose platform. Experiments comparing different implementations of the basic mechanisms for dynamic resource management are presented in Section VI. Section VII revisits our assumptions and discusses directions for future work. Finally, Section VIII concludes this paper.

II. Related Work

The priority processing concept is a fine-grained realization of the milestone approach proposed by Audsley et al. [6]. In [6] a computational model is presented to incorporate unbounded software components in real-time systems, such that bounded tasks are guaranteed. However, dynamic control strategies and resource management are not discussed in [6]. The dynamic resource allocation *mechanisms* described in this paper support dynamic resource allocation *control strategies* for SVAs based on priority processing. Because SVAs based on priority processing differ from SVAs based on more traditional approaches, their accompanying control strategies are also different. Similarly, dynamic resource allocation also differs from resource management techniques, such as reservation-based resource management, that have been applied in the context of more traditional approaches to SVAs. Whereas our dynamic resource allocation mechanisms are unique, the architectural approach taken for SVAs on programmable platform is similar to, for example, Hentschel et al. [2]. In particular, we also make a clear distinction between system and application responsibilities and address these responsibilities in dedicated components in the architecture of our system. As an example, the application specific control strategy is addressed by the decision scheduler, a constituent of the media application.

In the remainder of this section, we first consider the distinguishing characteristics of priority processing compared to more traditional approaches to SVAs. Next, we consider the impact of these characteristics on the accompanying control strategies. Finally, we address the commonality and differences of dynamic resource allocation and reservation-based resource management.

A. Scalable Video Algorithms

As mentioned above, SVAs trade resources usage against output quality at the level of individual frames. In traditional approaches, an SVA provides a fixed number of quality levels that can be chosen for each frame. Because a quality level can only be changed at the level of individual frames, a frame is entirely processed at a particular quality level or otherwise the processing has to be aborted. For cost-effectiveness reasons, it is common to take a work-preserving approach, i.e. upon an overload situation, the processing of the current frame is completed and a next frame is skipped [7], [8]. Note that buffering is inherent for a work-preserving approach.

Conversely, SVAs based on priority processing do not have quality levels. Moreover, the processing of a frame can be terminated at an arbitrary moment in time once initial output at a basic quality level has been created, yielding the best output for given resources.

B. Control Strategies

A control strategy for SVAs aims at maximizing (perceived) output quality. In [7], control strategies are presented for a single, work-preserving SVA based on the traditional approach. Given the highly fluctuating, content dependent processing times of frames for the SVAs considered, such as codecs and algorithms that contain motion estimation or motion compensation, working ahead is applied to even out the load in time. Hence, processing of the next frame is started immediately upon completing the previous one, provided that the input data is available. Processing is blocked when there is no input data. Latency and buffer constraints determine the extent to which working ahead can be applied [9]. The strategies select a quality level for processing the next frame upon completion of the previous frame, i.e. synchronous with processing.

The control strategies considered in this paper, allocate resources to multiple, priority processing based SVAs. Because each SVA must provide output at the end of the frame period, there is no need for either additional buffering techniques or additional synchronization techniques between audio and video. Instead, we need appropriate mechanisms to preliminary terminate SVAs. The strategies select the SVA to execute next upon completion of a time-slot, i.e. synchronous with time.

C. Resource Management

Dynamic resource allocation inherently provides *isolation* between priority processing algorithms, i.e. temporary or permanent faults occurring in one algorithm cannot hamper the execution of other algorithms. Dynamic resource allocation therefore has much in common with reservation-based resource management [10], [11]. The distinguishing characteristics of dynamic resource allocation are first the inherent lack of reservations, i.e. no resources are guaranteed to SVAs except for those allowing an SVA to produce a basic output at lowest quality. Secondly, the need for preliminary termination of SVAs at the end of a frame-period requires synchronization between processing and time. Note that, dynamic resource allocation can be complemented with reservation-based resource management to facilitate isolation between multiple priority processing applications. Whereas Wüst et al. [7] assume a fixed amount of resources per frame-period for a single SVA, the amount of resources allocated to each priority processing SVA may fluctuate per period. Moreover, even a priority processing application can handle fluctuating resource availability, although we will restrict ourselves in this paper to situations where a priority processing application has the full processor at its disposal.

III. Application Model

In this section we consider the application structure, without choosing an explicit underlying platform. Note that the application can still put requirements on the platform. The system architecture, including both application specific as well as system specific components, is shown in Figure 2.

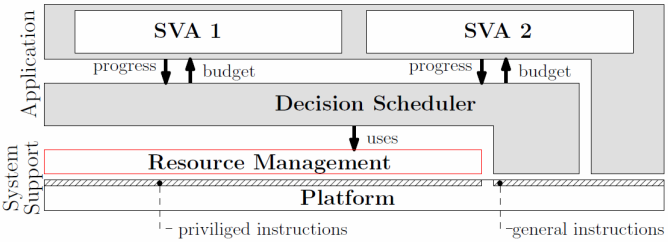

Fig. 2. System architecture including the priority processing application. The application components are indicated with a grey background. The decision scheduler assigns time-slots based upon progress information provided by the SVAs. In this paper the underlying mechanisms for dynamic resource management are investigated.

An application is composed of a decision scheduler and m independent SVAs, where $m \in N^+$. All SVAs are known upfront by the decision scheduler. The decision scheduler divides the available resources among the algorithms using an application specific strategy. The platform consists of the hardware and a corresponding operating system. The operating system manages resources, i.e. allocation and de-allocation of processor resources, at the level of tasks. It is assumed that the operating system provides a fixed priority scheduler. The platform can provide various interfaces to applications [12]. The resource management layer is allowed to use privileged instructions provided by the platform. The resource management layer can be seen as a library, providing an abstracted interface to the application. The decision scheduler uses the resource management library by linking to the library at compile time. Additionally, the application components have access to general instructions provided by the platform to process their workload.

A. SVA Characteristics

SVAs which are implemented following the priority processing paradigm are characterized as follows:

1) An SVA has three algorithmic phases: basic, analysis and enhancement see Figure 1. The analysis and enhancement phases constitute the scalable part of the SVA.
2) An SVA can be preliminary terminated after the basic function, e.g. during the analysis or enhancement phase.
3) All algorithms are synchronous with the period, which means that each period the SVAs start with a new frame and at the end of a period the processing is terminated.
4) Each period the input for the algorithms is available and the output can be written to an output buffer, i.e. SVAs are not blocked by input and output.
5) An SVA possibly contains an epilog which is responsible for writing the produced output to the (frame-) buffer at the end of each period.
6) Each algorithm accounts its own progress and updates the progress during runtime.

B. Task Model

We assume a set Γ of m strictly periodically released tasks $\tau_1, \tau_2, \ldots, \tau_m$, modeling the set of m priority processing algorithms of an application that are executed on a single processor. A job is an instance of a task, representing the work to be done by an algorithm for a single video frame. A task τ_i is characterized by a period $T_i \in R^+$, a (relative) deadline $D_i \in R^+$, a computation time $C_i \in R^+$, where $C_i \leq D_i \leq T_i$, and a phasing $\varphi \in R^+ \cup \{0\}$, representing the start-time of τ_i. A task τ_i can be viewed as a sequence of three subtasks representing a basic part $\tau_{i,basic}$, a scalable part $\tau_{i,scalable}$, and possibly an epilog $\tau_{i,epilog}$. Correspondingly, the computation time C_i of τ_i can be viewed to consist of a basic part $C_{i,basic}$, a scalable part $C_{i,scalable}$ and an epilog $C_{i,epilog}$, i.e.

$$C_i = C_{i,basic} + C_{i,scalable} + C_{i,epilog}$$

The basic part and the epilog are mandatory parts of a task, whereas the scalable part of a task can be preliminary terminated. The epilog is ideally constant and as small as possible. Typically, it is not feasible to compute a realistic estimate of the worst-case computation time of an SVA, since multimedia processing algorithms are characterized by heavily fluctuating, data-dependent workloads.

All tasks modeling SVAs have the same period T, deadline D, and phasing φ. Moreover, it is assumed that the period and deadline are equal, i.e. $T = D$. During every period T, an amount Q processor time, with $Q \leq T$, is available for executing the SVAs. When the entire processor is available to the SVAs, this available processor time (or budget) is equal to the period, i.e. $Q = T$. The mandatory parts of the SVAs are

required to fit within this budget Q to guarantee a minimal output upon depletion of the budget, i.e.

$$\sum_{i=1}^{m}(C_{i,basic} + C_{i,epi\log}) \le Q$$

In our models it is assumed that the budget Q is large enough to perform all m mandatory sub-tasks, and therefore no further admission test is required for the integration of scalable subtasks [6]. The remaining time of a budget during a period can be used for the scalable parts of the algorithms, and is divided among the SVAs by the decision scheduler. To facilitate this division of time, the execution of the scalable parts of tasks are delayed till all tasks have completed their basic part. Hence, whereas the SVAs are independent, the executions of their corresponding tasks are explicitly synchronized. Moreover, the decision scheduler has to reserve time during every period to allow for the execution of the epilogs of all tasks. Finally, all pending executions of the scalable parts of the tasks have to be terminated when the remaining time of a budget has been consumed; see also Figure 3.

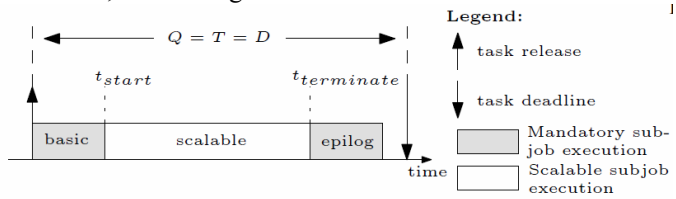

Fig. 3. Division of a period in three parts for execution of: 1) basic parts, 2) scalable parts, and 3) epilogs of all tasks.

For ease of presentation, we assume that the amount of time required for the mandatory parts are negligible, i.e. $T_{terminate} - t_{start} = T$. Assume that each period of length T contains N time-slots with a fixed size Δt_s, i.e. $T = N \times \Delta t_s$. At the start of each time-slot the decision scheduler is activated to assign the time-slot to an SVA.

C. Control Strategies

The decision scheduler decides at any given time which of the available scalable subtasks $\tau_{i,scalable}$ to run. In [6] a straight-forward fixed priority assignment is proposed, whereas we dynamically allocate the available resources aiming to maximize the overall system progress. Different strategies for dynamic resource allocation are implemented within the decision scheduler and have been compared by Schiemenz [4]. In this paper, we restrict to two strategies for allocating time-slots to algorithms:

1) **Round-robin (RR):** competing SVAs are assigned time-slots in a strictly alternating way, independent of the individual progress of the algorithms.

2) **Reinforcement learning (RL):** competing SVAs are assigned time-slots based upon a self-learning algorithm. All SVAs provide the decision scheduler with a progress value that indicates the actual processing state, and therefore provides a measure to estimate the benefit to the entire system of its execution in the next time-slot. The estimation of benefit becomes better when taking into account the progress value per time-unit of execution, i.e. a time-slot Δt_s. The decision scheduler acts as an agent [13], periodically observing the reached states of the SVAs. Based upon the gathered information, a decision is made aiming at maximizing the overall progress values. On the one hand, the RR strategy predefines the amount of time-slots assigned to each competing SVA per period, reducing the amount of control parameters compared to the RL strategy. On the other hand, the RL strategy balances the available resources more efficiently over the algorithms compared to the RR strategy.

IV. Dynamic Resource Allocation

Our dynamic resource allocation scheme defines the mechanisms to support the application semantics from the system perspective to optimize resource utilization. The interaction between system and application components is shown in Figure 2. The distinction between application specific and system specific responsibilities is summarized in Table I. The application specifies the *strategies* to describe its desired behavior whereas the system should provide appropriate *mechanisms* to implement the strategy.

TABLE I
Application versus System Specific Responsibilities

Application Specific (Strategy)	System Support (Mechanism)
Skip remainder of video frame upon budget depletion	Preliminary termination
Resource distribution (Scheduling on meta-level, e.g. RL strategy)	Allocation of processor (use platform primitives)
Monitor progress values of the individual SVAs	Account consumed time (time-slots) and activate the decision scheduler

We identified three basic mechanisms for dynamic resource allocation: *preliminary termination*, *resource allocation* and *monitoring*. Note that preliminary termination is a pre-requisite for running a priority processing algorithm on a resource constraint platform. Resource allocation and monitoring come into play when multiple algorithms share a platform. A brief description of these mechanisms and their implementations is given below.

A. Preliminary Termination

Priority processing relies on mechanisms to preliminary terminate an SVA at the end of each period, and skip the remainder of the pending frame. Preliminary termination can be done by means of either (*i*) cooperative termination through polling on regular intervals or (*ii*) signaling of tasks by the decision scheduler control component. Preliminary termination by means of polling intuitively allows to trade-off termination latency versus computational overhead by choosing an appropriate polling granularity. We compare the following alternatives:

1) **Cooperative termination**: The initiative for preliminary terminating SVAs is shared by the SVAs themselves and the decision scheduler. In this paper we distinct two granularities:

(*a*) **Per pixel polling**: At the beginning of each pixel computation, the termination flag is polled in order to check whether the budget is sufficient to continue processing; and (*b*) **Per block polling**: The video screen is divided in blocks of 8 × 8 pixels. Many video algorithms work at the granularity of a pixel block. At the beginning of each block computation, the termination flag is polled in order to check whether the budget is sufficient to continue processing.

2) **Signaling (no polling):** The whole video frame is processed without any flag polling which requires preliminary termination of jobs by the decision scheduler triggering a signal. The initiative for preliminary terminating SVAs is clearly to the decision scheduler.

B. Resource Allocation

To allocate CPU time, a task implementing an SVA is assigned to the processor by means of either (*i*) suspending and resuming the task or (*ii*) manipulating the task priority such that the native fixed priority scheduler (FPS) of the platform can be used. The latter option implicitly reduces unnecessary blocking of tasks, and allows the consumption of gain-time [7]. Gain-time is the amount of time allocated to but not used by an algorithm, and therefore becomes available as additional time to other algorithms.

C. Monitoring

Mechanisms for monitoring have to keep track of the amount of processing time (resources) consumed by each SVA. After each time-slot the decision scheduler is activated which can be achieved by either (*i*) putting the decision scheduler in a highest priority task which is periodically activated or (*ii*) activating the decision scheduler by means of a software-interrupt. The amount of time consumed by an SVA is measured by the decision scheduler in terms of time-slots allocated to that SVA. Gain-time is therefore accounted to a gain-time provider rather than its consumer.

V. Application Deployment

This section discusses the implementation of a priority processing application on a general purpose simulation platform.

A. Experimental Setup

The priority processing applications are implemented in a Matlab/Simulink environment and executed on a general purpose multi-core platform under Microsoft Windows XP using standard sequences from the Video Quality Experts Group (VQEG). The SVAs are executed on a single core. The decision scheduler is executed on another core. More extensive descriptions of the experimental setup and the obtained results are available in [14].

B. Application Mapping to Processes and Threads

The Matlab environment runs in a process on Microsoft Windows XP. Our prototyped priority processing application is contained in this process. A process is a program in execution that defines a data space and has at least one associated thread. A thread is a unit of concurrency and a unit of scheduling. The Matlab process, and therefore the application, is put in the so-called real-time class and its threads are scheduled using fixed priority scheduling (FPS).

Different alternatives can be considered to map the application's tasks on threads. Specifically, we can trade-off the following implementations for mapping SVAs to threads:

1) **Map a job onto a thread:** Each job (a frame-computation) can be executed by starting a new thread. When the budget is depleted, the job is terminated and therefore also the corresponding thread is terminated.

2) **Map a task onto a thread:** All jobs corresponding to a task are multiplexed on a single thread. A thread is reused by consecutive jobs of the same task, such that for each frame computation the cost of creating and destroying threads is saved. Instead of creating and destroying threads periodically, additional time-synchronization mechanisms are required. Upon preliminary termination of a job, all internals are reset and a new frame computation is started, discarding all remaining work of the pending frame. Each SVA is mapped onto a single thread, multiplexing its job instantiations over this thread. These threads, implementing the SVAs, are created upon initialization of the simulation and destroyed upon finalization of the simulation.

The alternative implementation of mapping jobs directly onto threads shows a high operating system overhead for periodically creating and destructing threads.

VI. Experiments and Results

As a leading example, we consider an application composed from two independent priority processing algorithms: a sharpness enhancement filter [3] and a deinterlacer [5]. This example application can be visualized as a picture-in-picture application, in which both algorithms work on an independent input stream. In the remainder of this section we compare different implementations of the mechanisms for dynamic resource allocation.

A. Preliminary Termination

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

1) **Termination latency:** the time interval measured from the time that the deadline expires (polling flag is toggled), until the time that the algorithm gives back control to the decision scheduler (a new job is started). Based on extensive tests, we derived that the lower bound on termination latency for cooperative termination through polling on Windows and GNU/Linux is determined by the granularity of the operating system scheduler rather than the granularity of polling. In particular, there is only a minor difference in polling each block- or pixel-computation, as shown in Figure 4. Typical termination latencies are in the order of 40~45 μs. The block based polling variant gives a more deviated termination latency compared to pixel based polling, see Figure 4, due to the higher intervals between each polling statement. Average

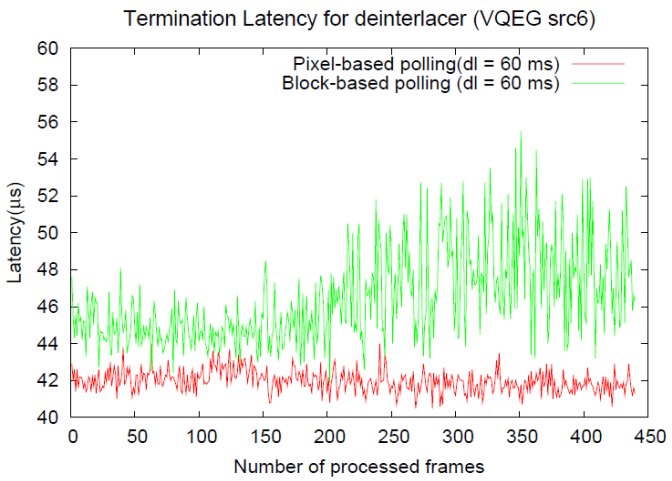

Fig. 4. Termination of jobs by polling with deadlines of 60 ms. The latencies are influenced by the content dependent block execution times, resulting in deviated latencies.

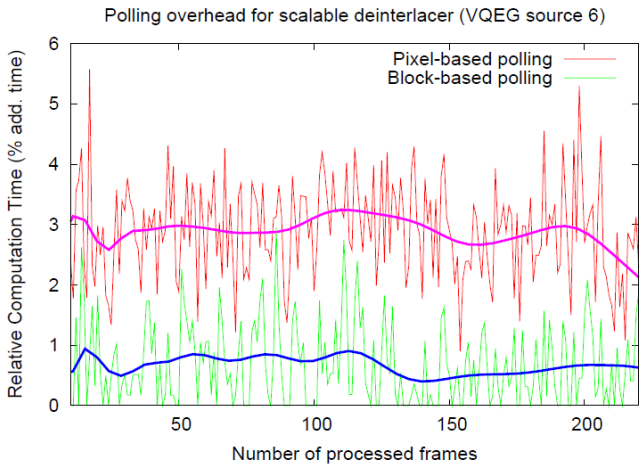

Fig. 5. Relative computation times to complete the scalable parts of the deinterlacer application for VQEG source 6, for a scene with relative high overhead. Flag polling on the granularity of a per pixel and per block computation is compared with an implementation without polling.

latencies for block based polling, with a 95% confidence interval, are 45.2 µs ± 0.127, whereas for pixel based polling the average latency is 42.2 µs ± 0.039. Since a block consists of 8 × 8 pixels, it is expected to see a factor of 64 difference between the latencies of both polling variants. The explicit time-synchronization requires transfer of control between job-instances, i.e. semaphores are used to wait until the activation of the successive period. The semaphore is released by the decision scheduler, causing a lower bound on the latency.

The main motivation for signaling is the decreased effort for code maintenance compared to polling based preliminary termination approaches. When a signal arrives, indicating expiration of the period, the signal-handler is executed. The signal-handler replaces the polling statements which pollute the signal processing code.

Since Windows lacks support for asynchronously signaling threads, termination of jobs by external signaling is tested on GNU/Linux platforms using POSIX signals [15]. Experimental results for preliminary termination by means of polling variants show similar results on GNU/Linux as on Windows. Although the Linux kernel supports signaling of threads, signal-handling relies on the granularity of the kernel clock, and the scheduler which has to schedule the thread to which a signal is sent. Signaling has shown to be less reliable compared to polling based preliminary termination, and causes relative high and unpredictable latencies in the order of 70~90 µs.

2) **Computational overhead:** regularly checking whether a flag has been set in order to preliminary terminate a frame-computation requires processor time. In case of preliminary termination by means of signaling, it is aimed to reduce the computational overhead to a minimum. Polling involves com-putational overhead and affects signal processing, in particular on pipelined architectures. In our simulation environment on a general purpose machine, there is just a small computational overhead measurable of maximal 3%, see Figure 5.

B. Resource Allocation

In [4], two strategies for allocating time-slots to algorithms have been investigated, one based on round-robin (RR) and another based on reinforcement learning (RL). Unlike RR, the RL strategy typically assigns multiple consecutive slots to the same algorithm. An initial implementation of the RL strategy by means of suspend-resume caused unnecessary context-switching overhead, as shown in Figure 6. This overhead can be reduced by only suspending algorithms in favor of other algorithms.

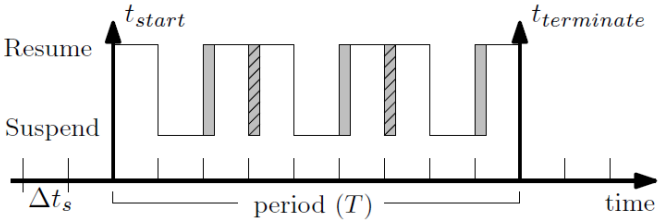

Fig. 6. Context-switching overhead: grey areas show the time lost due to context-switching upon resuming an SVA-task. This overhead can be reduced, when multiple consecutive slots are assigned to a task (i.e. the shaded-grey areas can be removed). At the end of the period $t_{terminate}$, all pending SVAs must be preliminary terminated.

Alternatively, an efficient way to reduce this overhead is to manipulate priorities which has as additional advantage that gain-time can be consumed by algorithms with a lower priority. Figure 6 helps us to identify two types of gain-time: (*i*) an algorithm completes a frame within an allocated time-slot, such that the remainder of that time-slot becomes available as gain-time; and (*ii*) the scheduling strategy assigns a time-slot to a completed task such that an entire time-slot becomes available as gain-time.

The first type of gain-time is just a fraction of a time-slot, only provided by an algorithm at most once per period, and is neglectable. The second type of gain-time occurs very often using the RR strategy. The RL strategy has a notion of the progress of the algorithms, and the self-learning characteristic tries to avoid non-optimal time-slot assignments. Note that gain-time is not meant to be consumed by competing SVAs,

since gain-time is not accounted to its consumer. Therefore gain-time consumption influences the reinforcement learning strategy which decides time-slot allocation based on the SVA's progress related to the amount of assigned time.

In order to have a unit of measurement at the one hand, and to demonstrate the capability of priority processing to efficiently consume additional available resources on the other hand, gain-time is assigned to competing SVAs in our simulation environment.

Experiments with RR show no performance differences between two alternatives for the resource allocation mechanism, suspending/resuming of tasks versus priority manipulation, when gain-time consumption is disallowed. This result is as expected, since RR does not take advantage of reduced context-switching overhead. The RL strategy can take advantage of the resulting reduction of control overhead and gain-time consumption, as illustrated in Figure 7.

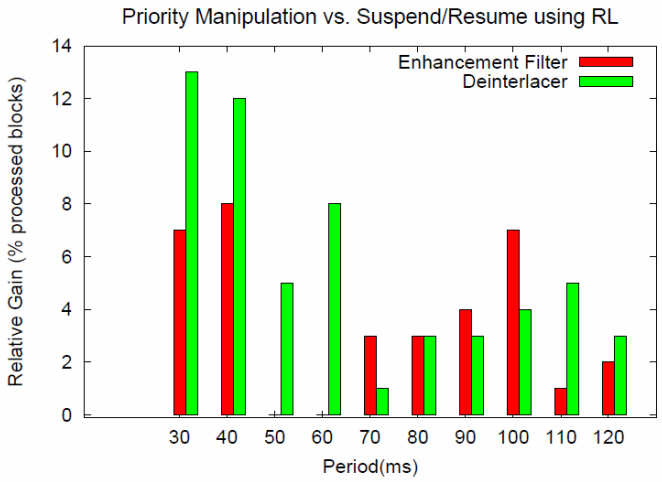

Fig. 7. Relative gain for the RL strategy of scheduling based on priority manipulation compared to suspend/resume. Priority manipulation gives a significant improvement over suspend/resume due to reduced control overhead and gain-time consumption. Measurement data is obtained from an example application composed from two independent priority processing algorithms for deinterlacing and sharpness enhancement of video content.

C. Monitoring

Monitoring mechanisms keep track of the consumed time on a time-slot scale. We assume $\Delta t_s = 1\ ms$, based upon earlier research results [4]. Software interrupts on a 1 ms scale on a Windows platform showed relative high overhead, causing violation of periodic constraints. Theoretically, Windows supports timers with a 1 ms granularity. However, the internal clock runs at 100 Hz, such that higher frequencies can not be guaranteed.

To cope with the lack of precise timers, we implemented the periodic activation of the decision scheduler by means of a busy-waiting loop in our simulation environment. The busy-waiting loop repeatedly reads the Time Stamp Counter (TSC) register. Note that the usage of the TSC requires a static assignment of the decision scheduler to a core, since each core in a multi-processor environment has its own TSC which are not necessarily synchronized [16]. In a real-time environment it is more straightforward to implement the decision scheduler with a software interrupt by making use of high-resolution timers.

D. Processor Utilization

During simulations, the processor utilization of the cores is observed using the Windows Task Manager Performance visualizer. The processor on which the SVAs run is utilized for approximately 40% which means that the SVAs are limited in using full processor capacity by other factors. A possible disturbing factor is the Matlab/Simulink environment which runs many background threads, for example to gather measurement information during simulation run-time. Because extracts statistical information from the decision scheduler and the SVAs, the caused inter-thread communication between Matlab threads and application specific threads induces additional time-synchronization overhead.

Another potential impairment is non-optimal memory usage. The priority processing algorithms use a sorted processing order to enhance video content which differs from the traditional approach in which content is processed in consecutive order. The sorted order leads to non-linear memory access which causes a high I/O bound. Note that pre-fetching techniques can be used as soon as the sorted order is anchored.

The combination of both factors can cause an even worse scenario. Since multi-core machines make use of a shared cache, the background threads of Matlab possibly pollute the cache of the video-processing threads and vice versa.

VII. Discussion

The trend in today's multimedia systems design is an ever increasing number of applications which are mapped on the same device. To cope with the entailed demands on computational power and increased system integration complexity on resource constrained platforms of today's multimedia systems, current challenges in embedded system designs move to multiprocessor platforms and virtual platforms. Note that multiple processors and virtualization techniques can co-exists next to each other [17].

In this paper, we assumed a basic model with a multimedia application containing independent SVAs that are running on a single processor, and have the entire processor at their disposal. Moreover, we compared implementations of dynamic resource management mechanisms based on a deployment on a general purpose platform. In this section, we briefly discuss the challenges in lifting these assumptions and mapping priority processing applications on embedded platforms.

A. Deployment on an embedded platform

A straightforward successive step is to map the priority processing application on an embedded platform. Whereas the priority processing concept seems very promising from a processor's resource management perspective, it is expected that for embedded platforms the emphasis will be on managing data transport.

On general purpose platforms, preliminary termination by means of polling has shown to be an attractive alternative. Computational overhead for polling is expected to be much higher on dedicated streaming processors, such as DSPs or VLIWs. On these pipelined processor architectures, (*i*) polling affects the pipelined nature of signal processing applications reducing its computing efficiency, and (*ii*) the data-bus has become the bottleneck in today's embedded system design due to the data intensive nature of signal processing applications. The limited bandwidth of the data-bus will become the dominating factor for the performance, and the dynamic resource allocation scheme will therefore probably move from a time-partitioned scheduling approach on the processor (as described in this paper) to a data-partitioned scheduling approach on the data-bus, i.e. processing of video content takes place on the granularity of fixed data-chunks, such as blocks, rather than time-slots.

B. Virtual Platforms

In this paper it is assumed that an application has the entire processor at its disposal. The increasing complexity of real-time systems demands for a decoupling between (*i*) development and analysis of individual applications and (*ii*) integration of applications on a shared platform, including the analysis at the system level [18]. Hierarchical scheduling frameworks (HSFs) have been extensively investigated as a paradigm to facilitate this decoupling. Consider a two level HSF, where a system is composed of a set of independent applications, each of which is composed of a set of tasks. A global scheduler is used to determine which application should be allocated the processor at any given time. A local scheduler determines which of the chosen application's tasks should actually execute.

The decision scheduler and the SVAs are mapped on different physical processors, imposed by the platform limitations of our simulation environment. Given the priority processing application, the decision scheduler task is assigned the highest priority within the application, such that upon activation it can immediately preempt the SVAs. Activation of the decision scheduler is based on fixed size time-slots, Δt_s. Since within an HSF the application is provided with a virtual share of the processor resources, the decision scheduler needs to be activated relative to the consumed budget instead of the global time progression. Hence, on a shared platform the priority processing application requires virtual timers to trigger timed events relative to the consumed budget, to activate the decision scheduler for monitoring the progress of the SVAs. Literature [19] outlines an implementation for virtual timers, targeted at embedded real-time platforms.

C. Multi-processor Platforms

The increasing availability of (homogeneous) multi-processor platforms allows different mappings of (sub-) tasks to the processing elements. Typically, in multiprocessor embedded systems a static scheduling approach is chosen, because of the relative high overhead and implementation complexity of dynamic schedules [20]. Although dynamic scheduling techniques can optimize processor utilization, this immediately raises issues with our decision scheduler which needs to be adapted to multi-processor scheduling. The mapping of priority processing applications on (multiprocessor) embedded systems and its challenges are left for future work.

D. Dependent SVAs

It is assumed that the multiple SVAs composing a priority processing application are independent. This assumption might be too restrictive for future applications. By creating a chain of time-driven tasks (SVAs), separated by buffers, we can simply lift the assumptions on independence of SVA. The synchronization mechanisms between tasks consuming and producing buffered data is similar to the mechanisms described in this paper. After preliminary termination of the concurrent SVAs the epilog sub-tasks write the data to their output buffers, such that the data becomes available in the next period for consumption. This solution comes at the cost of increasing the end-to-end delay of the final output frame.

VIII. CONCLUSION

Scalable video algorithms using novel priority processing can guarantee real-time performance on programmable platforms even with limited resources. According to the priority processing principle, scalable video algorithms follow a priority order. Dynamic resource allocation is required to maximize the overall output quality of independent, competing priority processing algorithms that are executed on a shared platform.

We described three basic mechanisms for dynamic resource allocation to competing priority processing algorithms, i.e. preliminary termination, resource allocation, and monitoring, and compared the performance of different implementations of these mechanisms on a general purpose platform. The identified mechanisms and architectural model are independent of the chosen platform. Changing the underlying platform leads to a repeated deployment process, which includes the trade-offs for implementing the mechanisms for dynamic resource allocation.

A. Preliminary Termination

For preliminary termination we compared cooperative termination and a signaling approach. Cooperative termination is preferred, since most platforms, including Windows, do not support reliable termination of tasks through signaling. Cooperative termination allows a trade-off in termination latency and computational overhead, induced by the granularity of polling.

B. Resource Allocation

Resource allocation can be implemented by either priority manipulation or suspending-resuming of tasks. Priority manipulation using the native operating system's scheduler, e.g. FPS of Windows, allows consumption of gain-time and reduces unnecessarily blocking of tasks.

C. Monitoring

Monitoring is done on a time-slot scale and relies on the availability of high-resolution timers. Due to unavailability of high-resolution timers on the Windows platform, the decision scheduler monitoring task is implemented by means of a busy waiting loop.

REFERENCES

[1] R. Bril, C. Hentschel, E. Steffens, M. Gabrani, G. van Loo, and J. Gelissen, "Multimedia QoS in consumer terminals (invited lecture)," in Proc. IEEE Workshop on Signal Processing Systems (SIPS), September 2001, pp. 332–343.

[2] C. Hentschel, R. Bril, Y. Chen, R. Braspenning, and T.-H. Lan, "Video Quality-of-Service for consumer terminals – a novel system for programmable components," IEEE Transactions on Consumer Electronics (TCE), vol. 49, no. 4, pp. 1367–1377, November 2003.

[3] C. Hentschel and S. Schiemenz, "Priority-processing for optimized real-time performance with limited processing resources," Int. Conference on Consumer Electronics (ICCE). Digest of Technical Papers, January 2008.

[4] S. Schiemenz, "Echtzeitsteuerung von skalierbaren Priority-Processing Algorithmen," Tagungsband ITG Fachtagung - Elektronische Medien, March 2009, pp. 108 – 113.

[5] S. Schiemenz and C. Hentschel, "De-interlacing using priority processing for guaranteed real time performance with limited processing resources," in Proc. Int. Conference on Consumer Electronics (ICCE). Digest of Technical Papers, January 2009.

[6] N. C. Audsley, A. Burns, R. Davis, and A. Wellings, Integrating Unbounded Software Components into Hard Real-time Systems, series: Imprecise and Approximate Computation, Kluwer Academic, 1995, vol. 318, ch. 5, pp. 63–86.

[7] C. C. Wüst, L. Steffens, W. F. Verhaegh, R. J. Bril, and C. Hentschel, "QoS control strategies for high-quality video processing," Real-Time Syst., vol. 30, no. 1-2, pp. 7–29, 2005.

[8] W. Zhao, C. C. Lim, J. W. Liu, and P. D. Alexander, Overload Management by Imprecise Computation, series: Imprecise and Approximate Computation, Kluwer Academic, 1995, vol. 318, ch. 1, pp. 1–22.

[9] D. Isovic, G. Fohler, and L. Steffens, "Timing constraints of MPEG-2 decoding for high quality video: Misconceptions and realistic assumptions," in Proc. 15th Euromicro Conference on Real-Time Systems (ECRTS), July 2003, pp. 73–82.

[10] C. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in IEEE Int. Conference on Multimedia Computing and Systems (ICMCS), May 1994, pp. 90–99.

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

[12] A. S. Tanenbaum and M. van Steen, Distributed Systems: Principles and Paradigms (2nd Edition). Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[13] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction. Cambridge, MA, USA: MIT Press, 1998.

[14] M. van den Heuvel, "Dynamic resource allocation in multimedia applications," Master's thesis, Eindhoven University of Technology, July 2009

[15] D. B. Thangaraju, "Linux signals for the application programmer," Linux Journal, vol. 2003, no. 107, pp. 1–6, March 2003.

[16] G.-S. Tian, Y.-C. Tian and C. Fidge, "High-Precision Relative Clock Synchronization Using Time Stamp Counters", in Proc. 13th IEEE Int. Conference on Engineering of Complex Computer Systems (ICECCS), April 2008, pp. 69–78.

[17] I. Shin, A. Easwaran, and I. Lee, "Hierarchical scheduling framework for virtual clustering of multiprocessors," in Euromicro Conference on Real-Time Systems (ECRTS), July 2008, pp. 181–190.

[18] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in Proc. 24th IEEE Real-Time Systems Symposium (RTSS), December 2003, pp. 2–13.

[19] M. van den Heuvel, M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien, "Virtual timers in hierarchical real-time systems," Work-in-Progress (WiP) session of the 30th IEEE Real-time Systems Symposium (RTSS)., December 2009, pp. 37–40.

[20] S. Sriram and S. S. Bhattacharyya, Embedded Multiprocessors: Scheduling and Synchronization. Boca Raton, FL, USA: CRC Press, Inc., 2009.

BIOGRAPHIES

Martijn M.H.P. van den Heuvel received a B.Sc. in computer science and a M.Sc. in embedded systems from the Technische Universiteit Eindhoven, the Netherlands. In September 2009, he started research in the System Architecture and Networking (SAN) group of the Mathematics and Computer Science department of the Technische Universiteit Eindhoven. His main research interests are in the area of real-time embedded systems.

Reinder J. Bril received a B.Sc. and a M.Sc. (both with honors) from the University of Twente, and a Ph.D. from the Technische Universiteit Eindhoven, the Netherlands. He started his professional career in January 1984 at the Delft University of Technology. From May 1985 till August 2004, he has been with Philips, and worked in both Philips Research as well as Philips' Business Units. He worked on various topics, including fault tolerance, formal specifications, software architecture analysis, and dynamic resource management, and in different application domains, e.g. high-volume electronics consumer products and (low volume) professional systems. In September 2004, he made a transfer back to the academic world, i.e. to the System Architecture and Networking (SAN) group of the Mathematics and Computer Science department of the Technische Universiteit Eindhoven. His main research interests are currently in the area of reservation-based resource management for networked embedded systems with real-time constraints.

Stefan Schiemenz received his Dipl.-Ing. (electrical engineering) in 2003 at the Brandenburg University of Technology (BTU) in Cottbus, Germany. Since then he is working in the field of scalable video algorithms using the principle of Priority Processing at the department of Media Technology. Currently, he is pursuing a Dr.-Ing. (Ph.D. degree) at the BTU. He is a member of the IEEE and of the FKTG in Germany.

Christian Hentschel received his Dr.-Ing. (Ph.D.) in 1989 and Dr.-Ing. habil. in 1996 at the University of Technology in Braunschweig, Germany. He worked on digital video signal processing with focus on quality improvement. In 1995, he joined Philips Research in Briarcliff Manor, USA, where he headed a research project on moiré analysis and suppression for CRT based displays. In 1997, he moved to Philips Research in Eindhoven, The Netherlands, leading a cluster for Programmable Video Architectures. Later he held a position of a Principal Scientist and coordinated a project on scalable media processing with dynamic resource control between different research laboratories. In 2003, he became a full professor at the Brandenburg University of Technology in Cottbus, Germany. Currently he chairs the department of Media Technology. In 2005 he received, together with his co-authors, the Chester Sall Award from the IEEE Consumer Electronics Society for the first place transaction paper. He is a member of the Technical Committee of the International Conference on Consumer Electronics (IEEE), active in the IEEE CE Society, and a member of the FKTG in Germany.