Constant-bandwidth supply for priority processing

Citation for published version (APA):

Heuvel, van den, M. M. H. P., Holenderski, M. J., Bril, R. J., & Lukkien, J. J. (2011). Constant-bandwidth supply for priority processing. *IEEE Transactions on Consumer Electronics*, 57(2), 873-881.
<https://doi.org/10.1109/TCE.2011.5955235>

DOI:

[10.1109/TCE.2011.5955235](https://doi.org/10.1109/TCE.2011.5955235)

Document status and date:

Published: 01/01/2011

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Constant Bandwidth Supply for Priority Processing

Martijn M. H. P. van den Heuvel, *Student Member IEEE*, Mike Holenderski, *Student Member IEEE*,
Reinder J. Bril, and Johan J. Lukkien, *Member IEEE*

Abstract — *Today's consumer electronic devices feature multiple applications which have to share scarcely available resources. We consider a priority-processing-based video application, which comprises multiple scalable video algorithms (SVAs) that are executed on a shared, virtual platform. This application is given a guaranteed processor share by means of a constant-bandwidth server (CBS). A decision scheduler distributes the assigned processor share among the SVAs on a time-slot basis, with the aim to maximize their overall output quality. To correctly distribute this processor share based on fixed-sized time slots, we introduce the concept of a virtual timer. This timer only advances when its associated virtual platform is executing. Because priority processing can guarantee real-time performance even under fluctuating load, we apply a resource reclaiming mechanism to our CBS which makes it possible to efficiently exploit spare processor time¹.*

Index Terms — **scalable video algorithms, resource management, reservations, dynamic resource allocation.**

I. INTRODUCTION

Reservation-based scheduling [1] has been successfully applied in consumer electronics [2] to decompose complex, fully loaded systems into well-defined components. It provides a mechanism to concurrently schedule multiple applications with different timing constraints on a shared platform and to guarantee each application a fixed share. 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 resource reservation techniques [2]. The combination of scarcely available resources and fluctuating, data-dependent load makes media processing inherently greedy. A reservation-based approach can effectively reduce interference due to dynamic load fluctuations. In this paper we focus on platform support to guarantee a *fixed share* of the system resources for each application, consisting of *multiple SVAs* rather than a single SVA as in [2].

We consider a video processing application containing competing, independent priority-processing algorithms. This

application may need to share the processor with (non-greedy) third-party applications. The principle of priority processing provides optimal real-time performance for SVAs on programmable platforms with limited system resources [3]. According to this principle, SVAs provide their output strictly periodically and processing of images follows a priority order. 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 the given resources.

To distribute the available processor share among priority-processing algorithms, a decision scheduler has been developed [4]. The decision scheduler aims at maximizing the total progress of the algorithms on a frame basis. It therefore divides the available resources within a period into fixed-sized quanta, termed time slots, and dynamically allocates these time slots to the algorithms. The 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. To maximize the overall output quality of an application, we need support for control strategies of its decision scheduler through (i) mechanisms for dynamic resource allocation to its algorithms and (ii) efficient implementations of these mechanisms. Strategies and mechanisms for dynamic resource allocation have been addressed in [4] and [5], respectively.

During run time, we provide temporal isolation between applications through budgets which are allocated to applications. These budgets are provided by constant-bandwidth servers (CBS) [6] to guarantee a minimum processor share to our application. We inherently have a two-level scheduled system, i.e. a global scheduler to assign a server to the processor, a fixed-priority scheduler [7] for tasks within an application, e.g. SVAs. The global scheduler isolates applications in a temporal manner. The decision scheduler uses the fixed-priority scheduler provided by the platform to distribute the application's processor share by trading off the priorities of the SVAs. We extend this framework with support for *reclaiming* of unused resources.

A. Problem Description

We consider multimedia applications that are composed of a set of competing, independent priority-processing algorithms and a decision scheduler. These algorithms are running on a virtual platform which has a guaranteed processor share at its disposal. This virtual platform is implemented by means of a *server*, which defines a budget

¹ Martijn M. H. P. van den Heuvel, Mike Holenderski, Reinder J. Bril and Johan J. Lukkien 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; m.holenderski@tue.nl; r.j.bril@tue.nl; j.j.lukkien@tue.nl)

allocation and enforcement policy for the serviced application.

To migrate and integrate our existing priority-processing application [5] into a virtual platform, it needs support for (i) triggering its decision scheduler relative to *virtual time* and (ii) efficient redistribution of unused resources. Virtual time is measured with respect to the consumption on of the application's budget.

B. Contributions

The contribution of this paper is fivefold. First, we present a light-weight virtualization technique based on processor reservations and *two-level* hierarchical scheduling. Secondly, we decouple timer management local and global to an application to limit the temporal interference of timers belonging to suspended applications. Thirdly, we extend our timer management with a novel virtual timer mechanism which generalizes local and global timer management. Fourthly, we implement and evaluate our reservation-based framework in a real-time microkernel, $\mu\text{C}/\text{OS-II}$ [8]. Finally, we show the mapping of a priority-processing-based video application on a virtual processor and show that such greedy SVAs can take advantage of reclaimed processor resources.

C. Overview

The outline of the remainder of this paper is as follows. First, Section II discusses related work. Section III presents the mapping of a priority-processing application on a virtual platform. Section IV presents our novel timer management module for our reservation-based scheduling framework. Section V presents our implementation of a light-weight virtualized platform by means of a CBS in $\mu\text{C}/\text{OS-II}$. Section VI evaluates the overheads of our reservation-based framework and shows that the addition of a budget reclaiming mechanism increases the output quality that a priority processing application can achieve. Section VII revisits our assumptions and discusses directions for future work. Finally, Section VIII concludes this paper.

II. RELATED WORK

In this section we first consider the distinguishing characteristics of priority processing compared to more traditional approaches to SVAs. Next, we compare dynamic-resource-allocation mechanisms with reservation-based resource management. We subsequently present alternative virtualization approaches. Finally, we present existing resource reclaiming mechanisms which extend reservations. We will reuse one of these mechanisms to maximize the output of priority processing on a virtualized platform.

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 [9], [10]. Buffering is inherent to 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. Unlike the traditional approach, priority processing does not inherently require buffering.

B. Temporal Isolation

The dynamic-resource-allocation *mechanisms* described in [5] support *control strategies* for SVAs based on priority processing. Whereas these 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 the system [5]. As an example, the application specific control strategy is addressed by the decision scheduler [4], a constituent of the media application.

Dynamic resource allocation facilitates *isolation* between priority-processing algorithms, i.e. temporary or permanent faults occurring in one algorithm cannot hamper the execution of other algorithms [5]. It therefore has much in common with reservation-based resource management [1], [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 premature termination of SVAs at the end of a frame-period requires synchronization between processing and time. In this paper we complement dynamic resource allocation with reservation-based resource management to facilitate isolation between multiple multimedia applications.

C. Virtualization and Timer Management

Virtualization techniques in which a guest operating system is hosted by a hypervisor or micro-kernel have become widely adopted in embedded systems to compose a system from independently developed and tested components. However, virtualization has shown to give considerable overheads [12], [13], i.e. interrupt latencies increased with an order of magnitude. To avoid such overheads we extended a real-time microkernel, $\mu\text{C}/\text{OS-II}$ [8], with light-weight virtualization mechanisms based on the CBS [6]. The CBS guarantees a fraction of the processor time to applications whose computation time cannot be easily bounded statically, i.e. applications containing greedy algorithms. We extend the CBS' reservation mechanism with support for virtual-timers.

The notion of a virtual timer already exists in the Portable Operating System Interface (POSIX) description [14]. Each application running on a POSIX-compliant platform has a

virtual timer available that expires relative to the processor time consumed by its process. When the virtual timer expires, a signal is sent to the process. This signal is queued and the time that the signal is handled depends on when the process is selected for execution by the scheduler. Signals, as described in the POSIX standard, are a form of inter-process communication. However, we do not only require signaling of our application relative to its consumed budget at the local level, but also to enforce that it does not exceed its reserved resource share at the global level. We therefore propose a more general notion of virtual timers that supports virtual time management globally and locally to an application.

D. Resource Reclaiming

A constant bandwidth server (CBS) provides a guaranteed fraction of the processor. When a budget depletes, a next budget becomes immediately available at a lower priority and may be consumed in advance. While a priority-processing application can handle fluctuating resource availability after receiving a minimum guarantee [3], its SVAs cannot consume these forwardly offered budgets. The reason for this is that (i) it may result in less resources for next frames and (ii) we do not know in advance how large the fraction of the forwardly offered budgets is before a frame is prematurely terminated. The latter information is necessary to prevent depletion of a budget during finalization of a frame, i.e. writing the produced output to the frame buffer.

Several resource reclaiming techniques for the CBS in which the reclaimed budgets become conditionally guaranteed [15] are proposed and compared in [16]. A greedy application may, for example, start consuming unused budgets of other applications, i.e. remaining budgets of servers with an earlier deadline [16]. For ease of presentation we will reuse this reclaiming mechanism to demonstrate that priority processing can efficiently exploit these reclaimed processor resources.

III. MAP AN APPLICATION ONTO A VIRTUAL PLATFORM

We consider a priority processing application attached to a server which provides a virtual share of a single processor platform. We first recapitulate our application model for priority processing for the case where this application has the entire processor at its disposal [4] and subsequently show the mapping of this model onto a virtual platform.

A. Priority-processing on an Entire Processor

We assume a set Γ of m strictly periodically released tasks $\tau_1, \tau_2, \dots, \tau_m$, 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 $T_i \in \mathbb{R}^+$, a (relative) deadline $D_i \in \mathbb{R}^+$, a computation time $C_i \in \mathbb{R}^+$, where $C_i \leq D_i \leq T_i$, and a phasing $\varphi \in \mathbb{R}^+ \cup \{0\}$, representing the start-time of the first job 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 terminated prematurely. The epilog is ideally constant and as small as possible. Typically, it is infeasible to compute a realistic estimate of the worst-case computation time of the scalable part 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 $Q \leq T$ of processor time 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.

$$\sum_{i=1}^m (C_{i,basic} + C_{i,epilog}) \leq Q$$

We assume 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. 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 1.

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

B. Mapping onto a Server

We attach a CBS to a priority-processing application. This application consists of a decision scheduler and at least two independent SVAs. The SVAs are not blocked by their input or output and share no resources except the processor. Each SVA accounts its own progress and updates the progress during runtime. Based on these progress values, the decision scheduler heuristically allocates time slots to SVAs.

Within a server, all application tasks are scheduled by a *fixed-priority preemptive scheduler*, so that at each time instant the application's task with the highest priority and with a pending load is assigned the processor. The decision scheduler is mapped on a task and is assigned the highest priority, so that it has full control over its SVAs. Upon activation, it changes the SVAs' task priorities according to its choice to allocate an SVA to a time slot [5]. All tasks comprising the priority-processing application are assigned to the same server.

A server has a replenishment period, P_b , for its budget, Q_b . The SVAs' consumed time is accounted to (and subtracted from) that budget. When a server uses up its budget Q_b within an interval of P_b , it is said to be depleted. At the end of the current interval P_b , the server will obtain a new budget Q_b and is said to be replenished. We distinguish two types of reservations based on their behavior with respect to depletion and replenishment: (i) a *hard reservation* which on depletion cannot be scheduled until it is replenished and (ii) a *soft reservation* which on depletion can be scheduled for execution along with spare processor time and other depleted reservations [1].

All SVAs are synchronous with the same frame period, T , which is a multiple of their server period, i.e. $T = i \times P_b$ with $i \in \mathbb{N}^+$. Activation of the decision scheduler is a virtual timed event and is triggered after consumption of a time slot, Δt_s , relative to budget Q_b . For example see Figure 2 where the video-frame period $T = 40ms$, the application is provided with a budget of $Q_b = 5.5ms$ each period $P_b = 20ms$, and $\Delta t_s = 1ms$.

Fig. 2. Example of budget replenishments and virtual timers, with $T = 40ms$, $P_b = 20ms$, $Q_b = 5.5ms$, and $\Delta t_s = 1ms$.

We need virtual timers to trigger timed events relative to the consumed budget to activate the decision scheduler. We therefore extended $\mu C/OS-II$ to implement such support. This timer management, including virtual timers, forms a basis to implement a reservation-based scheduler.

IV. TIMER MANAGEMENT FOR RESERVATION-BASED SCHEDULING

Intrinsic to our virtual platform is support for hierarchical timer management. We store timers in a queue ordered by expiration time. We express their expirations relative to each other by representing each expiration time relative to the expiration of the previous timer. The expiration time of the head timer is relative to the current time. Although this approach is not restricted to any specific hardware timer, we assume a high-precision periodic timer, which is in line with $\mu C/OS-II$. At every tick interrupt of the periodic timer, the time of the head in the queue is decremented. This timer

representation scheme is explained in [17]. In this paper we extend this approach for inclusion in a reservation-based framework, i.e. first we consider hierarchical timer support and subsequently we introduce virtual timers.

A. Hierarchical Timer Queues

To support hierarchical scheduling, we introduce (i) a *system queue* which keeps track of timers such as replenishment of server budgets, and (ii) one *local queue* for each server which keeps track of timers such as task deadlines or the arrival of periodic tasks within a server. At any time at most one server can be running on the processor; all other servers are *switched out*. When a server is switched out, its local queues are deactivated to make sure that the timers local to switched-out servers do not interfere with the currently running server. In this configuration the hardware timer drives two timer queues, i.e. the local queue of the active (running) server and a system queue.

When the running server is switched out, then the running server queue is replaced by the queue belonging to the newly scheduled server. As a result, the queue of the switched-out server will be *paused*, and the queue of the scheduled server will be *resumed*. To keep track of the time which has passed since the last server switch, we introduce one additional stopwatch queue.

The stopwatch queue contains one timer for each switched-out server. The accumulated time between the head of the stopwatch queue and a stopwatch timer represents the time since the corresponding server was switched out. At every tick the timer at the head of the stopwatch queue is incremented. Whenever a server is scheduled, its local queue is synchronized with the stopwatch, i.e. all timers in its local queue which would have expired if the server was running are handled. Subsequently its stopwatch timer is deleted from the stopwatch queue. When a server is switched out, its stopwatch timer is set to 0 and inserted at the head of the queue.

When a server executes and its budget depletes, a timer expires and triggers a handler which ensures that a server does not exceed its budget. We could resolve the budget-depletion timers in a way similar to [18], i.e. removing the budget-depletion timer for a particular server from its local queue every time this server is switched out and inserting it back when the server is scheduled. However, we opt for an alternative approach.

B. Virtual Timers

In Section III we identified the need to trigger our decision scheduler on a time-slot basis. These virtual timers should expire relative to the consumption of the server budget. In this section we present a general approach for handling both budget-related and application-related timers. Our notion of virtual timers avoids removing timers upon server switching and is therefore more efficient than that of [18].

We implement virtual timers by adding a *virtual queue* for each server. In this new configuration, at every tick the heads of at most four queues are updated: a system queue, active

server queue, stopwatch queue, and active-server virtual queue. The dedicated server queue managing virtual timers does not need to get synchronized when a server is resumed, because a switched-out server does not consume its budget. Figure 3 shows an example of our timer-management module with support for hierarchical-scheduling and virtual timers.

Fig 3. Example of our timer queue setup for reservations.

V. VIRTUAL PLATFORM IMPLEMENTATION

Given our timer-management module described in the previous section, we built a two-level hierarchical scheduling framework within $\mu\text{C}/\text{OS-II}$. Extending $\mu\text{C}/\text{OS-II}$ with reservation support requires the identification and realization of the following concepts:

1) *Applications*: $\mu\text{C}/\text{OS-II}$ tasks are bundled in groups of sixteen to accommodate efficient fixed-priority preemptive scheduling [8]. An application is therefore naturally represented by such a group. Each application is allocated a server that manages its reserved budget. Servers are scheduled by means of an earliest-deadline-first (EDF) scheduler [7].

2) *Fixed-priority scheduling* [7]: After allocating a server to the processor, $\mu\text{C}/\text{OS-II}$'s fixed-priority task scheduler determines the highest priority ready task within the server. The decision scheduler is assigned the highest priority within a priority-processing application and selects an SVA to execute for the duration of a given time slot. It manipulates task priorities accordingly [5].

In the remainder of this section we first describe the implementation of hard reservations. Next, we will show how to extend these to soft reservations by means of a CBS.

A. Periodic Bandwidth-preserving Reservations

In this section we consider the deferrable server [19], which provides a periodic processor reservation. We present this server, because it implements a hard reservation, whereas a CBS extends its design to a soft reservation. A deferrable server is *bandwidth-preserving* and can be in one of the states shown in Figure 4.

Bandwidth preservation means that a server is suspended into a *suspended* state when its workload is exhausted. In the meantime it preserves the remaining budget to service future arriving tasks during the remainder of the current budget period. The period of a server, P_b , serves as a deadline for a budget Q_b , i.e. the global timer queue is sorted by server deadlines. All servers with pending load and remaining budget are *ready to execute*. According to the EDF policy, the server with the highest (dynamic) priority among all ready servers, i.e. the earliest deadline, is scheduled and *running* on the processor, either until its workload is exhausted, or it is preempted by a higher priority server or its budget depletes. Budget depletion is implemented using a virtual timer. When the timer expires the server's budget is depleted and the server is de-allocated the processor. Figure 4 illustrates this behavior.

Fig 4. State transitions of a deferrable and constant bandwidth server. The replenishment transitions from the ready, waiting and running states to the same state are not shown.

A bandwidth preserving server may need to provide its preserved budget to tasks which arrive while it is *suspended*. Hence, we cannot simply deactivate its local queue when its current workload is exhausted. One option is to keep its local queue active, but this would increase the interference from switched-out bandwidth preserving servers. We therefore propose an alternative solution, based on *wake-up timers*. We observe that the timer at the head of the server's local queue represents the time when the server may need to provide its budget. When a server is suspended, we insert a wake-up timer into the system queue with the expiration time of the first timer in the server's local queue. When a wake-up timer expires, we change the server state to ready, allowing it to be scheduled by the global scheduler and handle the corresponding local event.

Many applications handle sporadic or a-periodic (external) events to interact with their environment. To prevent that handling such events causes overload, one may enforce a minimum inter-arrival time between two consecutive events. Although during run-time their exact arrival time remains unknown, we can reuse our wake-up timer mechanism based on the observation that for sporadic tasks the minimum inter-arrival time is known upfront. Hence, we know when to expect the earliest activation of a task, i.e. one period later than the previous release. When the previous release is

unknown, we must initially expect the sporadic event at any time. After one arrival, we can insert timers into the local queue to enable sporadic events and wake up the server. Bandwidth preserving servers may therefore decrease the average response time to an application [19], which is especially advantageous for multimedia applications.

B. Constant Bandwidth-preserving Supply

For the special case where we map a priority-processing application on a deferrable server, a server never reaches the waiting state. Such a greedy application which is constrained by a reservation does not behave differently than a periodic idling server, i.e. it behaves similarly as a periodic task without self-suspension. All its SVAs are strictly periodic and synchronous with their server period. However, not all applications may request their resources at the same time, so that unused budgets can be reclaimed by other applications.

The constant bandwidth server (CBS) [6] builds on top of a global EDF scheduler and absorbs unused processor resources. The system queue contains all servers ordered by their absolute deadline. A CBS is bandwidth preserving, but has no periodic replenishment. Hence, contrary to a deferrable server, a CBS does not get blocked in its depleted state, see Figure 4. In our implementation budget depletion as well as budget replenishment is implemented using a single virtual timer. When this timer expires the server's budget is immediately replenished and its deadline is postponed by its period, P_b . Effectively, a task within the CBS remains eligible for execution when the server's deadline is postponed, albeit at a lower priority. Postponing a server's deadline is implemented by updating deadline timers.

A CBS may advance allocated processor supplies due to the *work-conserving* nature of the EDF scheduling policy [7], i.e. it never idles away processor time if there is a task ready to execute. However, the explicit synchronization between processing and time upon premature termination of the processing of a video frame requires that we know upfront that we have sufficient supply to complete the epilog subtask of an algorithm. We guarantee this by prematurely terminating processing of a frame, so that the epilog can be executed *before* the server's budget, Q_b , depletes. This means that we artificially put the CBS in a waiting state, so that it preserves a negligible small budget until the next frame period. At the next frame arrival, the server's budget is replenished and its deadline is *automatically* synchronized with the frame period.

In addition, we allocate all reclaimed resources, i.e. allocated and unused budgets, to a priority-processing application. Such a reclaiming mechanism can be implemented using our timer management support by traversing the system queue. While traversing the system queue, the priority-processing application receives all remaining budgets Q_i in a deadline order of the servers that are earlier in the queue than the server belonging to priority processing with deadline P_b . When all these budgets are exhausted, then we start consuming our own budget. All consumed budgets are guaranteed to be supplied before the

deadline of the currently processed video frame expires by virtue of the EDF schedulability analysis [7], [16].

VI. EVALUATION

We created a port for $\mu\text{C}/\text{OS-II}$ to the OpenRISC platform [20]. This platform comes with a cycle-accurate platform simulator. The simulator allows software-performance evaluation via a cycle-count register. The measurement accuracy is approximately 5 instructions. We configured the processor speed to 400 MHz and the hardware clock to 1 kHz. These values correspond to the hardware found in state-of-the-art consumer devices which are capable to render video streams, e.g. mobile phones [21].

In the remainder of this section we first show the overhead imposed by our virtual-platform support, compared to a de facto fixed-priority $\mu\text{C}/\text{OS-II}$ system. Secondly, we show the effect of the CBS' processor reclaiming mechanism on the priority-processing application.

A. Virtual platform overhead

Since the main objective of a real-time operating system is to provide time predictability, it is important to know whether our extended real-time operating system behaves in a time-wise predictable manner. In virtualized systems the fractional overhead for tick-based schedulers can be considerable [13]. We therefore investigate the interrupt and scheduling overheads of our reservation-based scheduling framework.

1) Computational Complexity:

The timer interrupt handler synchronizes all active queues with the current time. For each expired timer it executes the corresponding handler and calls the scheduler. Handling timers may require traversing a queue. For example, when a task period event expires, an event representing the next periodic arrival is inserted into a queue. In our implementation, inserting an event requires a linear traversal of the queue. In our analysis we assume that the cost of traversing a queue is negligible compared to the fixed overheads associated with handling the event, and treat the overhead of handling an event as constant. There are two global queues, i.e. a system and a stopwatch queue, and two queues for the currently running server. The system queue contains only one budget replenishment and at most one wake-up timer per server. Its length is therefore proportional to the number of servers, N . The length of the active local queue is linear in the number of application tasks assigned to the server. Similarly, since the active-server virtual queue contains one depletion timer and at most one virtual timer for each task at each time instant, its length is linear in the number of tasks assigned to the server, m .

The scheduling overhead of our framework is linear in the number of servers, N , since we need to traverse the system queue in order to determine the earliest-deadline ready server. Selecting the highest priority task within a server is linear in the number of tasks assigned to a server, m . Although this can be done in constant time by selecting the first task from a

ready queue ordered by priority, it comes at the cost of a linear insertion of tasks into the ready queue whenever a task becomes ready. Given the constant overhead for handling a timer, the overhead of the tick handler and scheduler is proportional to the lengths of the four active queues, i.e. $O(N + m)$.

The overhead of context switching between servers depends on the number of timers inside the stopwatch queue and the server queue of the running server, which needs to be synchronized with the stopwatch. The stopwatch queue contains at most one stopwatch timer per server, i.e. N timers.

2) Simulation Results:

The scheduling overhead, including context switching, heavily depends on the chosen server and task parameters. We therefore use simulations to gain insight into the average overheads. We consider a setup with an increasing number of servers and tasks within a server. The worst-case computational overhead to handle expiring timers occurs when multiple timers expire at the same time. In our experiments, we therefore choose all server periods and budgets equally large and simultaneously released. A similar setup is chosen for tasks within a server. We show that each server bounds its overheads to other servers in the system.

Figure 5 relatively compares (i) the total timer handling and scheduling overheads for a system comprising multiple application serviced by a deferrable server to (ii) a plain fixed-priority-scheduled microkernel where all tasks belonging to different application are scheduled by a single scheduler. The overhead for reservation-based scheduling is lower than for a plain $\mu\text{C}/\text{OS-II}$ system. This is because our reservation-based system groups tasks in applications together, so that the overhead for local scheduling of (a relatively small number of) tasks is accounted to each individual application. This difference is especially visible for an increasing number of applications each having multiple concurrent tasks.

We repeated the same experiment for a system where applications are serviced by a CBS instead of a deferrable server. Figure 6 presents the corresponding results. These are similar to the results for the deferrable server in Figure 5. However, the CBS brings a minor additional overhead compared to a system with deferrable servers. This overhead is due to the extra server and task switches which occur in the case of CBS when it exploits the unused capacity, whereas the deferrable server simply waits until its next replenishment. Moreover, a CBS may require inserting more deadlines into the ready queue. However, our CBS-based scheduler also outperforms a fixed-priority-scheduled microkernel.

3) Memory Complexity

Our modular design of the reservation-based extensions in $\mu\text{C}/\text{OS-II}$ makes it possible to enable or disable the support for reservations and different server types during compilation. The complete implementation, which includes timer management, EDF-based scheduling, deferrable servers and CBSes, consists of approximately 2000 lines of code (excluding comments and blank lines). A basic $\mu\text{C}/\text{OS-II}$

setup consists of 8330 lines of code. Note that our timer management can replace the existing timer mechanisms in $\mu\text{C}/\text{OS-II}$. Furthermore, it provides a framework for easy implementation of other scheduler and servers types.

The memory footprint of our reservation-based extensions is 6 kB, compared to $\mu\text{C}/\text{OS-II}$'s de-facto size which can be scaled down to 10 kB. Furthermore, $\mu\text{C}/\text{OS-II}$ predefines a fixed amount of stack space for each task. Each application has the additional memory footprint of a server data structure.

Fig. 5. Total scheduling and timer handling overheads for a reservation-based hierarchical scheduling framework (HSF) based on EDF-scheduled deferrable servers versus a de facto fixed-priority scheduled $\mu\text{C}/\text{OS-II}$.

Fig. 6. Total scheduling and timer handling overheads for a reservation-based hierarchical scheduling framework (HSF) based on an extended $\mu\text{C}/\text{OS-II}$ with EDF-scheduled deferrable servers versus CBSes.

B. Reclaiming unused processor reservations

We now consider a fully loaded system, i.e. all available processor resources are reserved for dedicated applications. Although today's consumer devices feature many applications, these typically do not simultaneously require all processor resources. In occasional situations of overload, we guarantee a minimum output quality of our video application. Based on our measurements in [5], we reserve a guaranteed budget of 10 ms every period of 40 ms for our priority-processing application, which is sufficient to complete an SVA's mandatory processing part with a basic output quality and a fixed frame rate of 25 frames per second. These

guaranteed resources effectively provide a virtual processor to our application which is four times slower than the full processor speed. In other words: if we would have an entire processor available, similar to [5], then our new frame period has shrunk to one fourth of its original length. Hence, reclaiming unused resources can be thought of as increasing the frame period of an SVA running on a fully available processor. We therefore simulated our priority-processing application for different frame periods to demonstrate the effect of our budget reclaiming mechanism on its SVAs.

As a leading example, we reconsider a basic priority-processing application composed of two independent SVA [4]. These SVAs are fed with standard video sequences from the Video Quality Experts Group (VQEG). Since priority processing has a greedy nature, it exploits all unused processor time that is reclaimed by the CBS. Figure 7 shows that an SVA for sharpness enhancement [3], which processes VQEG sequence 5 and 6, reaches a higher progress value when given more processing time per frame, i.e. when the remaining processor time is not fully allocated and used by other applications. If all 30 ms of remaining processor time in a period of 40 ms is reclaimed, as in the bottom case of Figure 7, then priority processing has a full processor at its disposal.

Fig. 7. Distribution of an SVA's additional progress for 10, 20 or 30 ms more processing time per frame. Measurement data is obtained from an application comprising two SVAs for sharpness enhancement [3], [5] on video sequence 5 and 6 from the Video Quality Experts Group (VQEG).

VII. DISCUSSION

We presented a mapping of a priority-processing application containing independent SVAs on a light-weight virtual platform. In this section, we briefly discuss further challenges towards an efficient deployment of priority processing on an embedded platform which needs to be shared with third-party applications.

A. Dependent Applications

In this paper we assumed independent applications and independent SVAs within an application. By creating a chain of time-driven SVAs, separated by buffers, we can simply lift the assumption on independence of SVAs at the cost of increasing the end-to-end delay of the final output frame [4].

In practice, however, applications may also share resources, e.g. operating-system services, shared (frame) buffers and other memory mapped devices. This means that resource sharing expands across a budget which calls for specialized resource access protocols. When a task accesses such a shared resource, one needs to consider the priority inversion between servers as well as local priority inversion between tasks within the server. To accommodate such resource sharing, three synchronization protocols have been proposed in literature, which have been qualitatively and quantitatively compared in [22] and [23], respectively. We consider further elaboration on this topic out of the scope of this article.

B. Handling Fluctuating Resource Availability

In Section VI-B we showed that priority processing can provide a higher output quality by exploiting reclaimed processor resources. However, a large fluctuation in resource availability may result in a fluctuation of the perceived output quality [9]. Since each application is assumed to be allocated a minimum guaranteed share, large resource fluctuations are a likely result of a change in operational mode, e.g. turning on or off a media application. We leave further considerations to balance reclaimed resources among competing applications on a fully loaded consumer platform as future work.

VIII. CONCLUSION

Virtual platforms become indispensable to guarantee secure and predictable behavior in today's heavily loaded consumer platforms. In this paper we presented the mapping of a priority-processing-based video application on a virtual processor. This application consists of a decision scheduler and a set of independent scalable video algorithms (SVAs). Compared to full virtualization, we presented light-weight architectural support to (i) provide temporal isolation between priority-processing applications by means of a constant bandwidth server (CBS) and (ii) efficiently distribute the available processor reservations by means of a virtually timed decision scheduler. We inherently have a *two-level* hierarchical scheduling framework, i.e. a global scheduler to assign a reservation to the processor and a fixed-priority scheduler for tasks within an application, e.g. SVAs.

We developed a solution to decouple timer management local and global to an application. This limits the temporal interference of timers belonging to suspended applications. We extended our approach with a novel virtual timer mechanism which generalizes local and global timer management to an application. Using this support, we implemented different strategies to provide processor reservations, i.e. deferrable and constant bandwidth servers.

Our design has been implemented in a real-time operating system, $\mu\text{C}/\text{OS-II}$, and evaluated on the OpenRISC platform. We showed that a reservation-based scheduling framework can significantly reduce timer handling and scheduling overheads on heavily loaded consumer platforms where multiple applications need to share resources. Moreover, we

showed that priority-processing algorithms can take advantage of reclaimed processor resources. This application-level support in reservation-based approaches makes virtualization a promising solution for future consumer electronics.

REFERENCES

- [1] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: A resource-centric approach to real-time and multimedia systems," in Proc. Conference on Multimedia Computing and Networking, pp. 150–164, January 1998.
- [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," in Proc. Int. Conference on Consumer Electronics (ICCE), January 2008.
- [4] S. Schiemenz, "Echtzeitsteuerung von skalierbaren Priority-Processing Algorithmen," Tagungsband ITG Fachtagung - Elektronische Medien, pp. 108–113, March 2009.
- [5] M. M. H. P. van den Heuvel, R. J. Bril, S. Schiemenz, and C. Hentschel, "Dynamic resource allocation for real-time priority processing applications," IEEE Transactions on Consumer Electronics (TCE), vol. 56, no. 2, pp. 879–887, May 2010.
- [6] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in Proc. Real-Time Systems Symposium (RTSS), pp. 4–13, December 1998.
- [7] G. C. Buttazzo, "Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)," Springer-Verlag TELOS, 2004.
- [8] J. J. Labrosse, MicroC/OS-II. R & D Books, 1998.
- [9] C. C. Wüst, L. Steffens, W. F. Verhaegh, R. J. Bril, and C. Hentschel, "QoS control strategies for high-quality video processing," Real-Time Systems Journal, vol. 30, no. 1-2, pp. 7–29, 2005.
- [10] W. Zhao, C. C. Lim, J. W. Liu, and P. D. Alexander, Overload Management by Imprecise Computation, series: Imprecise and Approximate Computation, Kluwer Academic, vol. 318, ch. 1, pp. 1–22, 1995.
- [11] C. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in Proc. Int. Conference on Multimedia Computing and Systems (ICMCS), pp. 90–99, May 1994.
- [12] F. Armand and M. Gien, "A practical look at micro-kernels and virtual machine monitors," in Proc. Consumer Communications and Networking Conference (CCNC), pp. 1–7, January 2009.
- [13] S. Yoo, Y.-P. Kim, and C. Yoo, "Real-time scheduling in a virtualized CE device," in Proc. Int. Conference on Consumer Electronics (ICCE), pp. 261–262, January 2010.
- [14] "Information technology - Portable Operating System Interface (POSIX) Base Specifications," ISO/IEC/IEEE 9945 (First edition), pp. c1–3830, September 2009.
- [15] R. J. Bril, W. F. J. Verhaegh, C. C. Wüst, "A Cognac-Glass Algorithm for Conditionally Guaranteed Budgets," in Proc. Real-Time Systems Symposium (RTSS), pp. 388–400, December 2006.
- [16] M. Caccamo, G. C. Buttazzo, D. C. Thomas, "Efficient reclaiming in reservation-based real-time systems with variable execution times," IEEE Transactions on Computers, vol. 54, no. 2, pp. 198–213, Feb. 2005.
- [17] M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien, "Multiplexing real-time timed events," in Proc. Conference on Emerging Technologies and Factory Automation (ETFA), July 2009.
- [18] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril, "Towards hierarchical scheduling on top of VxWorks," in Proc. Int. Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPert), pp. 63–72, July 2008.
- [19] J. Strosnider, J. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," IEEE Transactions on Computers, vol. 44, no. 1, pp. 73–91, Jan. 1995.
- [20] M. Bolado, H. Posadas, J. Castillo, P. Huerta, P. Sánchez, C. Sánchez, H. Fouren, and F. Blasco, "Platform based on open-source cores for industrial applications," in Proc. Conference on Design, Automation and Test in Europe (DATE), pp. 21014, 2004.
- [21] N. Uchihara, H. Kasai, Y. Suzuki, Y. Nishigori, "Asynchronous prefetching streaming for quick-scene access in mobile video delivery," IEEE Transactions on Consumer Electronics, vol. 56, no. 2, pp. 633–641, May 2010.
- [22] M. Behnam, T. Nolte, M. Åsberg, and R. J. Bril, "Overrun and skipping in hierarchically scheduled real-time systems," in Proc. Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp. 519–526, August 2009.
- [23] M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Protocol-transparent resource sharing for hierarchically scheduled real-time systems," in Proc. Int. Conference on Emerging Technologies and Factory Automation (ETFA), September 2010.

BIOGRAPHIES

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

He is a member of the IEEE Consumer Electronics Society.

Mike Holenderski is a Ph.D. student at the Eindhoven University of Technology, the Netherlands. He received his B.Sc. in 2003 and M.Sc. (with honors) in 2007, both from the Eindhoven University of Technology. His main research interests are in the area of reservation-based multi-resource scheduling in embedded real-time systems. He is a member of the IEEE Consumer Electronics Society.

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.

Johan J. Lukkien is head of the System Architecture and Networking Research group at Eindhoven University of Technology since 2002. He received M.Sc. and Ph.D. from Groningen University in the Netherlands. In 1991 he joined Eindhoven University after a two years leave at the California Institute of Technology. His research interests include the design and performance analysis of parallel and distributed systems. Until 2000 he was involved in large-scale simulations in physics and chemistry. Since 2000, his research focus has shifted to the application domain of networked resource-constrained embedded systems. Contributions of the SAN group are in the area of component-based middleware for resource-constrained devices, distributed coordination, Quality of Service in networked systems and schedulability analysis in real-time systems.