Virtual Timers in Hierarchical Real-time Systems

Martijn M.H.P. van den Heuvel, Mike Holenderski, Wim Cools, Reinder J. Bril and Johan J. Lukkien
Technische Universiteit Eindhoven (TU/e)
Den Dolech 2, 5600 AZ Eindhoven, The Netherlands

***Abstract*—Hierarchical scheduling frameworks (HSFs) provide means for composing complex real-time systems from well-defined subsystems. This paper describes an approach to provide hierarchically scheduled real-time applications with virtual event timers, motivated by the need for integrating priority processing applications in an HSF. Specifically, the paper proposes a technique to minimize the overhead of event handling in HSFs and outlines a simple implementation.**

I. Introduction

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. Hierarchical scheduling frameworks (HSFs) have been extensively investigated as a paradigm to facilitate this decoupling, see for example [1]. In this paper we 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 *server* is allocated to each application. A global scheduler is used to determine which server should be allocated the processor at any given time. A local scheduler determines which of the chosen application's tasks should actually execute.

Multimedia applications define a well-studied class of real-time applications. To enable cost-effective media processing in software, scalable video algorithms (SVAs) have been developed that allow trading quality for resource needs. The principle of priority processing provides optimal real-time performance for scalable video algorithms on programmable platforms even with limited system resources [2]. 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, followed by less important parts in a decreasing order of importance. After creation of an initial output by a basic function, processing can be *preliminary terminated* at an arbitrary moment in time, yielding the best output for given resources, see Figure 1.

To distribute the available resources, i.e. CPU-time, among independent priority processing algorithms, an application specific strategy has been developed [3]. This strategy is implemented in a *decision scheduler* and aims at maximizing the total relative progress of the SVAs. The relative progress of an algorithm is defined in terms of the fraction of the performed work relative to the consumed budget and the total amount of work to be done in a video frame.

As a leading example, we consider a priority processing application, composed of multiple independent SVAs and a decision scheduler, which divides the available virtual processor resources among the SVAs, as described in [4]. They assume that the application has the full processor at its disposal.

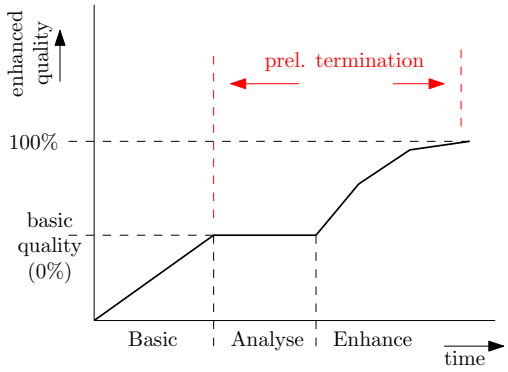

Fig. 1. Priority processing, as a function of time consumption versus output quality, can be divided in three time-frames: 1) produce a basic output at the lowest quality level; 2) Identify the most important image content; 3) Enhance the quality of the output by processing the most important picture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

A. Problem Description

In this paper we consider the scenario in which a priority processing application is provided a virtual share of the available processor resources, by assigning the decision scheduler and the SVAs a single virtual processor.

The decision scheduler implements the control strategy and divides the available processor time within the application budget into fixed-sized quanta termed *time-slots* of the size Δt_s. The control strategy selects the SVA to execute next upon completion of a time-slot, i.e. synchronous with virtual time. Activation of the decision scheduler is triggered by the depletion of a time-slot. Hence, the 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.

B. Contributions

Given the need for virtual timer events on shared virtual platforms, we outline a low-overhead implementation of virtual timers, targeted at embedded systems. Additionally, the outlined solution aims at minimizing the overhead of handling events of inactive applications in HSFs.

C. Outline

The remainder of this paper is as follows. Section II describes the related work. Section III describes the virtual

platform model used as a reference for our implementation directions. Section IV describes an approach to realize virtual timed events. Finally, Section V concludes the paper.

II. Related work

The notion of a virtual timer within an application already exists for POSIX-compliant operating systems [5]. Each process running on such a platform has the availability of a virtual timer that counts processor time used by that process. When the virtual timer expires, a signal is sent to the process. Upon expiration of a timer, the corresponding signal is queued for the corresponding process, whereas arrival of a signal depends on the granularity of the kernel clock. Signals, as described in the POSIX standard, are a form of inter-process communication. Processes are the primitive units for allocation of system resources. Each process has its own address space and one thread of control. When considering our priority processing application, it is natural to map the application on a single process. The decision scheduler and the SVAs are each mapped on its own thread contained in the process, whereas a thread is used as a scheduling unit. We require signalling of the decision scheduler's thread upon expiration of the virtual timer, instead of the main process. Although the concept is similar, we require a more general notion of virtual timers.

Partitioning the system to independent subsystems, which are each provided with a virtual platform, is currently researched in two slightly different directions. On the one hand, directions go towards hierarchical scheduling schemes. Hierarchical real-time system development is based on sound analysis and well-defined application interfaces [1]. On the other hand, virtual real-time operating systems (RTOSs) are investigated to obtain a strong partitioning of the system [6], [7]. Main challenges with respect to satisfying real-time constraints in virtualizing a RTOS are (*I*) increasing responsiveness with respect to hardware interrupts and (*II*) synchronization of virtual machine related timer events. Both issues require a low level virtual machine monitor to manage interrupts and events messages [7]. Issue (*I*) includes the problem that an interrupt can be generated for a particular virtual machine which is not assigned to the processor at that moment in time. The virtual RTOS is not allowed to disable interrupts, which entails additional demands on the virtual machine's monitoring layer. The virtual machine monitor has to queue all interrupts for an inactive RTOS and block the interrupts for an RTOS when it requests to temporarily disable interrupts. Issue (*II*) relates to managing event queues. When a virtual RTOS is active, it can handle all timer events directly. All queued timer events for a particular inactive virtual RTOS, as addressed by issue (*I*), must be synchronized with the local event queue upon activation of the virtual machine. Support for virtual timers, as required by our priority processing application, is lacking in the description in [6], [7].

[8] presents a novel design for managing timed event queues, RELTEQ, applicable for embedded operating systems demanding low memory and processor overhead. Reservation based real-time systems provide applications with the facility to request their remaining budget within the current replenishment period. Reservation based kernels rely on mechanisms for *admission control*, *scheduling*, *monitoring* and *enforcement* [9]. Note that virtual timed events are different from monitoring the consumed budget within an application. Although RELTEQ can be exploited to support budget monitoring, it does not support the generalized concept of virtual timed events. Enforcement of budgets requires expiration of timers upon depletion of the budget. [10] implemented enforcement timers by setting a single timer on activation of a server, indicating the depletion of the server's budget. The enforcement timer is set to the minimum value of the remaining budgets of all levels in the hierarchical resource chain, e.g. levels in an HSF. Every budget has its own replenishment timer. Finally, all timers are added to a single global event queue.

[11] keeps track of budget depletion by using separate event queues for each server in the HSF by means of absolute times. On activation of a server, an event indicating the depletion of the budget, i.e. the current time plus the remaining budget, is added to the server event queue. On preemption of a server, the remaining budget is updated according to the time passed since the last server release and the budget depletion event is removed from the server event queue. When the server's budget depletion event expires, the server is removed from the server ready queue, i.e. it will not be rescheduled until the replenishment of its budget.

In this paper we show how to extend the RELTEQ [8] approach to manage virtual timed event queues.

III. Virtual Platform Model

Given an HSF mapped on a single processor, we consider a priority processing application, attached to a server within the HSF. For simplicity, we assume an *idling periodic server* [12], however the proposed approach is expected to be easily adaptable to other server types. We say that tasks assigned to a server consume processor time *relative to the server's budget* to signify that the consumed processor time is accounted to (and subtracted from) that budget.

Given the priority processing application, the decision scheduler task is assigned the highest priority, such that upon activation it can immediately preempt the SVAs. The SVAs are each mapped on a strictly periodic task. All SVAs are synchronous with the same period, P_f, i.e. each period the SVAs start with a new video frame and at the end of a period the processing is terminated. The SVAs are not blocked by their input and output and share no resources except the processor. All tasks comprising the priority processing application are assigned to the same server.

A server has a replenishment period, P_b, and a budget, Q_b. Activation of the decision scheduler, i.e. a virtually timed event, is triggered after consumption of a time-slot, Δt_s, relative to the budget Q_b. For convenience we assume that P_f is a multiple of P_b and has the same phasing. For example, the video frame rate $P_f = 20ms$, the application is provided with a budget $Q_b = 5.5ms$ every period $P_b = 10ms$, and $\Delta t_s = 1ms$. This scenario is sketched in Figure 2.

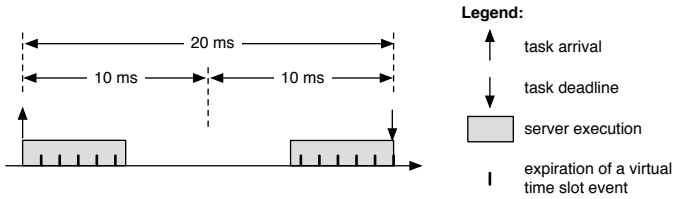

Fig. 2. Example of budget replenishments and virtual events, with $P_f = 20ms$, $P_b = 10ms$, $Q_b = 5.5ms$, and $\Delta t_s = 1ms$.

IV. Proposed Approach

We start this section by summarizing the RELTEQ [8] approach to multiplexing timed events on a single hardware timer. Then we describe how RELTEQ can be extended to support hierarchical scheduling. Finally, we outline an efficient RELTEQ implementation of virtual timers.

A. Basic RELTEQ timer management

RELTEQ stores the arrival times of events relative to each other, by expressing the arrival time of an event relative to the arrival time of the previous event. The arrival time of the head event is relative to the current time, as shown in Figure 3. While RELTEQ is not restricted to any specific hardware timer, in this paper we assume a periodic timer. At every tick of the periodic timer the time of the head event in the queue is decremented.

Two operations can be performed on an event queue: new events can be inserted and the head event can be popped. When a new event e_i with absolute time t_i is inserted, the event queue has to be traversed, accumulating the relative times of the events until a later event e_j is found, with $t_i < t_j$, where t_i and t_j are both absolute times. When such an event is found, then *(i)* e_i is inserted before e_j, *(ii)* its time is set relative to the previous event, and *(iii)* the arrival time of e_j is set relative to e_i. If no later event was found, then e_i is appended at the end of the queue, and its time is set relative to the previous event.

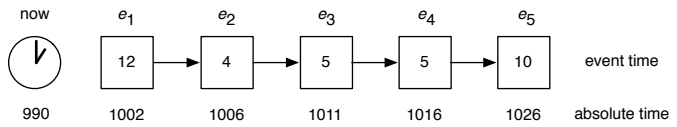

Fig. 3. Example of the RELTEQ event queue.

The first event: The arrival time of the first event is expressed in *absolute* time. To prevent the first event from overflowing, RELTEQ inserts *dummy events* at times when the absolute time would overflow, as shown in Figure 4.

The time will overflow once in 2^n ticks (assuming an n-bit time representation), requiring to insert one dummy event every 2^n ticks. Since the number of proper events within that time interval is likely to be high, the overhead of using dummy events to handle absolute time overflows is small.

[8] also describes how to use dummy events to provide unbounded interarrival times between events.

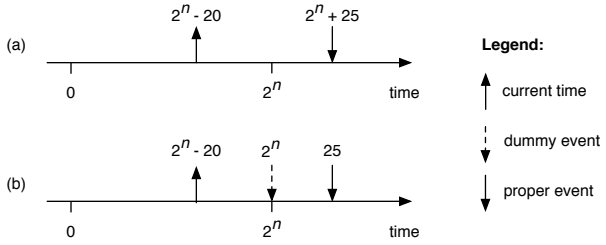

Fig. 4. Example of (a) overflowing absolute time of the first event (b) RELTEQ inserting a dummy event to handle the overflow.

B. Extending RELTEQ with hierarchical scheduling

The original description of RELTEQ [8] revolved around a periodic hardware timer driving a single event queue. To support hierarchical scheduling, we add an additional *server queue* for *each* server, to keep track of the events local to the server. At any time at most one server can be active; all other servers are inactive. The additional server queues make sure that the events local to inactive servers do not interfere with the currently active server.

In this new configuration the hardware timer drives two event queues:

1) the *system queue*, keeping track of events such as replenishment of periodic servers,
2) the *server queue* of the *active* server, keeping track of events such as task deadlines or the arrival of periodic tasks.

At every tick of the periodic timer the heads of both queues are decremented. The inactive server queues are left untouched.

When the active server is switched out (e.g. a higher priority server is resumed, or the active server gets depleted) then the active server queue is replaced by the queue belonging to the new active server. As a result, the queue of the switched out server will be "paused", and the queue of the switched in server will be "resumed".

To keep track of the time which has passed since the last server switch, we introduce one additional *stopwatch queue*. Initially it contains a single "dummy" event with time 0. At every tick of the periodic timer the head of the stopwatch queue is decremented. Time overflows are handled by setting the overflowing event to -2^n and inserting a new "dummy" event at the head of the queue with time equal to the overflow.

When the active server is switched out, the head event in the server queues of all inactive servers is decremented with the sum of all event times in the stopwatch queue, and the stop watch queue is reset to a single "dummy" event with time 0. Time overflows in the server queues are handled by inserting dummy events at the head of the queue, similar to handling overflows of the stopwatch queue.

When an inactive server is switched in, all leading events in its server queue are handled, until the head points to an event with a positive *absolute* event time. The absolute event times are computed in the same way as in the original RELTEQ,

by accumulating the relative times of subsequent events in the queue.

When the server budget is depleted an event must be triggered, to guarantee that a server does not exceed its budget. We could resolve the budget depletion events in a way similar to [11]. Because their approach requires to remove the budget depletion event from the server queue every time the server is switched out and to insert it back when the server is switched in, we opt for an alternative approach.

C. Extending RELTEQ with virtual timers

In Section I we have identified the need for "time-slot" events, which expire at times relative to the consumption of the server budget. In this section we present a general approach for handling both budget depletion and time-slot events and introduce the notion of *virtual timers*. Our approach avoids removing virtual events upon server switching and is therefore more efficient than that of [11].

We can implement virtual timers by adding a *virtual server queue* for *each* server. In this new configuration, at every tick of the periodic timer the heads of all four queues are decremented: system queue, active server queue, stopwatch queue, and active virtual server queue.

Similarly to the server queues introduced earlier, when a server is switched out, the active virtual server queue is paused and the switched in virtual server queue is resumed. The difference is that the stopwatch time is not subtracted from the head of the virtual server queue, since during the inactive period a server does not consume any of its budget.

An example of the proposed RELTEQ extension with hierarchical scheduling and virtual timers is shown in Figure 5.

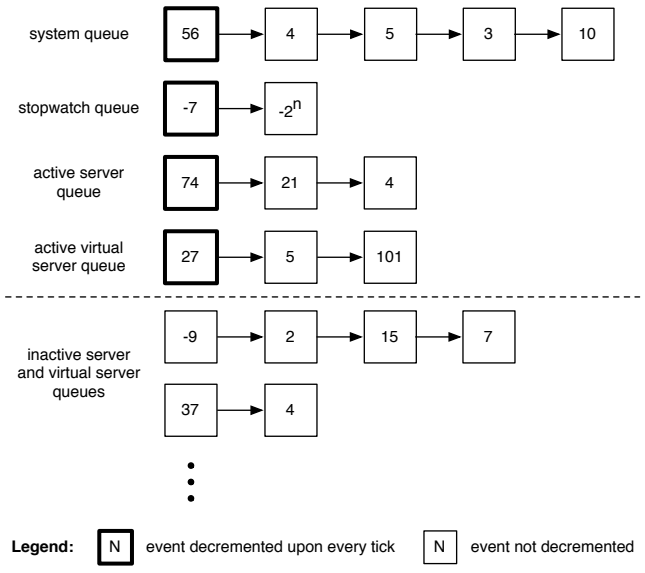

Fig. 5. Example of RELTEQ based implementation of reservations.

V. Conclusion

This paper generalizes the concept of a virtual timer to hierarchical real-time systems. Specifically, the paper proposes a technique to minimize the overhead of event handling in hierarchical scheduling frameworks that may be used in compositional design and analysis of complex real-time systems. In such systems, several applications execute on a shared processor where each application is given a virtual share of the processor and is responsible for local scheduling of tasks within itself. We outlined an implementation of hierarchical scheduling and virtual timers based on the RELTEQ approach to multiplexing timed events on a single hardware timer. The proposed implementation aims at minimizing the overhead of handling events belonging to inactive servers. In the future we would like to further investigate trade-offs between different design and implementation alternatives of HSFs with virtual timers in RELTEQ.

Our current research on providing temporal isolation between applications in real-time systems focusses on two-level HSFs. In the future work we would like to extend the proposed approach to multi-level hierarchical scheduling, where the hardware timer is driving a server queue for each server in the hierarchy of currently active servers.

References

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

[2] C. Hentschel and S. Schiemenz, "Priority-processing for optimized real-time performance with limited processing resources," in *Proc. 26th IEEE Int. Conference on Consumer Electronics (ICCE). Digest of Technical Papers.*, Jan. 2008.

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

[4] M. van den Heuvel, R. J. Bril, S. Schiemenz, and C. Hentschel, "Dynamic resource allocation for real-time priority processing applications," in *Accepted for 28th IEEE Int. Conference on Consumer Electronics (ICCE). Digest of Technical Papers.*, Jan. 2010.

[5] GNU-Project. (2009, Sep.) Setting an alarm - the gnu c library. [Online]. Available: http://www.gnu.org/s/libc/manual/html_node/Setting-an-Alarm.html

[6] S. Yoo, M. Park, and C. Yoo, "A step to support real-time in virtual machine," in *Proc. 6th IEEE Consumer Communications and Networking Conference (CCNC)*, Jan. 2009, pp. 1–7.

[7] D. Kim, Y.-H. Lee, and M. Younis, "Spirit-μkernel for strongly partitioned real-time systems," in *Proc. 7th Int. Conference on Real-Time Computing Systems and Applications.*, 2000, pp. 73–80.

[8] M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien, "Multiplexing real-time timed events," in *Proc. 14th IEEE Int. Conference on Emerging Technologies and Factory Automation (ETFA)*, July 2009.

[9] 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.

[10] S. Saewong and R. Rajkumar, "Hierarchical reservation support in resource kernels," 2001. [Online]. Available: http://www.cs.cmu.edu/afs/cs/project/rtml-2/Papers/hrsv.ps.gz

[11] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril, "Towards hierarchical scheduling on top of VxWorks," in *Proc. 4th Int. Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, July 2008, pp. 63–72.

[12] R. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *Proc. 26th IEEE Int. Real-Time Systems Symposium (RTSS)*, Dec. 2005, pp. 389–398.