CONTROL OF EVENT HANDLING TIMELINESS IN RTEMS

Manuel Coutinho
Instituto Superior Técnico - Universidade Técnica de Lisboa,
DEEC, Av. Rovisco Pais, 1049-001 Lisboa, Portugal,
e-mail: mabeco@comp.ist.utl.pt

José Rufino
Faculdade de Ciências da Universidade de Lisboa,
Campo Grande - Bloco C8, 1749-016 Lisboa, Portugal.
e-mail: ruf@di.fc.ul.pt

Carlos Almeida
Instituto Superior Técnico - Universidade Técnica de Lisboa,
DEEC, AC-Computadores, Av. Rovisco Pais, 1049-001 Lisboa, Portugal
e-mail: cra@comp.ist.utl.pt

ABSTRACT [1]
Embedded real-time applications that interact with the outside environment may be subjected to temporal uncertainty due to the potential asynchronous characteristics of events. If event handling, which is usually associated with interrupts, is not carefully controlled, overload scenarios can cause application tasks to miss deadlines, with severe consequences. In this paper we address the problem of controlling event handling timeliness, by enhancing the real-time multitasking kernel RTEMS with components to characterize event rate, decide if there is an overload situation, and switch between an interrupt mode and a polling mode event handling. This is done with minimal impact on the existing application, by replacing the interrupt handler by another one that implements those control mechanisms before calling the original application interrupt service routine. A case study using the keyboard as the input device is presented, and implementation issues are discussed.

KEY WORDS
Real-time embedded systems, Event handling, Interrupt rate, Temporal protection, RTEMS.

1 Introduction

Embedded control applications consist in general of several tasks that need to be executed in a concurrent fashion. Usually, these tasks (at least some of them) have real-time requirements, being of utmost importance to ensure timeliness properties. Most of these control applications also need to interact with the real-world, performing input/output operations through a set of devices such as sensors and actuators. Due to the basic requirements of these applications (concurrent tasks, real-time, input/output event handling), multitasking real-time kernels are a fundamental component in their development. They provide an adequate infrastructure to help embedded systems programmers in the implementation of such applications.

However, when there is interaction with the external environment, care must be taken if we want to preserve timeliness guarantees. If external events are not controlled, they may cause uncertainty in the time domain due to potential overload situations. This happens when the event rate associated with the interaction to the external world is not bounded (or at least can be higher than the assumed value at design time).

In order to solve this problem we need to address the main methods to deal with input/output event handling and incorporate timeliness control mechanisms. Event filtering, interrupt rate control, switching between interrupt and polling modes and tuning of polling cycle times, are examples of actions to take to achieve the desired goals. Ideally, these timeliness control mechanisms should be integrated in the system as smoothly as possible, and in a generic way so as to be usable for several different types of events.

In this paper, we present work done to solve this problem in the context of the real-time multitasking kernel RTEMS, but it can be applied to other real-time kernels.

The paper is organized as follows: in the next section we give a brief overview of RTEMS; in Section 3 considerations about input/output event handling and the use of interrupts are made; in Section 4 timeliness control mechanisms are explained and solutions are introduced; in Section 5 the results of the incorporation of those timeliness control mechanisms into the RTEMS real-time kernel are presented. In Section 6 we present some related work, and the paper ends with the conclusions.

2 The Multitasking Kernel RTEMS

RTEMS (Real-Time Executive for Multiprocessor Systems) is a well-known, real-time multitasking kernel, with a modular architecture, offering interesting characteristics to support the development of real-time embedded applications [1]. It was designed to support applications with real-time requirements while being compatible with open standards. It is an open source operating system, that is currently maintained by OAR (On-Line Applications Research Corporation). It is available for several different platforms/architectures, including the PC386 [2], which is the one we use in this work. It provides a high performance environment including the following features:

- multitasking capabilities
- homogeneous and heterogeneous multiprocessor systems
- event-driven, priority-based, preemptive scheduling

[1] This work was partially supported by FCT, through Project POSC/EIA/56041/2004 (DARIO).

- optional rate monotonic scheduling
- intertask communication and synchronization
- priority inheritance
- responsive interrupt management
- dynamic memory allocation
- high level of user configurability

From the point-of-view of its internal architecture RTEMS can be viewed as a set of layered components that provide a set of services to a real-time application system. The executive interface presented to the application is formed by grouping directives into logical sets called resource managers (Figure 1).

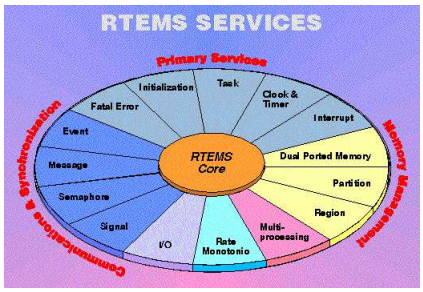

Figure 1. RTEMS resource managers ([1])

A more detailed description of each RTEMS resource manager can be found in [1], but in a generic way, the set of available RTEMS managers provide the ability to have real-time multitasking applications where it is possible to have communication and synchronization between cooperating tasks and between tasks and interrupt service routines (ISRs). Functions utilized by multiple managers such as scheduling, dispatching, and object management are provided in the executive core, which depends on a small set of CPU dependent routines. As embedded systems resources are usually scarce, RTEMS was specifically designed to allow unused managers to be excluded from the run-time environment. This way, an application does not have to pay (in memory size) for something that it does not need.

All these aspects make RTEMS a suitable operating system for the development of most real-time applications. However, as we explain in this paper, there are situations where additional control is needed when interacting with the external world. This is particular important when the application has strict timeliness requirements and the overall system may be subjected to event overload. In the next sections we will present solutions to deal with this problem and increase RTEMS robustness.

3 Event Handling - Interrupts

External events are detected either by polling the specific device or having the device generate an interrupt. While polling the device provides a more controlled way to deal with the events, it can be very inefficient because there may be nothing to read or, if the polling period is too high, events may be lost. An event-triggered approach, through the use of interrupts, provides a more flexible architecture. In scenarios where the occurrence of events is not uniformly distributed in the time domain, or where there are dynamic aspects in the overall system, an interrupt approach is usually more effective.

In RTEMS, it is the interrupt manager that allows the application to specify an interrupt service routine (ISR) by connecting a function to a hardware interrupt vector. The RTEMS directive to establish an ISR (rtems_interrupt_catch) accepts the address of the user's ISR and its associated CPU vector number. It installs the RTEMS interrupt wrapper in the processor's Interrupt Vector Table and the address of the user's ISR in the RTEMS' Vector Table. This directive also returns the previous contents of the specified vector in the RTEMS' Vector Table. When an interrupt occurs, the processor will automatically vector to RTEMS, which saves and restores all registers that are not preserved by the normal C calling convention for the target processor and invokes the user's ISR. The user's ISR is responsible for processing the interrupt, clearing the interrupt if necessary, and for device specific manipulation. It is called with the vector number as argument which allows the application to identify the interrupt source and thus being able to use the same routine to service different interrupts, if desired. Once the user's ISR has completed, it returns control to the RTEMS interrupt manager which will perform task dispatching and restore the registers saved before the ISR was invoked. Using the interrupt manager insures that RTEMS knows when a directive is being called from an ISR. The ISR may then use some system calls to synchronize itself with an application task. The synchronization may involve messages, events or signals being passed by the ISR to the desired task.

Although flexible, interrupts may however introduce uncontrolled situations. In most multitasking kernels, RTEMS included, interrupts have higher priority than any task running on the processor. If an interrupt overload situation takes place, the application timeliness may be in jeopardy. In order to have the flexibility of interrupts, but at the same time being able to preserve application timeliness properties, we need timeliness control mechanisms in event handling to filter potential interrupt overloading. These overload situations may occur due to unanticipated load, or due to faulty scenarios. In order to support dependable real-time applications, those control mechanisms are of utmost importance.

4 Control of Event Handling Timeliness

Asynchronous event handling, based on interrupts, introduces temporal uncertainty that may interfere with application timeliness. The rate at which interrupts are generated, if not bounded, may affect task execution time and, in a worst-case scenario, deadlines may be missed. To solve this problem, one must ensure that a given maximum al-

lowed event rate is not overtaken. There must be a *filter* to enforce that behavior (Figure 2).

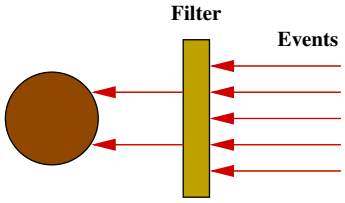

Figure 2. Applying a filter to control event rate

This control mechanism must intercept interrupts, determine the interrupt rate to evaluate if there is an overload situation, and, if necessary, disable interrupts and switch temporarily to a polling mode.

4.1 Event Characterization

In order to implement these timeliness control mechanisms, we need to have the means to characterize event occurrence, thus allowing to determine its rate and decide if there is, or not, an overload situation (Figure 3). Just measuring the time interval between any two interrupts may not be enough. Although it is useful in specific situations to detect a fault scenario if a pre-defined maximum inter-arrival time is exceeded, in a more generic situation it may be possible (from the point-of-view of the application) to have several interrupts in a given time interval, being that load tolerated. We would like to have more information about the event occurrence before making a decision.

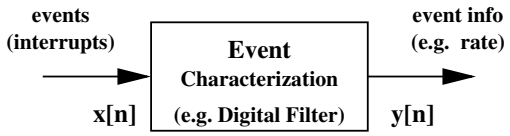

Figure 3. Module for event characterization

As interrupts are discrete events, in the context of this work, we decided to exploit discrete signal processing knowledge [3], with adaptations, to deal with this problem. The idea is to have a digital filter, that is applied to events and provides information about them. For now, we are mainly interested in obtaining the event rate, but in a more generic situation, depending on specific filter parameterization, it will be possible to obtain different type of information about the events.

Discrete Event Processing

In order to apply the discrete event processing, we need to discretize time [4, 3]. Although we may consider the computing system internal clock as discrete (resolution $\sim 1\mu s$), in a macroscopic scale, from the point-of-view of event generation intervals, it can be considered as continuous. Discretization will be done through *sample/hold*. Assuming a given sampling period (T_{sample}), system state is periodically checked (*sample*) and kept (*hold*) until the next sampling. Continuous time t is thus transformed to a discrete time n through $t = nT_{sample}$. To represent that an event has occurred at discrete time n, a scalar discrete function $x[n]$ $(x : Z \rightarrow 0, 1)$ is used:

$$x[n] = \begin{cases} 1, & \text{if an event occurs in } n \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

Consider another discrete function $z[n]$ which represents the number of detected events until n

$$z[n] = \sum_{i=1}^{n} x[n] \quad (2)$$

assuming that no events occurred for $n < 1$ (that is, $x[n] = 0, \forall n < 1$). This is, in the discrete signal literature[3], described as *running sum* or *"discrete integral"*. It simply adds the number of events that occurred in the past plus if an event is occurring now, at discrete time n. In a recursive form, Equation 2 can be represented as

$$z[n] = z[n-1] + x[n] \quad (3)$$

The event rate can therefore be described as another discrete function $y[n]$ given by

$$y[n] = \frac{z[n]}{n} \quad (4)$$

which gives the average of the number of events occurred until the time n. For a recursive form of $y[n]$, the following equation can be derived from equations 3 and 4

$$y[n] = \frac{z[n-1] + x[n]}{n} = \left(1 - \frac{1}{n}\right) y[n-1] + \frac{1}{n} x[n] \quad (5)$$

Although the average number of events may be important from the point-of-view of event characterization, for our current purpose (overload detection), we would like $y[n]$ to give, not the average number of events since the origin of time, but information about a more instantaneous event rate generation. Otherwise, if the system has been running for a long time (large n), it will be very slow to react to an event overload. This happens because all events are assumed to have the same importance regardless of having occurred recently or a long time ago.

The most obvious solution to this problem consists in considering only the last events inside a temporal window (D). In the literature this is known as a FIR (Finite Impulse Response) filter, where $z[n]$ would be given by

$$z[n] = \sum_{i=n-D+1}^{n} x[n] \quad (6)$$

The parameter D is known as the FIR dimension and represents the number of samples of $x[n]$ to be considered in the past. The event rate is now determined by the average of the last D samples (note that this filter is equal to the first one presented if $D = n$).

$$y[n] = \frac{z[n]}{D} = \frac{1}{D} \sum_{i=n-D+1}^{n} x[n] \quad (7)$$

Another solution to have a more responsive system consists in weighting events based on the time of its occurrence. For that, we can use an IIR (Infinite Impulse Response) filter (recursive filter) and weight the events based on the event occurrence time. The event rate is given by

$$y[n] = \alpha y[n-1] + (1-\alpha)x[n] \quad (8)$$

where the parameter α belongs to the interval $]0;1[$ in order to have a stable system and $y[n]$ positive. Note that this filter can also be converted to the first one if we make $\alpha = (1 - 1/n)$ (see Equation 5).

Using our methodology it is possible to implement specific filters to handle special cases such as a *maximum burst size*, for example. It is also possible, if desired for performance, to implement these filters using hardware.

4.2 Overload Detection

Having a modular component able to characterize events, it is now possible to use it to detect if/when there is an overload situation. The way events are registered/applied to that component could be done using a specialized task that periodically samples system state. However, that method is inefficient, implying to run the task even when there are no events and, if the polling period is too high, may also miss events. Moreover, if an overload situation is in progress, the polling task may be overrun by interrupts and so the goal for which it was designed is not achieved.

As we are mainly interested in overload situations caused by interrupts, detecting an overload situation inside the Interrupt Service Routine (ISR) is a preferred method. All events (interrupts) are accounted for, and there is only associated processing when a new event is generated. It also allows for a faster response to an overload situation by being able to immediately disable the interrupt. The downside of this method is the time overhead required inside an ISR to calculate the event rate, and possible restrictions about the use of floating-point operations. However, it is possible to make some optimizations, as will be explained later. The fact that the system is not sampled with a periodic rate (assumed in the discrete event processing literature [4, 3]), is easily overcome by registering the time of event occurrence and transform it to discrete time as explained before ($t = nT_{sample}$). When a new event occurs we know the elapsed time since the last registered event and thus are able to *"reconstruct"* the *sampling data* because we know that there were no events in the meantime.

4.3 Overload Handling

When an overload is detected and the corresponding interrupt is disabled, two methods can be used to allow a graceful degradation of the services while maintaining a bounded interference with the rest of the system. By allowing interrupts in certain time intervals, the system can limit the interference and admit some interrupts to be processed. Adapting the time interval in which interrupts are enabled, the system can cope with transient situations and decide whether the overload has passed. A timer can be used to inform when to re-enable the interrupt.

A second solution consists in having a special periodic task that polls the device, checking for events and determining if an overload is still present. If not, the system returns to the normal state by enabling again the interrupt. The period and priority of this task must be such that it does not interfere with the deadlines of the rest of the system. The component to determine the current event rate to decide if there is still an overload situation, or not, can be the same as before. However, to provide a more stable environment, a hysteresis cycle can be used with the definition of two different thresholds M and m, associated with mode switching. The value of the low threshold (m) is even much lower than M because when in polling mode there are events that might be missed (due to a larger sampling period). When controlling the overload of more than one interrupt source, the same task can be used acting as a cyclic executive calling the processing routines associated with each event.

4.4 Implementation in RTEMS

The mechanisms described above can be implemented in RTEMS with minimal impact on the existing infrastructure. The original Interrupt Service Routine (ISR) associated with the event is replaced by another one that, using the previous described components, determines the interrupt rate, decides if there is an overload, disabling the interrupt and switching to a polling mode if necessary, and calls the original ISR to process the event (see Figure 4). Note that the RTEMS directive to specify an ISR returns the address of the existing ISR. The pseudo-code of the new ISR, implementing an IIR filter to determine the interrupt rate, is described in Figure 5.

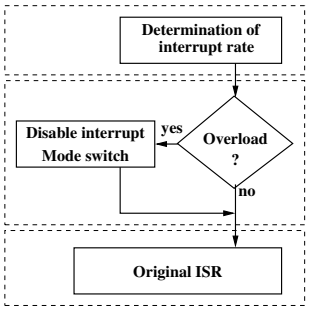

Figure 4. Flowchart of the new Interrupt Service Routine

As explained before, due to the fact that events are registered in an asynchronous way (when an interrupt occurs) instead of periodically (as assumed in the discrete signal processing literature), some adaptations must be done. Continuous time must be converted to discrete time taking into account the sampling period. As system state (event occurrence) is only registered when events do oc-

```
newISR(){
                        // event rate determination
    t = getTime();
    n = sampleHold(t);
    y[n+1] =α^(n−last_n) y[n] + (1-α);
    last_n = n;
                        // overload detection and mode switching
    if( y[n+1] > M ){
        state = overload;
        disableInterrupt();
        switchPollMode();
    }
                        // event processing
    originalISR();
}
```

Figure 5. Pseudo-code of ISR using IIR filter

cur, we must also account the event *non-occurrence* during the elapsed time. As $x[n] = 0$ when an event does not occur in n, $y[n]$ is given by only the first term of Equation 8. This way, during an interval N without events, $y[n]$ can be obtained by Equation 9.

$$y[n] = \alpha y[n-1] = \alpha^2 y[n-2] = ... = \alpha^N y[n-N] \quad (9)$$

As the term α^{n-last_n} is too computationally expensive to be determined at runtime in the ISR, a table is used (built during system initialization) to perform this calculation with only one memory access. The exponent, which is an integer, is used for indexing the correct table position. If the exponent is higher than the table length, a value of zero is used instead (recall that $\alpha < 1$).

The time granularity offered by RTEMS is a clock tick, which is user defined but typically around $10ms$. This resolution is not suitable for this kind of filters due to the sampling time being of the same order as the minimum inter-arrival time of the interrupts, which is around $10 - 100\mu s$. A new function was built that reads from the internal count of `Timer0` (in the Intel-386 architecture) offering a time resolution of $1\mu s$.

5 Results

To demonstrate the ability of the proposed control mechanisms to cope with overload scenarios, a simple test involving a rapid keyboard user and a deliberate *stuck key* is presented. Although the keyboard is not a very fast device (it can generate 33 interrupts per second), it can, nevertheless, be used as a representative input device. To determine the interrupt rate, we used an IIR filter with the following parameters: $T_{sample} = 1ms; M = 0,02; m = 0,002; \alpha = 0,999; PollingPeriod = 300ms$ (polling task).

Figure 6 represents the situation where no protection mechanism is implemented. In the first 4000 samples, a user is pressing keys normally and $y[n]$ tends to stabilize to a relatively low value. In between $n = 7000$ and $n = 11000$ a *stuck key* is experimentally simulated, resulting in the rise of $y[n]$. One can see that the user is not fast enough to trigger overload detection, whereas the rhythm

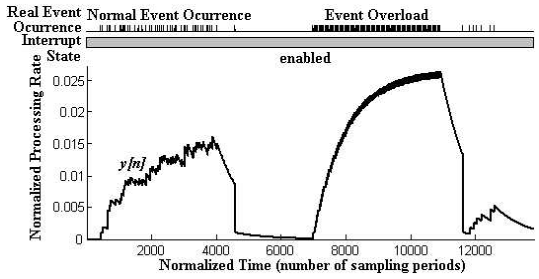

Figure 6. Event rate (IIR filter, no protection mechanism)

of the stuck key is more than enough. At approximately $n = 4500$ and $n = 11000$, $y[n]$ decreases rapidly due to the way the function α^{n-last_n} is implemented in the ISR. The table used has a dimension of 200 sampling periods, meaning that if two interrupts are separated by more than this time interval, $y[n]$ is set to $(1 - \alpha) = 0,001$. Besides reducing the size of the table, this is also a good mechanism to rapidly decrease $y[n]$ when the user stops pressing keys.

In Figure 7 the same scenario is represented, but now with the protection mechanism activated. When $y[n]$ goes above M, the system detects an overload, disabling keyboard interrupts and activating the polling task. During the overload, the system reads the keyboard and because there were pressed keys, the system stabilizes with approximately $y[n] = 1/300 = 0,0033$. When the overload is over ($y[n] < m$), the system enables the interrupts again and returns to normal mode.

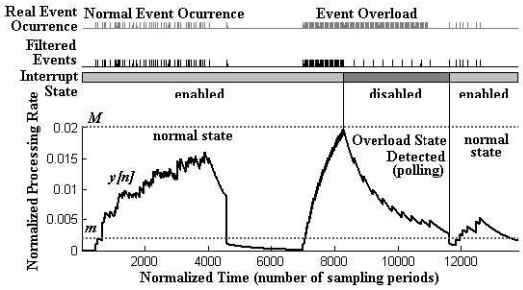

Figure 7. IIR Filter and protection mechanism

6 Related Work

Ensuring timeliness properties in real-time applications is very important. When applications interact with the external world this problem is harder to solve. Traditional work related to real-time addresses mainly scheduling algorithms such as RMS (Rate Monotonic Scheduling) and EDF (Earliest Deadline First) [5, 6], that are concerned with meeting the deadlines of processes without having much interference from input/output events. Other works reserve specific percentages of CPU to critical activities thus providing a temporal protection to those activities in more dynamic

environments where overload scenarios are not ruled-out [7]. However, when there is also event overload, event handling must be addressed as well.

I/O throttling is a technique that can be used to control the rate of input/output operations. In [8] this technique is used together with other techniques to control resource usage. They use a sliding window of recent events to compute the average rate for a target resource. The assigned limit is enforced by the simple expedient of putting application processes to sleep when they issue requests that would bring their resource utilization out of the allowable profile. However, dealing with external events implies a different approach to enforce temporal protection. Events must be filtered at the interface between the system and the environment. From the point-of-view of event handling there are mainly two alternatives: the event-triggered approach and the time-triggered approach [9, 10]. As their names suggest, in the first case the system reacts to events, whereas in the second case it is the elapsing of time that regulates system behavior. Although the time-triggered approach makes it easier to reason about system properties in the time domain [11], it is not always realistic to assume this type of interaction with the real-world. An event-triggered approach is more flexible and, with the right control mechanisms, can be used most of the time.

The work presented in [12], is a recent work, developed in parallel with our work, that has a similar goal: to prevent interrupt overload. However, they are mainly concerned in bounding the interrupts, whereas we are also interested in a more complete characterization of event occurrence before deciding about the existence of an event overload. In [13] there is an effort to integrate task scheduling and interrupt scheduling. Although an important issue, it is not always possible to have this type of approach, due to specific system limitations.

7 Conclusions

In this paper we addressed the problem of controlling the timeliness associated with event handling in overload scenarios. This is a very important issue for embedded real-time applications that need to interact with the external environment. These applications are usually concurrent and have tasks with strict deadlines. Interrupts associated with event handling, if not carefully controlled, may introduce system overload and jeopardize those real-time deadlines.

In this work, we explained how to incorporate control mechanisms to provide temporal protection in the context of the multitasking kernel RTEMS. Following a structured and modular approach, event occurrence is characterized, enabling to make a decision about the existence, or not, of an overload situation. When an overload is detected, the interrupt associated with the event is disabled, and event handling is done using a polling mode, thus enforcing the desired maximum allowed rate. When the event rate returns to a normal situation, the original interrupt mode is restored. This control mechanism is done with minimal impact on the existing application by replacing the original interrupt service routine (ISR) by another one that performs the desired control before calling the original ISR.

The case study presented uses a keyboard, which, although a slow device, illustrates the achievements that are possible to obtain. The same type of control mechanisms can be applied to other input devices that have a smaller time granularity.

References

[1] On-Line Applications Research Corporation (OAR). *RTEMS C User's Guide*, edition 4.6.2, for rtems 4.6.2 edition, August 2003. (The RTEMS Project is hosted at http://www.rtems.com.).

[2] On-Line Applications Research Corporation (OAR). *RTEMS Intel i386 Applications Supplement*, for rtems 4.6.2 edition, August 2003.

[3] S. Smith. *The Scientist and Engineers Guide to Digital Signal Processing*. California Technical Publishing, San Diego, California, 2nd ed., 1999. Analog Devices Tech. Library.

[4] A. V. Oppenheim, R. W. Schafer, and J. R. Buck. *Discrete Time Signal Processing*. Prentice-Hall Int., 2nd ed., 1999.

[5] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 20(1):46–61, 1973.

[6] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proc. IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[7] C. W. Mercer, R. Rajkumar, and J. Zelenka. Temporal protection in real-time operating systems. In *Proc. of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, May 1994.

[8] Kyung D. Ryu, Jeffrey K. Hollingsworth, and Peter J. Keleher. Efficient network and i/o throttling for fine-grain cycle stealing. In *Proc. of the 2001 ACM/IEEE conference on Supercomputing*, Denver, Colorado, USA, 2001. ACM Press.

[9] P. Veríssimo and H. Kopetz. Design of real-time systems. In S.J. Mullender, editor, *Distributed Systems, 2nd Ed.*, ACM-Press, ch. 19, pp 491–536. Addison-Wesley, 1993.

[10] H. Kopetz. Event-triggered versus time-triggered real-time systems. In *Proc. of the International Workshop on Operating Systems of the 90s and Beyond*, volume 563 of *LNCS*, pages 87–101. Springer-Verlag London, UK, 1991.

[11] H. Kopetz and B. Gunther. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, January 2003.

[12] J. Regehr and U. Duongsaa. Preventing interrupt overload. In *Proc. of the ACM Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2005)*, Chicago, IL, June 2005.

[13] L. E. Leyva del Foyo and P. Mejia-Alvarez. Custom interrupt management for real-time and embedded system kernels. In *Embedded and Real-Time Systems Implementation (ERTSI 2004) Workshop*, December 2004.