RTOS Scheduler Implementation in Hardware and Software for Real Time Applications

Melissa Vetromille, Luciano Ost, César A. M. Marcon, Carlos Reif, Fabiano Hessel

PPGCC - FACIN – PUCRS - Av. Ipiranga, 6681, Porto Alegre, RS – Brazil

{mvetromille, ost, marcon, reif, hessel}@inf.pucrs.br

Abstract

In order to enhance performance and improve predictability of the real time systems, implementing some critical operating system functionalities, like time management and task scheduling, in software and others in hardware is an interesting approach. Scheduling decision for real-time embedded software applications is an important problem in real-time operating system (RTOS) and has a great impact on system performance. In this paper, we evaluate the pros and cons of migrating RTOS scheduler implementation from software to hardware. We investigate three different RTOS scheduler implementation approaches: (i) implemented in software running in the same processor of the application tasks, (ii) implemented in software running in a co-processor, and (iii) implemented in hardware, while application tasks are running on a processor. We demonstrate the effectiveness of each approach by simulating and analyzing a set of benchmarks representing different embedded application classes.

1   Introduction

The development of real-time embedded systems is continuously increasing. Real-time applications, which require fast response and high synchronization, are becoming even more popular. The operating system is without hesitation the most important software of all system programs in a real-time embedded system. Hence, a Real-Time Operating System (RTOS), which handles both soft and hard real-time tasks, is extremely necessary to the effectiveness of those designs.

As the system becomes larger, the scheduling of tasks and communications becomes more complex and its impact on the entire system performance becomes more significant [1]. Furthermore, real-time demands inject an additional correctness criterion into embedded systems. It is not just the result that is important, timing issues also have to be considered. Moving RTOS scheduler functionalities from software to hardware can enhance performance of RTOS systems. However, this approach can increase design complexity and enlarge silicon area occupation.

This work investigates and discusses the pros and cons of three different scheduler implementations: software, software-software, and hardware/software. A software implementation considers a processor running the scheduler and the application tasks. A software-software implementation considers a processor running the application tasks, and a co-processor running the scheduler. In a hardware-software implementation, the scheduler is implemented directly in hardware, and a processor running the application tasks.

Figure 1 and Figure 2 illustrate a comparison of a scheduler implemented in the same processor where application tasks are running, with the one implemented in a different processing element (hardware or software).
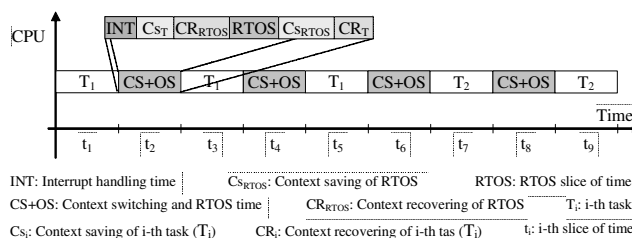

INT: Interrupt handling time CS_{RTOS}: Context saving of RTOS   RTOS: RTOS slice of time
CS+OS: Context switching and RTOS time CR_{RTOS}: Context recovering of RTOS T_i: i-th task
CS_i: Context saving of i-th task (T_i) CR_i: Context recovering of i-th tas (T_i) t_i: i-th slice of time

Figure 1: Example of a RTOS scheduler implemented in software running on a single processor.

In Figure 1, the system is entirely implemented in software running on a single processor. As a result, every determined time slice (t_2, t_4, t_6 and t_8) the processor is interrupted to enable a new task scheduling. When the interrupt occurs, if the scheduler is implemented as a process [11][12], the processor takes time dealing with the interrupt routine, performs the RTOS functionalities as well as four context switches: (*i*) execution task information saving; (*ii*) RTOS execution status recovering; (*iii*) RTOS execution status information saving; and, (*iv*) next task information recovering. It occurs; even if the previous task would be elected to continue running, as it is illustrated in time slices t_2, t_4 and t_8 of Figure 1. Here, it is obvious that the processor wastes time with OS tasks scheduling and unnecessary context switches.

In Figure 2, the RTOS scheduler is being executed in parallel with the application tasks processing, enabling to interrupt the task processing only if a priority task has to be

executed. With this implementation the processor does not waste time with unnecessary context switches, increasing the overall system performance.
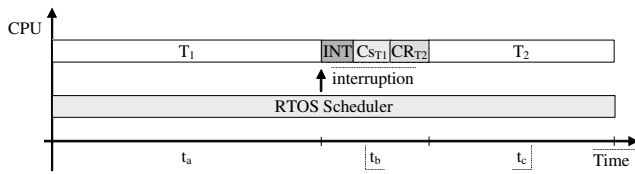

Figure 2: Example of a RTOS scheduler running on a co-processor or specific hardware.

Let *RTOS interrupt interval* be the interval of time between to instant that the processor is interrupted to perform RTOS scheduling. Therefore, when the scheduler is totally implemented in software, we must consider the time necessary to fulfill deadlines, and this time varies from zero to RTOS interrupt interval.

Normally, RTOS interrupt intervals are based on a multiple of clock system frequency, and in general, not all application tasks deadline are multiple of this value, implying variations in the deadline fulfilling. These variations characterize the *jitter problem* (Figure 3). For some application, the jitter problem can damage the real-time operation, since real-time applications depend not only on the results achieved, but also on the instants that the results are achieved.
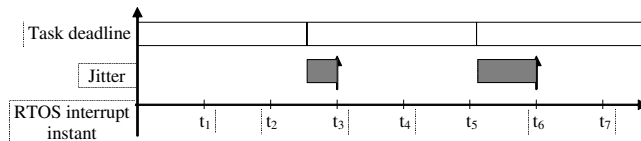

Figure 3: Jitter problem.

The remaining of this paper is organized as follows. The related work is presented in Section 2. Section 3 provides an overview of three scheduler models implementation.

Section 4 shows a case study and Section 5 presents our conclusions, and future work.

2 Related work

Few researches address hardware/software RTOS implementation. Table 1 shows a comparison among them.

Mooney [8] proposed a framework to generate a partitioned hardware/software RTOS. Independent of tasks requirements, this approach generates only one OS that is replicated on every processor. The designer does not have the flexibility to choose which components are implemented in software or hardware. Additionally, the designer cannot control the task mapping onto the target processors.

Nakano [9] implemented a partitioned OS, called STRON (Silicon TRON). Nevertheless, the system does not allow choosing which components are going to be implemented in hardware and which ones are going to be developed in software.

Ortiz [3] described the implementation of the scheduler and the processes control queues directly in hardware. Such as the others approaches presented, this one implements just a few predetermined components in hardware.

In order to investigate area overhead and performance, Cho [1] proposed the implementation of centralized and distributed schedulers in a multiprocessor SoC. This approach considers only static scheduler implementation.

Samuelsson [10] presented a performance comparison between a real-time kernel implemented in hardware and an equivalent one implemented in software. They used a hardware multiprocessor platform, called SARA. The hardware kernel implements the scheduler, inter-process communication methods, semaphores and timer.

Table 1: Comparison among related work.

	Kernel	Functions	Comparison	Result
Mooney	Atalanta	Deadlock control unit, block cache and memory management	Performance (kernel software x partitioned kernel)	Better performance in hardware
Nakano	µITRON	Event flags, task queues, module control, scheduler and timer	Performance (kernel software x partitioned kernel)	Better performance in hardware
Ortiz	KURT-Linux	Scheduler, event queues, interrupt handling	Performance (kernel software x partitioned kernel)	Better performance in hardware just for tasks executing in WCET
Cho	-	Scheduler	Performance and area (centralized scheduler x distributed scheduler)	Distributed scheduler occupies greater area, but presents better performance
Samuelsson	Kernel model	Scheduler, IPC methods, semaphores and timer	Performance (kernel software x kernel hardware)	Better performance in hardware
Vetromille	Kernel model	Scheduler and task queues	Performance (kernel software x partitioned kernel – software/software and software/hardware)	Better performance in hardware or software, depending on the application class

Applications can be classified according to some relevant characteristics, like communication and computation requirements. Applications of a given class have similar behavior front of a given stimuli, requiring similar mechanisms to work properly. For instance, hard real time applications need to operate with hard time constraints, implying in using hard scheduling policy.

Different from the others, this work analyzes what scheduler implementation is more suitable for a given application class: software using a single processor, software partitioned using two processors and hardware/software partitioned using a processor and a dedicated hardware block. These topics are better discussed in Sections 4 and 5.

3   Scheduler models implementation for real-time applications

This Section provides an overview of three scheduler models implementation: (*i*) SoRTS (Software Real-Time Scheduler), (*ii*) Co-SoRTS (Co-processor Software Real-Time Scheduler), and (*iii*) HaRTS (Hardware Real-Time Scheduler). The architecture and the execution of the scheduler policy for each approach are discussed next.

We implemented these schedulers on a Xilinx Virtex-II Pro XC2VP30 FPGA, using Xilinx Embedded Development Kit (EDK) and Modelsim.

In order to validate the software scheduler implementations, we used MicroBlaze processor available in the EDK environment. The MicroBlaze is a 32-bit Harvard RISC architecture and its operating frequency was determined to be 50 MHz, for prototyping purposes.

3.1   SoRTS

The SoRTS architecture (Figure 4) consists of six components: (i) MicroBlaze processor, (ii) Block RAM memory, (iii) OPB (On-chip Peripheral Bus), (iv) communication interface, (v) interrupt and time control, and (vi) UART.
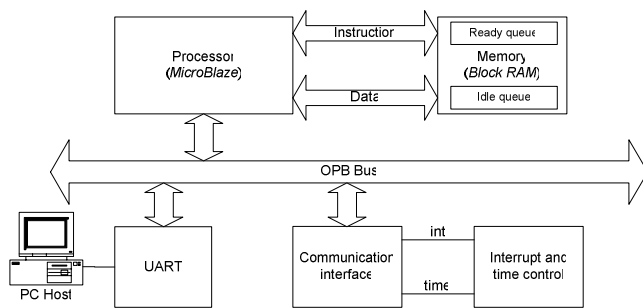
Figure 4: SoRTS block diagram architecture.

The MicroBlaze executes application tasks, which are characterized by: (i) period, (ii) deadlines, (iii) task ID, (iv) execution time. The Block RAM stores two structures: *Ready queue* and *Idle queue*. The *ready queue* contains an ordered list of tasks that can be executed according to their priorities, which is determined by the scheduling policy. The *idle queue* has a list of executed tasks that are waiting for a new time slice to execute. The communication among architecture components is performed by a 32-bit OPB. The *communication interface* is a specific model instanced by EDK that allows the communication between software (RTOS and application tasks running in the MicroBlaze) and proprietary hardware (interrupt and time control) via OPB. This communication is based on two registers: (i) *time* – which returns the system time, and (ii) the *int* – which is used to send an interruption to the MicroBlaze. These registers are accessed through functions available in EDK tool. Finally, the *UART* provides communication between the Xilinx development board and the host computer (EDK development kit), which has been used to validate our experiments.

3.2   Co-SoRTS

Co-SoRTS increases SoRTS architecture with an additional MicroBlaze. The first MicroBlaze processor executes a set of tasks stored in the Block RAM. The second MicroBlaze is used as a co-processor for RTOS scheduler implementation, as it is illustrated in Figure 5.
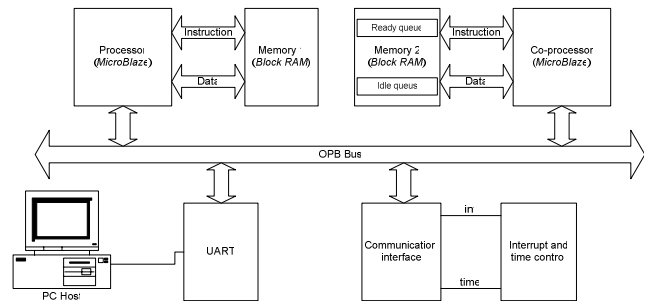
Figure 5: Co-SoRTS block diagram architecture.

This approach eliminates the incidence of non-necessary context switches and reduces the jitter problem. Here, the context switch only occurs if a new task is scheduled. At this moment, the co-processor send an interrupt signal to the processor to performs a context switch.

In some application class, the decrease of context switches is a potential advantage if compared to a scheduler running in a single processor. Three new internal registers had been used to attend the MicroBlazes intercommunication requirements. The remaining system components are responsible for supplying the same functionalities adopted and described in the Section 3.1.

IEEE
COMPUTER
SOCIETY

3.3 HaRTS

Similar to the Co-SoRTS, the HaRTS architecture uses a dedicated hardware component for scheduling tasks and list management, as shown in Figure 6.

The dedicated hardware (Figure 7) has four main modules: (i) scheduler module, (ii) queue control, (iii) communication interface, (iv) time control.

The *scheduler module* is composed by three blocks: (i) fail process; (ii) running process; and (iii) ready process. The running and ready process are responsible for task scheduling, according to parameters previously stored in the *queue control*. In order to reduce the area cost the queue control implements only one list for task management. This list is accessed to find out the current task state (fail, running or ready), in order to perform the scheduling policy. No task is removed from de list; just its states are updated. The *fail process* verifies the occurrences of task fails and signalizes the scheduler module.
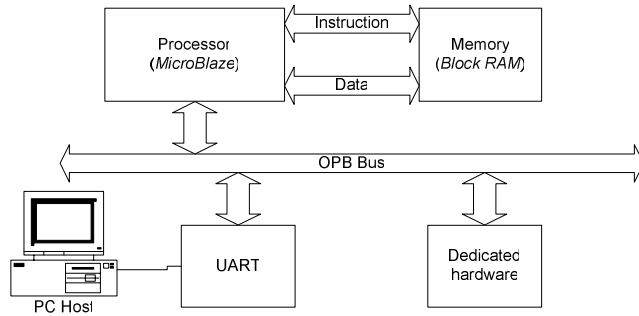

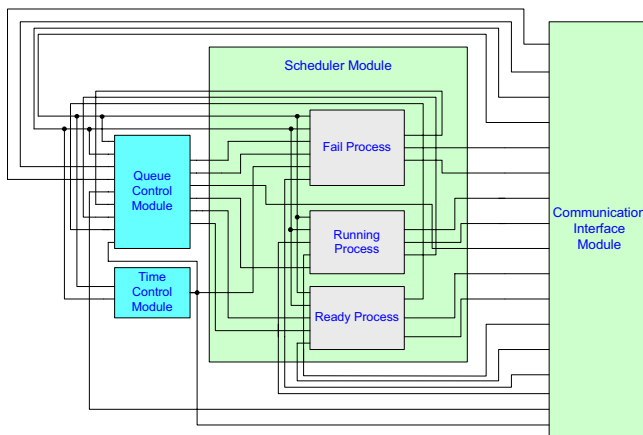**Figure 6: HaRTS block diagram architecture.**

Figure 7: Dedicated hardware architecture.

The HaRTS *communication interface* is much more complex than the one implemented in SoRTS and Co-SoRTS, implying the usage of sixteen internal registers due to native MicroBlaze communication protocol.

Finally, the *time control* is responsible for the time system management.

4 Case study

This Section presents a case study composed by a set of synthetic benchmarks, representing different embedded application classes. We are interested in compare the number of deadline fails, the number of context switches and the CPU occupation time dedicated to tasks execution. It allows verifying what scheduling implementation approach (SoRTS, Co-SoRTS, and HaRTS) is better suited to execute a specific application class. Each benchmark is composed by a set of tasks modeled by its period, deadline and average case execution time. We vary the context switches time of each benchmark in the range of 25, 50, 75 and 100 us. For each context switch time we applied three different values for RTOS interrupt interval (250, 400 and 500 us). Results were achieved by 10 seconds of execution of each benchmark. In all benchmarks we used RM (Rate Monotonic) as scheduling policy.

4.1 Context switch

Figure 8 illustrates the number of context switches (vertical axis) and the context switches execution time (horizontal axis) after 10 seconds of system execution, considering SoRTS implementations.
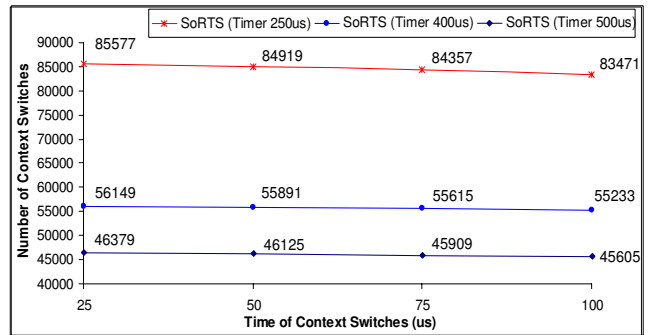

Figure 8: Comparison of number of context switches between different SoRTS implementation.

Figure 8 show that when the time of the context switches increases, the number of context switches reduces. This reduction happens due to the increase of tasks fails (Figure 10). Additionally, we can observe that the number of context switches decreases with the increase of the RTOS interrupt interval (250, 400 and 500 us). Obviously, it happens due to the increase of task interrupt frequency. However, this augment implies in the jitter increase, consequently some real-time tasks may not have the correct result in the correct time.

Analyzing HaRTS and Co-SoRTS scheduler implementations, we conclude that HaRTS presents less number of context switches than Co-SoRTS (Figure 9). It happens since the communication protocol used in Co-SoRTS is more complex because of the intrinsic processor communication interface, which generates larger communication overhead.
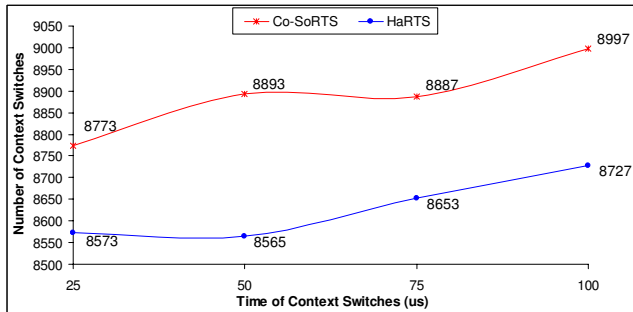

Figure 9: Comparison between Co-SoRTS and HaRTS.

Comparing Figure 8 and Figure 9, we can observe that Co-SoRTS and HaRTS present less number of context switches than SoRTS. It happens because application tasks run concurrently with Co-SoRTS or HaRTS schedulers, eliminating unnecessary context switches. In addition, these approaches reduce the jitter increasing the predictability of the real-time system.

4.2   Deadline fails

Figure 10 illustrates the number of fails (vertical axis) taking into account different time of context switches (horizontal axis).
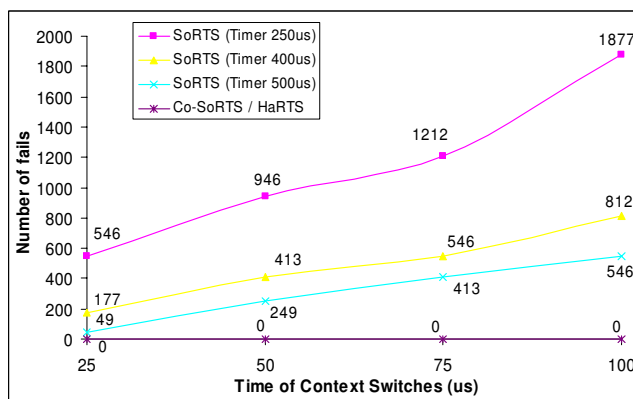

Figure 10: Number of fails after 10 seconds of execution.

We can observe that an increase of the context switch time also increases the number of fails for SoRTS scheduler implementation. Besides, Figure 10 shows that for SoRTS approach, the enlargement of the RTOS interrupt interval (250, 400 and 500 us) reduces the number of fails. Furthermore, as larger as the time of context switches,

lesser is the available CPU time for tasks execution (Figure 11), increasing the number of context switches and inducing the system to fail. Figure 10 shows that Co-SoRTS and HaRTS do not present deadline fails. We can conclude that Co-SoRTS and HaRTS are indicated scheduling approaches for hard real-time applications.

4.3   CPU utilization

Figure 11 shows the CPU availability for task execution (vertical axis) considering different time of context switches (horizontal axis).

In SoRTS approach, the increase of the context switches time decreases the CPU availability for tasks execution. This behavior is expected since the number of context switches and fails increases. As a result, the CPU wastes more time accomplishing context switches, providing less time to execute tasks. The context switch time does not affect the CPU availability for Co-SoRTS and HaRTS since for these approaches the scheduler executes in parallel with the application tasks.
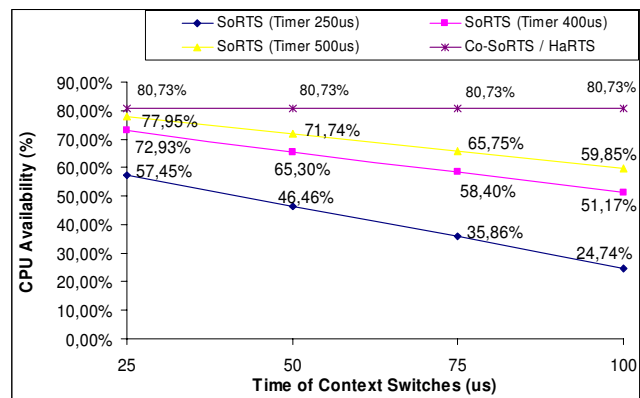

Figure 11: CPU availability.

5   Conclusions

This paper compares three scheduler implementations: SoRTS, Co-SoRTS and HaRTS, in order to relate application classes with scheduler approaches. The idea is to find out which approach is the most suitable for a given application class.

For all applications HaRTS scheduler implementation always achieved better performance results, fulfilling all application deadlines. Co-SoRTS and HaRTS have similar results. However, a scheduler implemented in the dedicated hardware of HaRTS can be implemented compromising less energy and area consumption if compared to an equivalent one implemented in a Co-SoRTS co-processor. The overall system performance for schedulers implemented in the same processor than application tasks

(SoRTS approach) is more affected by relative variations of the RTOS interrupt interval and the time necessary for context switch.

It's important to consider the implementation efforts and cost of each approach. The HaRTS approach is more complex and expensive compared to Co-SoRTS or SoRTS approach due to the complex nature of the hardware implementation. Comparing Co-SoRTS and SoRTS approaches, we also find an extra complexity. Relating the considerations discussed here, we conclude that Co-SoRTS and HaRTS present the best results for hard real-timer application. On the other hand, SoRTS is suitable for soft real-time systems.

Future work includes the development of a MPSOC RTOS scheduler that allows the processor task migration in an efficient way.

6   Acknowledgments

7   References

[1] Y. Cho, S. Yoo, K. Choi, N-E. Zergainoh, A. Jerraya. **Scheduler implementation in MPSoC Design**. In: Asia South Pacific Design Automation Conference (ASP-DAC'05), 2005, pp. 151-156.

[2] D. Andrews, D. Niehaus, and P. Ashenden. **Programming models for hybrid CPU/FPGA chips**. IEEE Computer, v. 37(**1**), 2004, pp.118–120.

[3] J. Ortiz. **Hardware/Software co-design of schedulers for real time and embedded systems**. Master's thesis on Computer Science, University of Kansas. 2004. Available at: http://www.ittc.ku.edu/research/thesis/documents/jorge_ortiz _thesis.pdf.

[4] P. Kohout, B. Ganesh, and B. Jacob. **Hardware support for real-time operating systems**. Design, Automation and Test in Europe Conference (DATE'03), 2003, pp. 45–51.

[5] V.Mooney III, J. Lee, A. Daleby, K. Ingstrom, T. Klevin, and L. Lindth. **A comparison of the RTU hardware RTOS with a hardware/software RTOS**. In: Design Automation Conference (DAC'03), 2003, pp. 683–688.

[6] M. Barabanov. **A Linux-based Real-Time Operating System**. Master's thesis, New Mexico Institute of Mining and Technology. 1997. Available at: http://www.fsmlabs.com/images/stories/pdf/archive/thesis.ps.

[7] K. Lahiri, S. Raghunathan, and S. Dey. **System-level performance analysis for designing on-chip communication architecture**. IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, v. 20(**6**), 2001, pp. 768-783.

[8] V. Mooney III. **Hardware/software partitioning of operating systems**. In: Design, Automation and Test in Europe Conference (DATE'03), 2003, pp. 338–339.

[9] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, and M. Imai. **Hardware implementation of a real-time operating system**. In: 12th TRON Project International Symposium (TRON'95), 1995, pp. 34–42.

[10] T. Samuelsson, M. Åkerholm, P. Nygren, J. Stärner, L. Lindh. **A Comparison of Multiprocessor Real-Time Operating Systems Implemented in Hardware and Software**. In: International Workshop on Advanced Real-Time Operating System Services (ARTOSS'03), 2003.

[11] W. Wolf. **Computer as components: principles of embedded system design**. Morgan Kaufmann Publishers, 2001, pp. 688.

[12] G. Buttazzo. **Hard real-time computing systems: predictable scheduling algorithms and applications**. Kluwer Academic Publishers, 1997, pp. 400.