On the design of flexible real-time schedulers for embedded systems

Hugo Marcondes[1], Rafael Cancian[2],
Marcelo Stemmer[2], Antônio Augusto Fröhlich[1]

hugom@lisha.ufsc.br, cancian@das.ufsc.br, marcelo@das.ufsc.br, guto@lisha.ufsc.br

[1] Laboratory for Hardware and Software Integration
[2] System and Automation Department
Federal University of Santa Catarina
PO Box 476 - Florianópolis - Brazil
88040-900

Abstract

A major subject in real-time embedded systems is the management of the tasks that compose the system respecting its real-time constraints. This is usually accomplished by the adequate selection of a scheduling policy. This work proposes a design and implementation of real-time schedulers for embedded systems within the context of Application Oriented System Design (AOSD), enabling the development of schedulers where the policy is detached from the scheduling mechanism. The results shows that such implementation could scale from 8 bits microcontrollers, 32 bits architectures, and to specific hardware implemented design.

1 Introduction

Embedded Operating Systems (EOS) should be adapted to provide only the necessary support for a well-defined target-application. Factoring out the operating system into selectable and configurable components is a good way to model and design an EOS. However, in the realm of dedicated systems, an integrated design of software and hardware is often induced. In order to accomplish with the system requirements, the design of such systems need to deal with a diversity of hardware architectures, as 8-bits microcontrollers to dedicated chips (ASICS), and digital signal processors (DSP). In this scenario, the task of modeling and implementing components that can be effectively reused in this distinct architectures is a challenge.

The adaptations needed to an EOS meet the requirements of the target-applications usually spreads among the components of the system. As an example, depending on the application, the EOS may need to support mono task, cooperative tasks, or concurrently tasks. Only in the later case a task scheduler is needed, and it may be implemented concerning with several features as policies, dynamic priority, preemption, etc. Schedulers also depend on time managers like alarms, and alarms depend on timer counters. They also need to know specific information that belong to the tasks that they will schedule (such as priority, arrival time, deadline, and period). The concurrent tasks also depend on synchronization mechanisms, such as semaphores and mutexes, which in turn need the schedulers; if they are real-time synchronization mechanisms, then they will depend on timers as well (i.e. to timeout P operations in a semaphore). This illustrates how related all those abstractions are. In this way, successfully adapt them to application requirements demand a good engineering strategy.

Even in a single family, the adaptation of components to different execution scenarios may not be easy. Task schedulers, for instance, have a myriad of algorithms and features, including relatively complex and highly specialized real-time policies. It is still a big challenge allow an embedded operating system to support any scheduling algorithm (real-time or not), independently of its features, and without requiring adaptations on the rest of the system.

The integrated design of software and hardware on

embedded systems allows features typically found on operating systems to be implemented in hardware, by using programmable logic devices or even designing ASICs. Nowadays is not rare implementing task schedulers on hardware, because this is a very often used component and it is source of considerable overhead. Indeed, an application oriented operating system has to allow their components to be implemented on both the domains (software and hardware) in an efficiently way.

All these problems can not be solved only with careful implementation. They need an appropriated and ingenious system design. In this paper, we focus on the description of the analysis and modeling of the components related to task schedulers and advanced implementation aspects that, only together, allow the appropriated solution for these challenges. We used domain engineering and a method that aggregates a set of programming paradigms and guide the design of systems adapted to the application. Our contributions includes an efficient model and the correspondent implementation to adapt schedulers on application oriented embedded operating systems, also allowing its execution on different architectures, including small 8 bits microcontrollers.

This paper is organized in this way: section 2 presents concepts and the description of some schedulers, as well as the description of the main techniques used in this work. The sections 3 and 4 present the development held and the results obtained, including the theoretical model and aspects of the proposed implementation. Finally, the section 5 presents some conclusions and final considerations.

2   Related Work

The tasks scheduling is considered the heart of a system and dozens of distinct algorithms have been proposed, and most of them are real-time schedulers for specific applications classes. Many schedulers can be reduced into a simple sort of a ready tasks queue, according to a specific criterion. Indeed, this includes the most known schedulers, such as FIFO, round robin, priority, SPF (*Shortest Process First*), RM (*Rate Monotonic*), and EDF (*Earliest Deadline First*) [7]. However, several other algorithms are much more complex than that. Algorithms such as DSS (*dynamic sporadic server*) and *dynamic priority exchange server* [3] need separated queues for periodic and aperiodic tasks, and at least one special periodic task to deal with the aperiodic tasks with specific timing rules, consumption, and budgets grant. Algo-

rithms such as *Elastic Task Model* [6] allow changes on tasks parameters, such as its period, in order to adapt themselves to the current system load. Therefore, to support them in a transparent and efficient way for embedded systems is far from be a simple task.

Several embedded operating systems already allow the adaptation of their schedulers, even dynamically. However, this adaptation is always restricted to a few and simple specific algorithms, such as FIFO, round robin, priority, EDF, and RM. Besides that, many real-time operating systems use scheduler algorithms that do not considerate the tasks' deadline, i.e. they use non real-time schedulers to schedule real-time tasks. Finally, in addition to require guarantee of the tasks' deadline, many embedded real-time applications require further more from the schedulers. As an example, multimedia applications, require that the constant execution rate of audio and video tasks (minimizing the jitter). To deal with this kind of requirement is necessary the use of specific algorithms, such as CBS (*Constant Bandwidth Server*) [6], just because usual algorithms do not take the jitter into account. From this we can conclude that there is no adequate support for this kind of application when the EOS does not provide specific schedulers, and this is the case of many EOS, including the real-time ones.

Several researches on hardware/software co-design for real-time systems have been proposed in this and in last decades. Hardware support for task schedulers was proposed, among others, by [9], that had implemented a cyclical schedule, and by [10], that had implemented the RM and EDF priority algorithms. Beyond the support for tasks scheduling, [8] had developed hardware support for time and events management, because these activities are very often on the real-time systems and have a high intrinsic parallelism. However, this support is limited to fixed priority schedules. The *HThread* project [2] proposes a programming model that allows tasks implemented on hardware interact with tasks at software, by the implementation of schedulers and synchronization devices on both the domains (hardware and software). Others kinds of support had also been proposed, like memory management [13] and resource access protocols [1], that had implemented the priority inheritance protocol to prevents deadlocks and unlimited task blocking. Supports that are more complete include, beyond the scheduling, inter process communication, interrupts management, resources management, synchronization and time management. This

support on hardware is usually called real-time Unit (RTU).

In this scenario, the EPOS (*Embedded Parallel Operating System*) raises up as one viable choice of multiplatform real-time operating system for embedded systems. The EPOS includes frameworks and tools for operating system generation, and it is result of the *Application-Oriented System Design* (AOSD) [4], that combines several design paradigms that aims guide the development of high adaptable e reusable components. The AOSD brings innovations as scenario adapters [5] e hardware mediators [11] that allow high efficiency on automatic generation of application dedicated systems.

EPOS does not address only the automatically generation of the software support for an specific application, but also the support for hardware blocks (IPs - *Intellectual Properties*) necessary and sufficient to them, i.e., the application oriented and automatic generation of SoCs (*Systems-on-a-chip*). The extensions on EPOS to SoCs generation are based on the abstraction concepts, hardware mediators and IPs, associating one IP for each mediator [12]. Currently the EPOS has functional support for several architectures such as IA32, PPC, SparcV8, MIPS, and AVR.

3 Analysis and Design

The analysis and design process begin with domain engineering, following the guidelines of the AOSD methodology, which enables the identification of the main commonalities and differences between the concepts that compose the domain. Using this strategy the main entities related to real-time scheduling were identified. The figure 1 presents the design of such entities.

In this design, the task is represented by the class `Thread` and defines the execution flow of the task, implementing the traditional functionality as the *suspend* and *resume* operations. This class models only aperiodic tasks. Periodic tasks, a common abstraction of real-time systems, are in fact a specialization of the `Thread` class which aggregate the mechanisms related to the re-execution of the task periodically, using the `Alarm` abstraction, responsible for reactivating the task when a new period begins. The `Alarm` abstraction uses the `Timer` hardware responsible to manage the timing duties of the system.

The classes `Scheduler` and `SchedulingCriteria` define the structure that realizes the task scheduling. Traditional design and implementations of scheduling algorithms are usually done by a hierarchy of special-
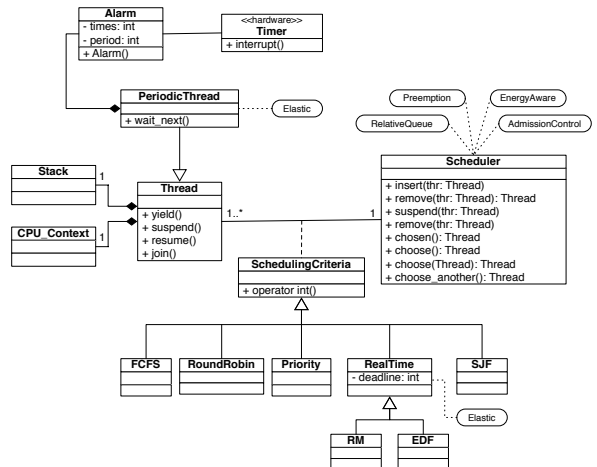

Figure 1: Proposed task scheduling design

ized classes of an abstract `Scheduler` class, which can be further specialized to bring new scheduling policies to the system. In order to reduce the complexity of maintenance of the code (generally present in such hierarchy of specialized classes), as well as to promote its reuse, our design detaches the scheduling policy (criteria) from it's mechanisms (lists implementations) and also detaches the scheduling criteria from the thread it represents. Such division of responsibilities is yield from the domain engineering process.

Such separation of the mechanism from the scheduling policy was fundamental for the construction of the scheduler in hardware. In fact, the hardware `Scheduler` component implements only the mechanisms that realize the ordering of the tasks, based on the selected policy. In this sense, the same hardware component can realize distinct policies, without any hardware reconfiguration, as the definition of the policy is confined in the `SchedulingCriteria` component. This is achieved by the isolation of the element's comparison algorithm of the scheduler in the criteria, analogous to the separation of algorithm and the elements of data structures defined in the STL library.

Additionally to the analysis and domain engineering process, several characteristics were identified as configurable features of those components. In fact, such characteristics represent fine variations within an entity of the domain, which can be set in order to change slightly the behavior of the component. Among such configurable features, preemption, admission control of tasks, energy consumption

384

and changes in tasks' information were identified as slightly variations of the scheduler. Admission control of tasks (i.e. based on CPU utilization of the current set of tasks), as well as the consideration of energy consumption of energy could be evaluated as configurable features responsible by the implementation of quality of service policies (QoS). The capability that a scheduler have to change the properties of tasks that are been used is useful for several more complex algorithms. As seen on section 2, elastic scheduling algorithms (as the *elastic task model*), assume that the period of a task could be changed, as the CPU utilization rate are getting higher or lower. Others schedulers, as the CBS and DSS (section 2) have analogous behavior. Such characteristic is designed as a configurable features that are applied to the `SchedulingCriteria` related to periodic tasks, as well as the `PeriodicThread`, enabling the functions to change the period of one task, once the scheduler requests. In this sense, algorithms that are more complex could be supported and adapted without incurring new specialization of classes.
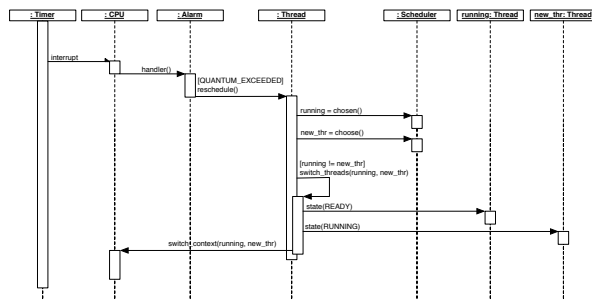

Figure 2: Task rescheduling sequence diagram

In order to illustrate the interactions between the components of the proposed design, the figure 2 presents the interactions of the components during the rescheduling, occurred when the time slice of the current task expires. In this context, the `Timer` is responsible for generating periodic interruptions, which are counted by the `Alarm`. When the CPU time slice (quantum) given to the current running thread is expired, the `Alarm` invokes the `Thread` method responsible for rescheduling the tasks. Then, the rescheduling method verifies which is the actual running thread, as well as to verify which one should be the next to run, invoking the *choose()* method from the `Scheduler`. This method returns a pointer to the thread that should be running. The actual running thread pointer is compared with the pointer returned by the *choose()* method, in order to verify if a switch

context of the CPU is necessary. If a switch context is necessary, the threads states are updated and the corresponding method for switch the CPU context is invoked.

4 Implementation and Results

This section presents the implementation details of the main components of the proposed scheduler schema, specially the implementation of the scheduling mechanism in the software and hardware domain. The main scheduling policies implemented through the `SchedulingCriteria` are also presented.

4.1 Software Scheduler

The implementation of the software scheduler follows the traditional design of lists. Such list implementation its realized as a conventional ordering list of its elements, as well as a relative list, where each element stores its ordering parameter relative with its predecessor. In this sense each element will hold the difference of its ordering parameter from the previous element, and so on. Such kind of implementation is necessary when the scheduling policy has dynamic priority that increases over time, as the very known EDF policy, as an example. In such policy, as the absolute deadline is always a crescent value, the use of a conventional ordering, using the absolute deadline will lead to an overflow of the variable (which can occur in a few hours on 8 bits microcontrollers). Instead of, the use of a relative queue insures that the deadline is always stored relatively to the current time, and in this way, the variable will never overflow.

Independently of the use of relative queues or conventional one, the criterion used by the ordering algorithm of the queue is realized by the `SchedulingCriteria`. In general, this component can be visualized as a specialization of the integer type, which defines the ordering of the queue. Policies that are more complex can be established by overloading its arithmetical operators. As example, in the case of multi-queues algorithms, a `SchedulingCriteria` can encapsulate two parameters for ordering: the identification of the queue, and the priority of the element inside that queue, as well as overload the comparison operator less-equal ($\leq$) in order to evaluate both parameters when the elements are compared to establish its position inside the queue implemented on the `Scheduler` component. This approach allows the efficient implementation of more complex scheduling algorithms.

4.2 Hardware Scheduler

The component `Scheduler` was also implemented in the hardware domain. The figure 3 illustrates the organization of the logical blocks of this component.
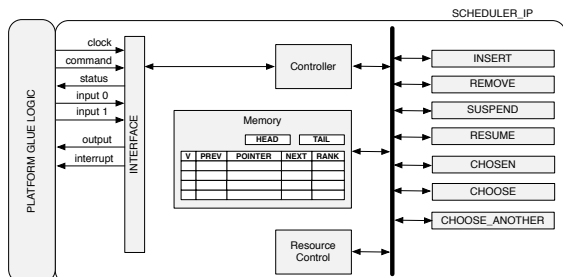

Figure 3: Block diagram of the proposed component in hardware.

The implementation of the scheduler in hardware follows a well-defined structure. It has an internal memory that implements an ordered list. One module (`Controller`) is responsible for interpreting all the data received by the interface of the component in hardware and then to activate the process responsible for implementing the functionality requested by the user (through the `command` interface register). This implementation, as the software counterpart, realizes the insertion of its elements already in order, that is, the queue is always maintained ordered, following the information that the `SchedulingCriteria` provides.

It worth's highlight two aspects of the implementation of this component regarding its implementation on hardware, especially for programmable logic devices. Both of these aspects are related to the constraints in terms of resources of such devices. Ideally, a hardware scheduler should exploit as most the inherent parallelism of the hardware resources. However, such resources are very expensive, especially when the internal resources are used to implement several parallel bit comparators to search elements on the queue, as well as to find the insertion position of an element in queue. Moreover, the use of 32 bits pointers to reference the elements stored on the list (in this case `Threads`) becomes extremely costly for implementing these comparators. On the other side, the maximum number of tasks in an embedded system is usually known at design time, and for that reason, the resources usage of this component could be optimized by implementing a mapping between the system pointer (architecture word-size) and an internal representation that uses only the necessary

number of bits, taking into account the maximum number of tasks running on the system.

Another aspect is related to the search of the element's insertion position. Ideally, such searching could be implemented through a parallel comparison between all elements on the queue, in order to find the insertion point in only one clock cycle. However, such approach, besides increasing the consumption of the resources, as the number of tasks increase it could lead to a very high critical path delay on the synthesized circuit, and thus, reduce the operating frequency of the component.

By this reason, the insertion of elements was implemented doing a sequential search of the insertion position of the element, which will take N cycles in the worst-case. Besides the fact that this approach inserts variable time for this operation, such variation is hidden by the fact that the insertion could be realized in parallel to the software running on the CPU.

4.3 Evaluation

The evaluation of the proposed scheduler was realized implementing a synthetic real-time application, where a set of periodic tasks was defined. The implemented components were configured using the appropriate tools, which generate a set of configuration parameters binding the interface of the component with its implementation (that could be realized as software or as hardware, through the binding of the interface with its mediator).

The application was compiled for the PowerPC (32 bits) and AVR (8 bits) architecture, using the EDF, RATE MONOTONIC e PRIORITY. The table 1 present the footprint of the application for each selected policy and architecture. The tests also validated the implementation of each scheduling policy.

	PPC32		AVR8	
	.text	.data	.text	.data
EDF	51052	300	49246	853
Rate Monotonic	47908	272	36800	1003
Priority	47864	272	36790	1003

Table 1: Test application footprint

The tests were realized also using the scheduler in hardware. In this case, the experimentation platform was a VIRTEX4 FPGA, which combines on PowerPC 405 processor and logic cells, enabling the rapid prototyping of dedicated hardware accelerators.

# Máx. Tasks	Logic Usage	Slices	Máx. Freq.
2	5%	326	214.6 Mhz
4	10%	551	161.5 Mhz
8	19%	1078	138.8 Mhz
16	36%	2015	123.4 Mhz
24	51%	2833	114.6 Mhz
32	73%	3997	113.4 Mhz
48	103%	5665	82.0 Mhz

Table 2: FPGA resource utilization of the `Scheduler` component

The FPGA used on the experimentation platform (ML403) was the `XC4VFX12` that provides 5,412 slices of logic blocks for the implementation of the accelerators. The table 2 shows the consumed area in this FPGA, accordingly with the configured number of maximum task instantiation.

5 Conclusions

This paper presented a design of a flexible real-time scheduler. The use of refined techniques as domain engineering enabled the isolation of the differences of several scheduling policies, enabling a better reuse of the design artifacts (as scheduling policies and scheduling mechanisms), as well as providing a platform not only to design real systems, but also to do experimentation of new real-time scheduling algorithms. The tests show the feasibility of deploy the scheduler design on architectures that range from small 8 bits microcontrollers to 32 bits architectures and even dedicated hardware.

References

[1] B. Akgul. Hardware support for priority inheritance. In Kluwer Academic Publishers, editor, *24th IEEE International Real-Time Systems Symposium*, 2003.

[2] E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, E. Komp, R. Sass, and D. Andrews. Enabling a uniform programming model across the software/hardware boundary. In *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 89–98, 2006.

[3] Giorge Buttazzo. *Hard Real-Time Computing Systems.* Kluwer Academic Publishers, 1997.

[4] A. A. Fröhlich. *Application-Oriented Operating Systems.* PhD thesis, Sankt Augustin: GMD - Forschungszentrum Informationstechnik, 2001.

[5] A. A. Fröhlich and W. Schröeder-Preikschat. Scenario adapters: Efficiently adapting components. In *Proceedings of 4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, USA, 2000.

[6] G. Lipari G.C. Buttazzo and L. Abeni. Elastic task model for adaptive rate control. In *19th IEEE Real-Time Systems Symposium*, pages 286–295, Madrid, Spain, 1998.

[7] D. R. CHOFFNES Harvey DEITEL, Paul DEITEL. *Sistemas operacionais.* Prentice Hall, 2005.

[8] P. Kohout and B. Jacob. Hardware support for real-time operating systems. In *Proceedings of CODES - ISSS'03*, Newport Beach, CA - USA, 2003.

[9] V. Mooney and G. De Micheli. Hardware/software codesign of run-time schedulers for real-time systems. In *Proceedings of Design Automation of Embedded Systems*, pages 89–144, 2000.

[10] M. Shalan P. Kuacharoen and V. Mooney. A configurable hardware scheduler for real-time systems. In *Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms - ERSA'03*, 2003.

[11] F. V. Polpeta and A. A. Fröhlich. Hardware mediators: a portability artifact for component-based systems. In *Proceedings of International Conference on Embedded and Ubiquitous Computing*, volume 3207, pages 271–280, 2004.

[12] F. V. Polpeta and A. A. Fröhlich. On the automatic generation of soc-based embedded systems. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*, 2005.

[13] M. Shalan and V. Mooney. A dynamic memory management unit for embedded real-time system-on-a-chip. In *Proceedings of International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 180–186, 2000.