ON THE DESIGN OF FLEXIBLE REAL-TIME SCHEDULERS FOR EMBEDDED SYSTEMS

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

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

[2]*System and Automation Department*
Federal University of Santa Catarina
PO Box 476 - Florianópolis - Brazil
88040-900

{hugo,guto}@lisha.ufsc.br,{cancian,marcelo}@das.ufsc.br

Keywords: Hardware/software co-design; re-configurable architectures and applications

–

Contact Author:
Hugo Marcondes
Phone number: +55 (48) 8425-9911
Fax number: +55 (48) 3721-9516 ext: 16
hugom@lisha.ufsc.br

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

Blind-Review

Abstract

Embedded systems frequently imposes an integrated hardware/software design within real-time constrains. The essence of real-time is the management of the tasks that realize the system, and with the achievement of the real-time constrains of those tasks, which is usually done 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). The use of this approach enabled the development of schedulers where the policy is detached from the scheduling mechanism, fostering a better reusability of the scheduling components. The results show that such implementation could scale from 8 bits microcontrollers, 32 bits architectures and to specific hardware implemented design.

1 Introduction

Operating systems for dedicated systems, in contrast with general propose operating systems, should be adapted to provide only the necessary support for a well-defined application. With this in mind, factoring the operating system onto selectable and configurable components is a good way to model and to design dedicated operating systems. However, dedicated systems often induce an integrated design of software and hardware, needing to deal with a huge diversity of hardware architectures, from 8 bits microcontrollers to dedicated chips (ASIC), making hard the task of model and implement components that can be effectively reused in all these architectures.

The adaptations needed to Embedded Operating Systems (EOS) meet the necessities of the applications that they support usually require modifications on many components of the system. Therefore, according to the application, the EOS may need support mono task, cooperative tasks, or concurrently tasks. Only on the later case tasks schedulers needed, and they may be implemented by very distinct algorithms, may be separated in policies and mechanisms, may have fixed or dynamic priority, may be able to preempt or not, and so on. Besides that, schedules need time managers like alarms, and alarms need timer counters. Schedules also need to know specific information about the tasks that they will schedule (such as priority, arrival time, deadline, and period). The concurrent tasks also need synchronization mechanisms, such as semaphores and mutexes, which in turn need the schedules; if they are real-time synchronization mechanisms, then they will need timers as well (to timeout P operations).

Even in a single family, the adaptation of components to different execution scenarios may not be easy. Task schedules, for instance, has a myriad of algorithms and features, including relatively complex and highly specialized real-time schedules. At these cases, to allow an EOS to support any algorithm (real-time or not), independently of its features and without need adaptations on the rest of the system, is a big challenge, especially on an embedded system with resource constrains.

The integrated design of software and hardware on embedded systems allow features typically found on operating systems to be implemented in hardware, using programmable logic devices or even designing ASICs; Is usual implementing task schedules on hardware, because this is a very often used component and it is source of considerable overhead. So an operating system adaptable to the application has to allow this component to be implemented on both the domains (software and hardware) in an efficiently way.

All these problems cannot be solved only with a carefully implementation. They need of an appropriated and ingenious system design. At this paper, we focus on the description of the analysis and modeling of the components related with the schedulers and advanced implementation aspects that, only together, allow the appropriated solution for the presented problems. On the modeling was used the domain engineering together with 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 presentation of implementation techniques to adapt schedules on application oriented and 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 schedules found at bibliography, as well the description of the main techniques used in this work. The sections 3 and 4 present the development held and the obtained results, 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 the most are real-time schedulers for specific applications classes. Many schedules can be reduced onto a simple short of a read tasks queue, according to a specific criterion. This affirmation includes the most known schedulers, such as FIFO, round robin, priority, SPF (*Shortest Process First*), RM (*Rate Monotonic*), and EDF (*Earliest Deadline First*) [9]. However, other algorithms are much more complex. Algorithms such as DSS (*dynamic sporadic server*) and *dynamic priority exchange server* [7] 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. Algorithms such as *Elastic Task Model* [8] allow changes on tasks parameters, such as its period, in order to adapt their self to the current system load, and several other examples could illustrate the major differences between the dozens of task schedulers proposed in the literature. So, support them in a transparent and efficient manner for embedded systems is not an obvious task.

Several embedded operating systems already allow the adaptation of their schedulers, even dynamically. However, this adaptation always is restricted to a few specific algorithms, such as FIFO, round robin, priority, EDF, and RM. Besides that, many real-time operating systems use schedule 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 guarantees of meeting the tasks? deadline, many embedded real-time applications require further more of the schedulers. Among the several examples, we can cite the multimedia applications, that requires that the execution rate of the audio and video tasks to be constant (minimizing the jitter). To deal with this kind of requirement is necessary the use of specific algorithms, such as CBS (*Constant Bandwidth Server*) [8], simply because the usual algorithms do not consider the jitter. From this we can conclude that there is no adequate support to this kind of application when the EOS do not provide specific schedulers, and this is the case of many EOS, including the real-time ones.

Most real-time operating systems available today, such as Embedded RT Linux, QNX, and VxWorks, have their practical use on deeply embedded platforms limited because of the generated code size and the difficulty of portability. Besides that, most real-time operating systems not take in to consideration software and hardware co-design and then, ignore the possibilities of hardware configuration. So, though exist many supposedly available choices of real-time operating systems for embedded systems, only a few of then are really applicable on several architectures (mainly the ones with bigger resource constrains) and adaptable to the needs of the target applications.

Several works of hardware/software co-design for real-time systems have been proposed in this decade. Hardware support for task schedulers were proposed, among others, by [11], that had implemented a cyclical schedule, and by [12], that had implemented the RM and EDF priority algorithms. Beyond the support for tasks scheduling, [10] had developed 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 [6] 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 [5]. This last one implement the priority inheritance protocol to prevents deadlocks and unlimited task blocking. Supports that are more complete includes, 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).

On this scenario, the EPOS (*Embedded Parallel Operating System*) raise 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 a result of the *Application-Oriented System Design* (AOSD) [2], that combines several design paradigms that aims guide the development of high adaptable e reusable software components. The AOSD brings innovations as scenario adapters [1] e hardware mediators [3] that allow high efficiency on automatic generation of application dedicated operating systems.

Subsequently the EPOS was extended to automatically generate not only the software support, but also the support for hardware (IPs - *Intellectual Properties*) necessary and sufficient to the application, i.e., the application oriented and automatic generation of SoCs (*Systems-on-a-chip*), already detailed on previous publications. The extension on EPOS to SoCs generation is based on the abstraction concepts, hardware mediators and IPs, associating one IP for each mediator [4]. The software engineering really efficiently used for the EPOS design and its expansion for SoCs generation provides a base to it also be a multiplatform real-time operating system, and to fit co-design aspects and partitioning for a greater space project exploration when occurs integrated design of software and hardware. 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 begun with the task of domain engineering, following the guidelines of the used methodology, enabling 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 was identified. The figure 1 presents the design of such entities.

In this design, the task is represented by the class `Thread`, which defines the execution flow of the task, implementing the traditional functionality of such kind of abstraction, as described in the literature. This class models only aperiodic tasks. Periodic tasks, a common abstraction among 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

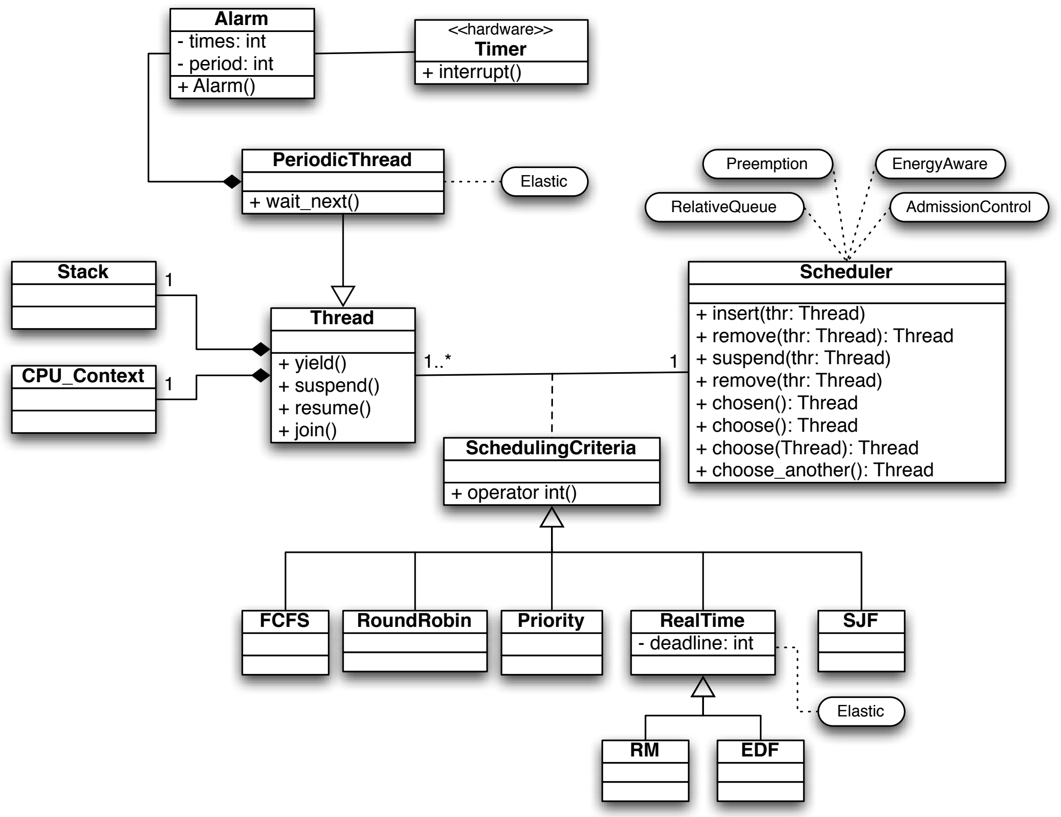

Figure 1: Proposed task scheduling design

reactivate the task when a new period expires. 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 specialized 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 it is yield from the domain engineering process that enabled the identification of all common aspects of the scheduling policies, enabling the separation of those aspects (confined in the `Scheduler` class) from the keys aspects that characterize the policies (implemented in the `SchedulingCriteria` component).

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 comparison algorithm between the elements of the scheduler queue in the criteria, analogous to the separation of algorithm and the elements of data structures defined in the STL library.

Additionally, on 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, the preemption was identified, as a slightly variation of the scheduler, as it?s enabled the preemption of a thread when a higher priority thread is ready for receive the CPU resource. 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).

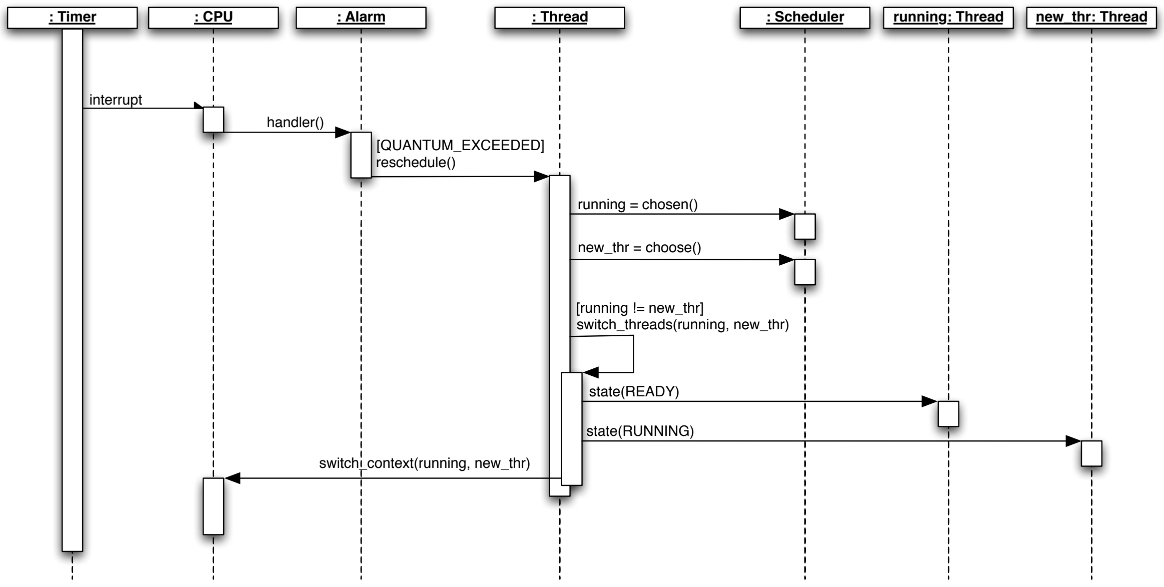

Figure 2: Task rescheduling sequence diagram

Another characteristic identified are related to scheduler that have to change the properties of tasks that are been used. 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.

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) provided to the current running thread is expired, the `Alarm` will invoke the `Thread` method responsible for rescheduling the tasks. Then, the rescheduling method will verify which is the actual running thread, as well as to verify with that should be running now, 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 (as this method could return the actual running thread if no higher priority thread is present in the ready queue). If a switch context is necessary, the threads states are updated and the corresponding method for actually switch the context of the involved threads.

4 Implementation and Results

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

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 and on. Such kind of implementation is necessary when the scheduling policy has dynamic priority increases over time, as the 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 as the execution time is always growing (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 not overflow. To better illustrate this question, consider the figure 3. This figure shows the behavior of the relative scheduling queue, after the occurrence of some events, as explained above.

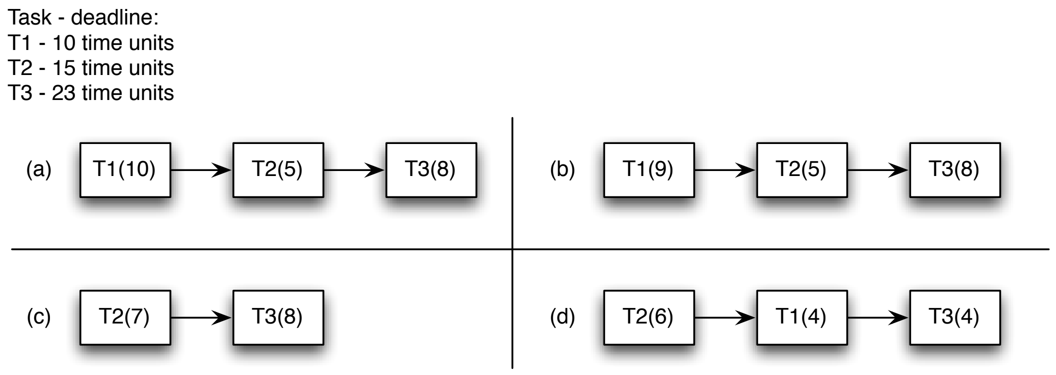

Figure 3: Relative scheduling queue behavior.

Consider the EDF scheduling, and the following tasks with its respective deadlines: T1 - 10 tu, T2 - 15 tu and T3 - 23 tu, where *tu* is the abbreviation for time unit. The figure 3(a) shows the scheduling queue just after the three task was activated. In this situation, the head of the queue stores its current deadline, while the other elements stores its deadline relative to the previous element. In this way, to calculate the actual deadline of an element, it?s necessary to sum its value, with the value of all its antecessors. At each occurrence of one time unit, the deadline of all tasks are decreased, however, as the queue is implemented relatively, only the value of the head element should be decreased, and in this way, all the others are updated implicitly. The figure 3(b) illustrate the queue after the occurrence of one time unit.

When the task finishes, it is removed from the scheduler and the remaining deadline is added to the next element, in order to maintain the coherence of the queue. The figure 3(c) present the scheduler queue when the Task T1 finishes its execution, after 8 time units, and, the remaining value of the deadline is added to the next element. In such scheme, deadline misses could be signalized by the turning to negative value of the head element of the queue. The figure 3(d), shows the scheduling queue when the task T1 is re-activated. Note that in this moment the task T2 has a deadline of 6 time units, thus has higher priority than T1, by the EDF scheduling policy. In this way, the task T1 is inserted between the tasks T2 and T3, adjusting the values of these components to maintain the coherence of the queue.

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 4 illustrates the organization of the logical blocks of this component.

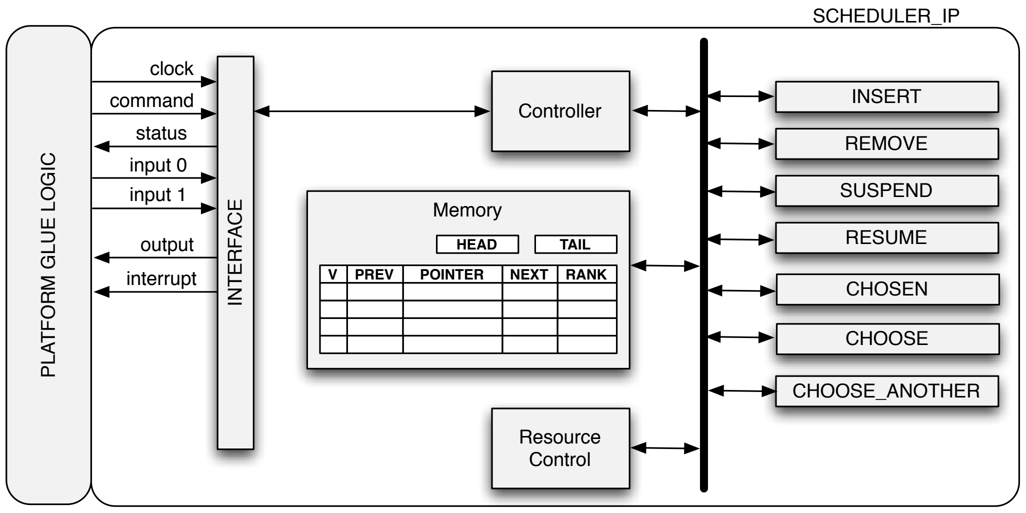

Figure 4: Block diagram of the proposed component in hardware.

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. In the memory of the component, a double-linked list is implemented.

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 in order 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 the comparators to search those elements. On the other side, the maximum number of task that a system will execute in an embedded system is usually know at design time, and for that reason, the resources usage of this component could be optimized implementing a mapping between the system pointer (32 bits) and an internal representation that uses only the necessary number of bits, taking into account the maximum number tasks running on the system.

Another aspect is related to the search of the position of insertion of the element on the queue. 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, such complex circuit, as the number of tasks increase, could lead to a very high critical path delay on the synthesized circuit, and thus, reducing the operation 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 in the worst-case will take N cycles. Besides in this approach, the insertion time could the variable, such variation is hidden by the effect 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 figure 5 shows the code of the test application. The figure 5(a) presents the implementation of each task, that simulates the task execution consuming the CPU cycles. The figure 5(b) presents the main application which is responsible for create and activate the tasks that will execute on the system (lines 5?10), defining the scheduling criteria, accordingly to the policy selected on the system. In this example, the EDF scheduling policy is used, defining the period of the task, the deadline, and the number of activations.

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.

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

```
int Tn(int n, RTC::Microsecond wcet) {
    RTC::Microsecond now, miliexec;
    int activations = 0;

    while (1) {
        now = Alarm::elapsed();

        // waste time in CPU
        miliexec = 0;
        do {
            while (now == Alarm::elapsed());
            miliexec++;
            now = Alarm::elapsed();
        } while (miliexec < wcet);

        activations++;
        Periodic_Thread::wait_next();
    }
    return 0;
}
```

(a)

```
int main() {
    cout << "Main will create the periodic threads...\n";

    Periodic_Thread *t1, *t2, *t3;
    t3 = new PeriodicThread(&Tn, 3, 60e3,
                                        SchedulingCriteria(300e3,300e3,10))
                                                    ;
    t2 = new PeriodicThread(&Tn, 2, 40e3,
                                        SchedulingCriteria(200e3,200e3,10))
                                                    ;
    t1 = new PeriodicThread(&Tn, 1, 20e3,
                                        SchedulingCriteria(100e3,100e3,10))
                                                    ;

    cout << "Main will wait for periodic threads to finish ...\ n";
    int status1 = t1->join();
    int status2 = t2->join();
    int status3 = t3->join();

    cout << "Main will destroy periodic threads...\n";
    delete t1; delete t2; delete t3;

    cout << "Main will finish...\n";
    return 0;
}
```

(b)

Figure 5: Test application: (a) task code and (b) task creation

	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 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 that the proposed system could scale from 32 bits architectures to very deeply embedded system running on 8 bit microcontrollers.

References

[1] Ommited - blind review. In *Proceedings of 4th World Multiconference on Systemics, Cybernetics and Informatics*, 2000.

[2] *Ommited - blind review.* PhD thesis, -, 2001.

[3] Ommited - blind review. In *Proceedings of International Conference on Embedded and Ubiquitous Computing*, volume 3207, 2004.

[4] Ommited - blind review. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*, 2005.

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

[6] 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.

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

[8] 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.

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

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

[11] 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.

[12] 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.

[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.