CPU Architecture Based on a Hardware Scheduler and Independent Pipeline Registers

Vasile Gheorghita Gaitan, *Member, IEEE*, Nicoleta Cristina Gaitan, *Member, IEEE*, and Ioan Ungurean, *Member, IEEE*

***Abstract*— Task switching, synchronization, and communication between processes are major problems for each real-time operating system. Software implementation of the specific mechanisms may lead to significant delays that can affect deadline requirements for some applications. This paper presents a hardware scheduler architecture integrated into the CPU structure that uses resource remapping techniques for the pipeline registers and for the CPU working registers. We present an original implementation of the hardware structure used for static and dynamic scheduling of the task, unitary management of events, access to architecture shared resources, event generation, and a method used for assigning interrupts to tasks that insures an efficient operation in the context of real-time control. One assembler instruction is used for simultaneous task synchronization with multiple event sources. This architecture allows a task switching time of one clock cycle (with a worst case scenario of three clock cycles for special instructions used for external memory accesses) and a response time of only 1.5 clock cycles for the events. Some mechanisms for improving program execution speed are also taken in consideration.**

***Index Terms*— Hardware scheduler, microprocessors and microcomputers, pipeline processors, real-time and embedded systems.**

I. Introduction

THE use of the current commercial and free real-time operating systems (RTOSs) for embedded systems encounters, in our opinion, two major problems. While the first one refers to the interrupt handler, the second addresses the fact that a task cannot synchronize simultaneously with events used for synchronization, resource sharing, and communication. Such events would be signals, semaphores, mutexes, messages, flags, and others. These problems were identified in RTOSs that run on microcontrollers without a virtual memory management unit and cache memory. Examples include the following RTOSs: μITRON, μTKernel, μC/OS-II, EmbOS, FreeRTOS, SharcOS, XMK OS, eCOS, Erika, Hartik, KeilOS, and PortOS.

The first problem, generated especially by the interrupt service routines on the simultaneous occurrence of many interrupts, is the jitter. Because of it, it is difficult to calculate the worst case execution time, an important component of the real-time systems. This can lead to deadline misses. A second problem is the extension of the task's execution time. This extension is generated by successive calls of the RTOS application programming interface (API) functions used to detect the occurrence of one of the events listed above. Another important issue is the time spent by RTOSs for task context switching (context switching is an operation carried out by the RTOS scheduler that requires a lot of time). Furthermore, API function calls become time consuming, especially if the processor requires transition from the user mode to the supervisor mode, and vice-versa.

The current general-purpose processors are used for embedded systems, but they can create problems, mainly due to nondeterministic performance and inefficient power consumption. In order to avoid these problems, conservative design techniques may be adopted. These techniques can create an oversized platform, enabling the proper behavior under worst case conditions. As a consequence, the use of these processors has limited applicability and they are unsuitable for the embedded systems with hard real-time features and low power consumption requirements. On the other hand, currently, field-programmable gate array (FPGA) devices [1], [2] at more efficient prices and with equivalent capacity in logic gates (more than millions) are widespread [3], [4]. For this reason, we propose a hardware support for real-time OS functionalities [5], based on the FPGA systems.

We present a custom scheduler architecture that is a hardware design with replication of resources [program counter (PC), pipeline registers, and CPU general purpose registers] as defined in [6] and [7]. Our architecture is based on the microprocessor without interlocked pipeline stages (MIPS) architecture that was especially adapted to support the operation of the hardware scheduler as part of the CPU itself. It employs a set of four pipeline registers for each task, used to hold the running instructions of the CPU. The register file is replicated for each task. This allows a very fast context switching, simply by remapping the active context of the task to be executed. This architecture, called multipipeline register architecture (MPRA) in [7], replaces the stack saving methods with a remapping algorithm that enables the execution of the new task starting with the next clock cycle.

The new architecture is characterized as follows: it contains an original implementation of the hardware structure used for static and dynamic scheduling of the tasks, it enables unitary management of the events and interrupts, it provides access to shared resources, event generation, and also defines a method used to attach interrupts to tasks, thus

Manuscript received August 7, 2013; revised December 14, 2013 and April 24, 2014; accepted August 3, 2014.

The authors are with the Department of Computers, Electronics and Automation, Stefan cel Mare University of Suceava, Suceava 720229, Romania (e-mail: gaitan@eed.usv.ro; cristinag@eed.usv.ro; ioanu@eed.usv.ro).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2014.2346542

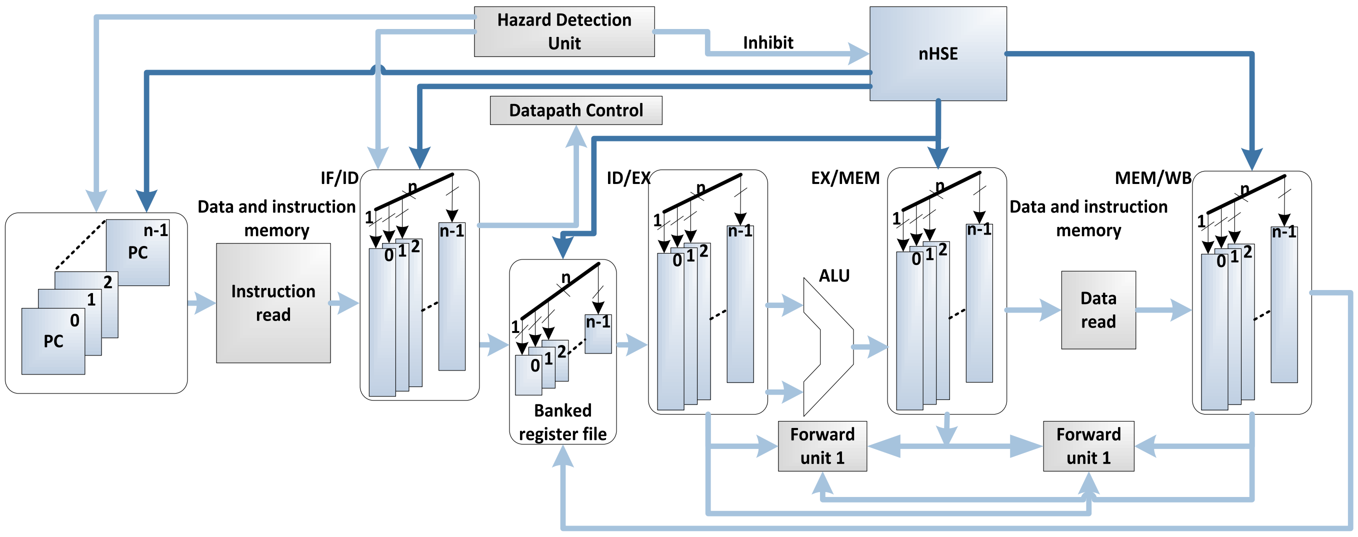

Fig. 1. nMPRA architecture. PC, IFID—instruction fetch instruction decode stage, ID/EX—instruction decode-execute stage, EX/MEM—execute-memory stage, MEM/WB—memory-write back stage.

ensuring an efficient operation in the context of real-time requirements.

The aim of the new architecture is to improve for microcontrollers the performances of the RTOSs. The performances are related to the following: task switching time, response time to external events, behavior of the interrupts, and execution time of the synchronization interprocess communication (IPC) primitives (events, mutexes, messages, and so on).

This paper is organized as follows. The nMPRA architecture is presented in Section II, and the nHSE architecture, including all RTOS facilities implemented in hardware, is presented in Section III. Section IV presents a series of tests carried out during the implementation of the proposed architecture. Section V includes related work and the comparisons with the nMPRA architecture. Finally, conclusions are drawn in Section VI.

II. nMPRA Architecture

We will refer to the enhanced architecture as nMPRA (for n tasks), while the embedded scheduler will be named hardware scheduler engine for n tasks (nHSE). The nMPRA architecture is shown in Fig. 1.

Before starting the description of the architecture, we have to define the following notation: an instance of the CPU will be named semi CPU (*sCPUi* for task i). Such a hardware instance comprises its own PC register, pipeline registers, register file, and its own control registers; it also runs the instructions of task $i(i = 0, \ldots, n-1)$. All *sCPUi* share the other components of the processor pipeline and, except for the *sCPU0* which is the only active unit after reset, are identical. The *sCPU0* is the only one allowed to access the monitoring and configuration registers of nMPRA (useful for scheduler and resource monitoring).

The new nMPRA is provided with both a static and a dynamic scheduler. The static scheduler is preemptive with static priorities. As a novelty, the scheduler can perform fast switching operations (specific to the MPRA architecture) at the occurrence of an event of the following type: time, interrupt, watchdog timer, mutex, deadline, intertask communication, and activation of the execution. The selection of these events can be achieved by executing a simple assembler instruction. Each such event can be activated or deactivated via a control register. The same feature applies to the dynamic scheduler.

Regarding the dynamic scheduler, there is a control register for each *sCPUi* (except *sCPU0*), which allows the changing of the task priority under the control of a dynamic scheduling algorithm. All events attached to the task inherit the new priority. For hardware interrupts, there is no specialized controller, but the architecture has a distributed scheme which allows attaching the interrupts to any task. The occurrence of an interrupt does not affect the contexts of the tasks (it does not flush pipeline registers for the instructions from the pipeline assembly line). In addition, during the execution, an interrupt may be redistributed to other task by a single WRITE operation to a register. The nMPRA was designed to respond to a wide range of time-related events, such as the periodic time event, watch dog timer event, and two deadline events (first is equivalent to an alarm and the second is equivalent to a fault). As a new architectural feature, we can indicate the presence of the deadline type events that can help the scheduling algorithms to meet the deadlines. In order to control access to shared resources, nMPRA implements mutexes in hardware. The mutexes are grouped in a mutex register file (MRF), which is composed of a set of registers that contain the mutex bit in the most significant bit of the register and the unique identifier (ID) of the proprietary task stored in the lowest priority bits. The access can be achieved by any task. The advantage of the scheme is that a single assembler instruction is used to acquire a mutex, faster than an API function call within an RTOS implemented in software.

Regarding the IPC, a unit containing events registers (ERs) is provided. These registers store the event state on the highest position bit. The next bits hold the ID of the task that activates the event, the following bits hold the ID of the destination task to which the event is assigned, and, finally, the last bits

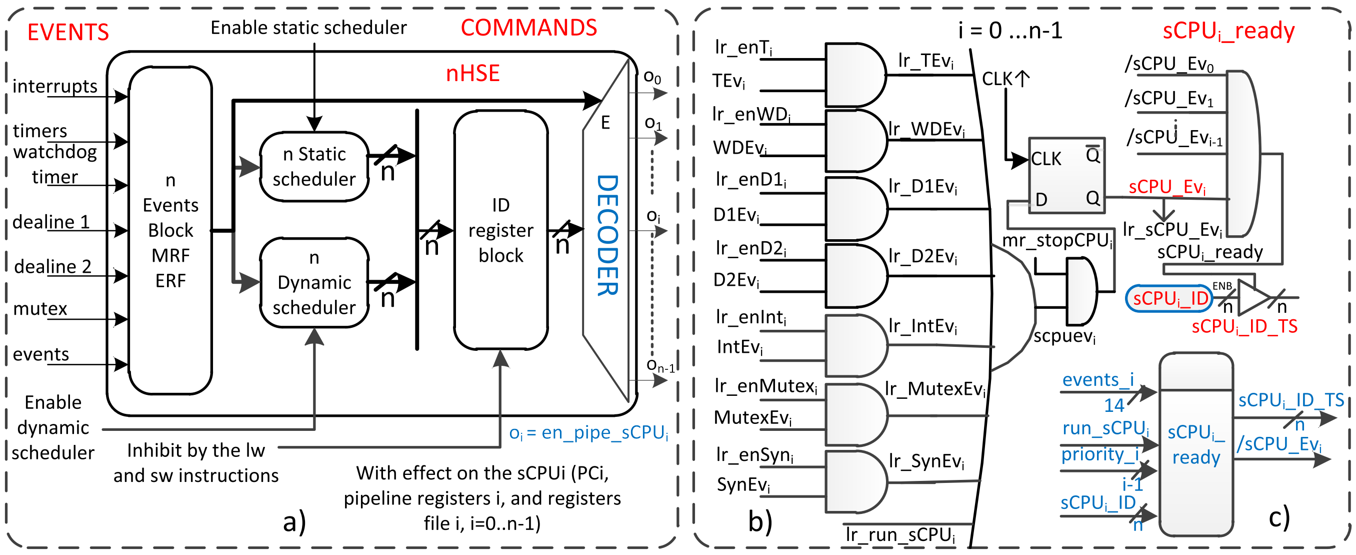

Fig. 2. nHSE architecture. (a) sCPUi level hardware scheduler (block of nHSE). (b) Digital logic for ready state. (c) Block diagram.

are used as a message field that the programmer can employ for any purpose. If the enabled event belongs to a highest priority task, a context switching operation is performed in the next clock cycle. Again, the advantage of the architecture is that it uses a single assembler instruction to activate an event, which is faster than the API function call within an RTOS implemented in software. As a consequence, if priority is high, the process of switching to the new task is very fast (1–3 processor cycles). This method eliminates entirely the time needed to search for a free event. If a task i is awakened by an event, it is important to find out the source of the event. But, searching the ER file (ERF) to find the source, may take an unacceptable amount of time. To avoid this problem, whenever the READ instruction is executed, the search is performed in one processor cycle, based on the content addressable memory (CAM) principle.

As we can see in Fig. 1, for each *sCPU*, there is an independent set of pipeline registers (IF/ID, ID/EX, EX/MEM, and MEM/WB), a PC, a register file, and a set of special registers (which are found in nHSE). Due to the fact that each task has its own set of pipeline registers and working registers for general purpose, the task context switching can be performed in a single clock cycle and the response to an external event can be achieved after 1.5 cycles. That is why, we can argue that the architecture is very fast, as it enables the tasks to switch with a minimum delay of 1 or 1.5 processor cycles and a maximum delay of three processor cycles (for working memory instructions). The other resources are shared by all tasks.

The run and idle counters for each *sCPUi* are implemented inside the *sCPU0*. This allows a periodic and precise deadline evaluation for each *sCPUi*. This choice takes into consideration the possibility to implement in software the dynamic scheduling algorithms on the *sCPU0*, always assigned with the highest priority.

In presenting the nMPRA, we define the following: the control registers with task level access (*cr*) specific to the *sCPUi*, the local registers (*lr_*) (that are part of the private memory space of each *sCPUi*), the global registers (*gr_*) (that are part of the global address space of the nMPRA and can be accessed by all *sCPU*), and the monitoring registers (*mr_*) (that can be accessed only by the *sCPU0*).

III. nHSE Architecture

The nHSE [Fig. 2(a)] is a finite state machine, which has inputs for events, such as interrupts, deadline, watchdog timers, timers, mutexes, messages, and self-support execution. Furthermore, nHSE has enabling signals for static and dynamic schedulers, and inhibits the signal generated by the execution of load and store instructions; the outputs are the activation signals of the *sCPUs* generated by the nHSE. Only one such output signal can be active at a given moment [as seen in Fig. 2(a), $oi \equiv en\_pipe\_sCPUi$]. The block diagram contains the ID register of the active *sCPU* together with the synchronization logic, the static scheduler, the dynamic scheduler, and the block related to the events. Each *sCPU* has a unique ID, which is an integer number from *0* to $n-1$. For example, if $n = 4$, we have four IDs 0, 1, 2, and 3. The ID register identifies the active *sCPU* at the occurrence of an event. If no event is active, the system is in idle state. The system becomes active when an event occurs, but if the *sCPU* attached clears the event, then the task will self-suspend. If the execution must be continued, the *sCPU* must activate the self-support event [Fig. 2(b) and (c)] before clearing the occurred event.

The static scheduler is task-oriented. The priority of each *sCPUi* is i, the same with the ID of the *sCPUi*; this means that priorities are constant during the execution of tasks. The static scheduler is disabled when the processor is connected to a power supply. The dynamic scheduler is provided with a priority register (PR) for each $sCPUi, i = 1, \ldots, n-1$. Based on these registers, every *sCPUi* may have a priority on a scale from 1 to $n-1$, where 1 is the highest and $n-1$ is

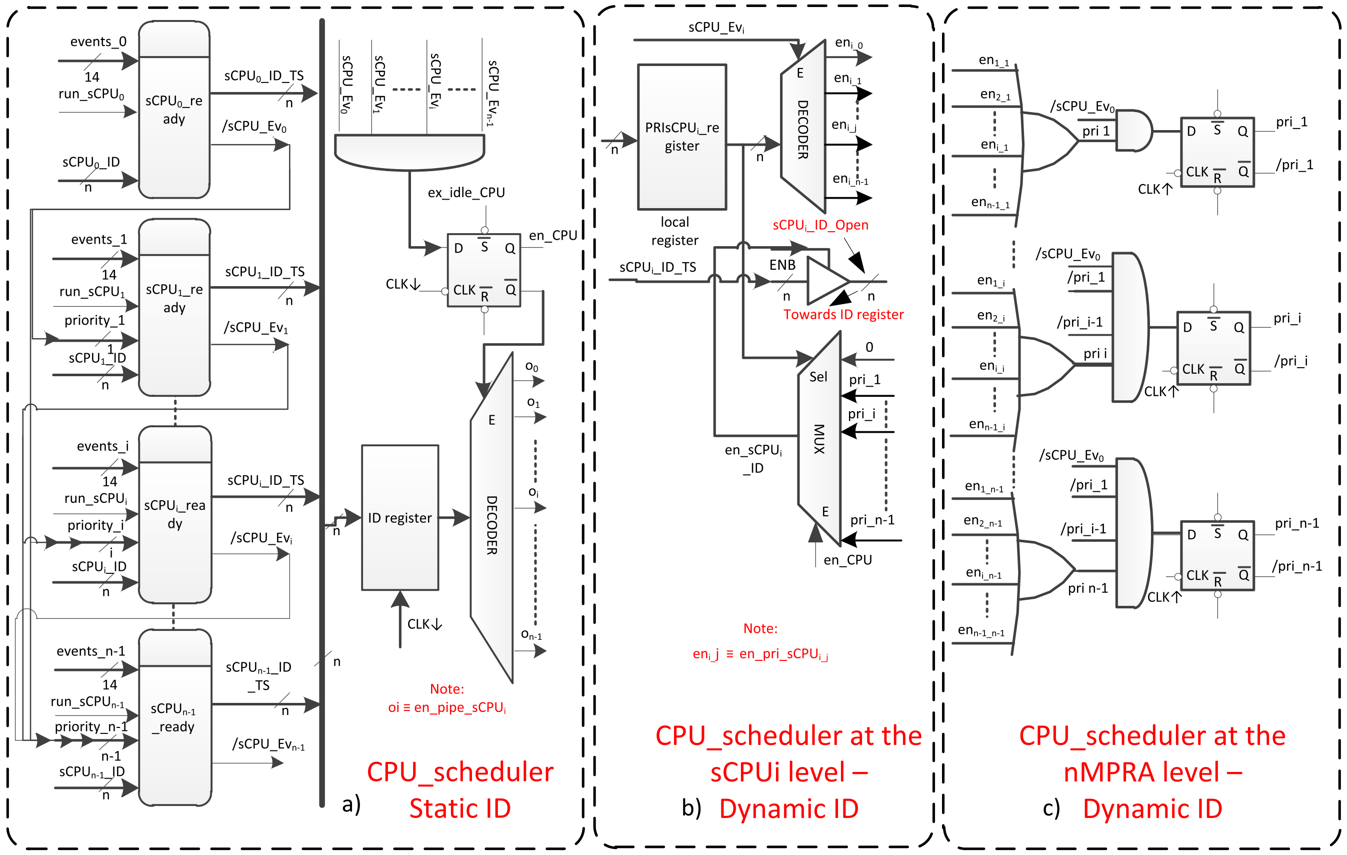

Fig. 3. Global hardware scheduler. (a) Static scheduler. (b) and (c) Support for dynamic scheduler.

the lowest. The *sCPU0* is always assigned with the highest priority (0) and that cannot be changed. The priority of a *sCPUi* can be changed dynamically by a dynamic scheduling algorithm implemented either in software, at the *sCPU0* level, or in hardware. When the processor is connected to a power supply, the dynamic scheduler is disabled; in this case, only the *sCPU0* remains active. The logic of events is divided among the *n sCPU*. The occurrence of an event (if it is validated) is signaled on the level of a *sCPUi*. If the *sCPUi* has the highest priority, the scheduler will validate its execution. In this case, either the system will be removed from its idle state, or other *sCPU* of lower priority will be stopped. Further, we present details about the generation of events and the nHSE design.

A. Hardware Static Scheduler

Nakano *et al.* [8] mention that hardware schedulers usually degrade the performance of the CPU pipeline. The present architecture has been designed taking into consideration all these disadvantages regarding the CPU pipeline.

The hardware structure of the scheduler belonging to each *sCPUi* (embedded in the logic block of the nHSE) is shown in Fig. 2(b). The scheduler constantly monitors the events that are associated to the *sCPUi*. Possible events of the *sCPUi* are: timer interrupts (*TEvi*), watchdog timer (*WDEvi*), two interrupts used for preventive signaling of the deadline (*D1Evi* and *D2Evi*), attached interrupts (*IntEvi*), mutexes (*MutexEvi*), synchronization and intertask communication events (*SynEvi*), and self-sustaining execution for the current *sCPUi* (lr_run_sCPUi).

Whenever a source generating an event/interrupt is cleared, the current *sCPUi* may lose the control of the CPU. The above-mentioned events can be validated with *lr_enTi, lr_enWDi, lr_enD1i, lr_enD2i, lr_enInti, lr_enMutexi*, and *lr_enSyni* signals. The only exception is *lr_run_sCPUi*. These signals must be stored in a special register, named task register (TR). The *sCPUiEvi* signal, which is used to signal the occurrence of an expected event, is enabled by the *mr_stopCPUi* signal. This is part of a monitoring register that is accessible only to the *sCPU0*.

The *sCPU0* is the only execution unit able to stop the other *sCPUi* ($i \neq 0$). For synchronization, we use a D flip-flop that stores information about a pending event on the rising clock of the CPU. The pending *sCPUi* signals imply the handler over the current *sCPUi* ID (*sCPUi_ID*). This action is performed by writing the value on the arbitration bus of the scheduler, if there is no task in execution, having a higher priority than that the *sCPUi*. The action is marked by the $/sCPU\_Ev0 \ldots /sCPU\_Evi - 1$ signals. A simplified block representation of the local scheduler, previously described, is shown in [Fig. 2(c)].

Fig. 3(a) shows the general design of the static scheduler. The schematic of scheduler shown in Fig. 3(a) contains the *sCPUi_ready* functional blocks (presented previously), the register that stores the ID of the highest priority *sCPUi*, and a decoder which activates the highest priority *sCPUi*. The AND gate and the D flip-flop from the schematic are activated when there is no other active *sCPUi*. The *en_CPU* signal can be used mainly for power saving. The activation or deactivation of any *sCPUi* specific resources can be accomplished with *en_pipe_sCPU0* through *en_pipe_sCPUn-1* signals. In conclusion, the proposed schematic can be used for static scheduling, if each task runs on a *sCPUi*. In this case, the static priorities are identified by the IDs of the tasks.

B. Hardware Dynamic Scheduler

Under certain conditions (for example, if a dynamic scheduling policy is used), some scheduling algorithms can bring performance improvement. The dynamic scheduler shown in Fig. 3(b) and (c) provides the possibility to set the priority for the *sCPUi* scheduling units, but it does not implement any specific scheduling algorithm. This hardware architecture can be applied to any of the solutions presented in [8]–[15]. The solutions can include external coprocessors or on-die implementations.

We need to emphasize that *sCPU0* always has the priority 0, which makes it the highest priority *sCPU* from the system. For the rest, we use a special register (*PRIsCPUi_register*) that stores the priority of the corresponding *sCPUi* for $i = 1$ to $n-1$. The priority is decoded by the decoder [shown in Fig. 3(b)], generating one of the signals $en\_pri\_sCPUi\_1, \ldots, en\_pri\_sCPUi\_n - 1$. As already mentioned, priority 0 is reserved for the *sCPU0*. The output of the same register is used for selecting the MUX multiplexor output from Fig. 3(b), which collects the prioritization scheme result from the inputs on the right side of the figure. The OR gates [Fig. 3(c)] allow the selection of priority for each *sCPUi* ($i = 1, \ldots, n$). The AND gate validates a certain priority, and the D flip-flop is used for synchronization with the system clock. The output of the multiplexor validates the value *sCPUi_ID_TS* which is the ID of the *sCPUi* at the input of the ID register. The same ID register was previously described in Fig. 3(a); the hardware structure from Fig. 3(c) is used in the same configuration for the dynamic scheduler.

The priority validation is activated by the *sCPU_Evi* signal only if the *sCPUi* expects the event. The *en_CPU* can be used as a global signal, part of a monitoring register that deactivates all *sCPUi*, except the *sCPU0*. The $pri\_1, \ldots, pri\_n - 1$ signals represent all $n-1$ possible priorities. The *PRIsCPUi_registers* that store the priority can be accessed as local registers for any of the *sCPUi* ($i \neq 0$) units. The access can be direct, through instructions, or, in the supervisor mode the *sCPU0* can READ/WRITE the registers.

The *PRIsCPUi_registers*, abbreviated PRs, are shown in Fig. 3(b). These registers can be found in any *sCPUi*, except the *sCPU0*. In order to use these registers, we propose the *wait Rj* control instruction that waits for the occurrence of any event marked in the *Rj* register (bits set to 1). The *Rj* is automatically transferred to the TR register. Whenever a *sCPUi* resumes execution after a wait instruction, *Rj* will store the occurred events. A more efficient and faster method implies the use of a dedicated mnemonic that stores the events as an immediate value*: wait Rj, events*. The events expected by the wait instruction are loaded in the TR register; when the task is resumed, these events are loaded in the *Rj*. We enhance this design by introducing a new register named ER that allows reading the signals shown in Fig. 2(b). This enables the identification of the occurred events attached to *sCPUi* and, optionally, their source (without reading the TR register). The instructions proposed for these registers are: *movcr TR, Rj; movcr Ri, TRj movcr EV, Rj; movcr Ri, EV*. In a similar way, for the PR registers we will use the following instructions: *movcr PR, Rj; movcr Rj, PR*. The TR and EV registers are control registers located in each *sCPUi*.

The *wait* instruction permits the synchronization of execution with the activation of several events. These events can be dealt with and cleared by software control according to the priority given by the *sCPUi*. The majority of the RTOSs have many built-in mechanisms for resource sharing, intertask communication, and synchronization, but their functionality is restricted to functions which implement these mechanisms. For example, an interrupt, the clearance of a semaphore, and the arrival of a message cannot be simultaneously waited. The solution proposed by this paper addresses this exact problem. Therefore, we can say that the wait instruction is almost as powerful as a software implemented RTOS.

C. Interrupt Events

The interrupts are very powerful mechanisms that allow the implementation of periodic, aperiodic, or sporadic events related to CPU operation. These events can be triggered by OFF-chip or ON-chip peripherals, or the execution of a program. According to the classical approach, the interrupt handlers are usually uninterruptible, and have always a higher priority than any real-time tasks.

A solution that handles interrupts as threads is presented in [16]. It was proposed for the Sun Solaris operating system and it is based on the unification of threads and interrupts into a single model. The interrupts are converted into threads, using a limited overhead. This allows a single synchronization model in the kernel.

The model proposed this paper, and described in Fig. 4(a), is similar to the interrupts as threads approach. In this new design, the interrupts are treated as events that are attached to real-time tasks, and therefore inheriting their priority. The system has p interrupts, and for each of them there is a global register (*INT_IDi_register*) with n useful bits that store the ID of the task to which the interrupt is associated. The activation of the *INTi* interrupt [Fig. 4(a)] validates the DECODER which activates one of the signals $INT\_i0, \ldots, INT\_in - 1$. The gate OR [Fig. 4(a)] can collect all interrupts from the system. They can be attached to *sCPUi* if all p *INT_IDi_register* ($i = 0, \ldots, p - 1$) registers are written with the i value. Correspondingly, no interrupt can be attached, if none of the p *INT_IDi_register* ($i = 0, \ldots, p - 1$) registers are written with an i value. The role of the D flip-flop is to synchronize the random appearance of the interrupt event *INTi* producing *IntEvi* [Figs. 2(b) and 4(a)]. This is accounted on the falling edge of the system clock.

The strong and interesting features of this design are the following: it does not contain a specialized interrupt controller (the interrupts inherit the tasks priority); a task can attach none, one, several, or even all the p interrupts from the

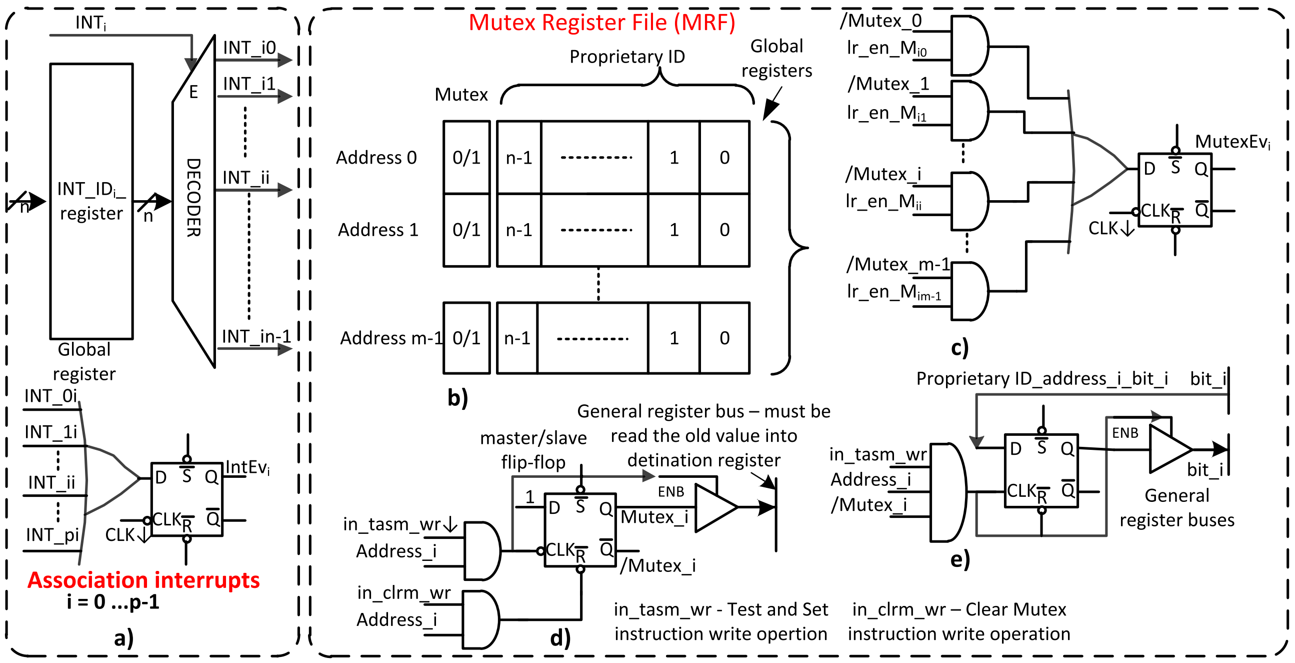

Fig. 4. (a) Implementation of the interrupts. (b)–(e) Implementation of the mutexes.

system; the programmer is able to establish the priority of the interrupts attached to the same task; an interrupt attached to a task can suspend a lower priority task; still, it cannot suspend the execution of the task to which it is attached, or a higher priority task; the interrupt can be a task, or can be attached to a single task. Furthermore, all interrupts can be attached to a single task; the interrupts do not affect the pipeline of other *sCPUi* units; the architecture does not imply saving or restoring of any context; interrupts can be nested and interrupts priorities can be dynamic (by reattaching to another task or by changing the priority of the tasks to which are attached).

The proposed solution has also some disadvantages, such as the limited number of possible nested levels, restricted to the number of *sCPUs*, or the lack of interrupt handler vectors. If more interrupts are attached to one *sCPU*, the handling order is assigned by the software, and this can lead to additional delays.

D. Time Related Events

As shown in Fig. 2(b), there are three types of time related events: 1) the periodic time events (*TEvi*); 2) the watch dog timer events (*WDEvi*); and 3) the deadline events (*D1Evi* is equivalent to an alarm and *D2Evi* is equivalent to a fault). For the implementation, each *sCPUi* has two dedicated timers. One of them has three comparators for *TEvi*, *D1Evi*, and *D2Evi*, while the other has only one comparator used for *WDEvi*. If the watchdog is not refreshed periodically, the *WDEvi* event can reset (if is activated) the *sCPUi*. For each of the two timers, the architecture has local registers (implemented in the local memory of each *sCPUi* and accessed with normal memory access instructions). In addition, these registers may be seen as monitoring registers that can be accessed by *sCPU0* with normal memory access instructions. The deadline values can be computed either with a local algorithm executed on *sCPUi*, with a global one that is executed on *sCPU0*, or even with a combination of the two. The architecture includes two timers for counting the CPU cycles, when a task is being executed or suspended. Hence, a software function can closely monitor the execution of a task on the *sCPUi*. The access to these counters can be done in the same manner as for the timers that were previously presented.

E. Mutexes

The mutual exclusion is an important topic related to the shared resources. The hardware support for the mutex, proposed by our design, is shown in Fig. 4(b)–(e). For implementation, we propose the solution shown in Fig. 4(b)–(e), which presents a set of global registers with fast access (for example, the access can be performed in the execution stage—EX). The MRF is composed of a set of m registers with length of $n+1$ bits. It contains the mutex state in the most significant bit, and the ID of the proprietary task in the lowest bits. The MRF registers can be accessed from any *sCPU* and therefore they are shared resources for all *sCPUi*. Each *sCPUi* has a hardware block, as the one described in Fig. 4(c), that generates *MutexEvi* events every time a blocked mutex is released. For each *sCPUi*, the decision related to the mutex taken into consideration is made based on the $lr\_en\_M0, \ldots, lr\_en\_Mm-1$ signals. These signals can be stored in local registers named enable mutex register (EMR). There can be one or several *EMRi* registers, depending on the number of mutex bits implemented in the MRF. The D flip-flop is used for synchronization with the CPU clock, and the information inside the element is latched on the rising edge of the CPU clock.

As shown in Fig. 4(d), the block and release operations of a mutex are performed in a single processor cycle, as an

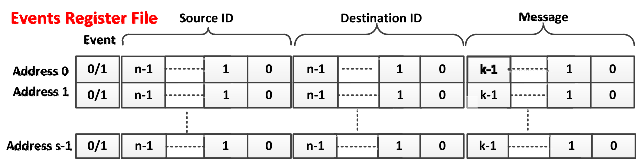

Fig. 5. ER file.

atomic operation. A test and set instruction (TAS—signals *in_tasm_wr* and *Address_i*) read the old value of a mutex bit from MRF (*Mutex_i*) and sets the *Mutex_i* bit. If the *Mutex_i* bit is set to 1, the mutex is considered available; the last *n* bits from the MRF register (*Address_i* signal) represent the ID of the mutex's owner. This is assured by the signals *in_tasm_wr, Address_i* and */Mutex_i* from Fig. 4(e). If the mutex bit is 0, then this is set to 1 and the ID of the *sCPUi* that executes the instruction is written into the appropriate MRF register [Fig. 4(e)—for the MRF bits different then mutex bit]. If the mutex read by the instruction is 1 and it is not assigned to a task, the value read from the ID register represents the ID of the mutex's owner. The erase instruction brings the *Mutex_i* to the value 0 using the *in_clrm_wr* and *Address_i* signals [Fig. 4(d)].

In working with mutexes, the following instructions can be used: *tst Rd, Rs* (*Rs* contains the mutex address and *Rd* contains the value from the selected MRF register), *clrm Ri* (*Ri* contains the address of the mutex), *movmr Rd, Rs* (it moves the mutex register—*Rs* contains the address of the associated mutex, and *Rd* contains the value from the MRF register without affecting the value of the register), *movcr EMRi, Ri* (it writes *Ri* in *EMRi*) and *movcr Ri, EMRi* (it writes *EMRi* in *Ri*). These instructions can be executed on all *sCPUi,* and, together with the *wait* instruction, assure the synchronization and the total access to all mutexes.

F. Intertask Synchronization and Communication

Intertask synchronization and communication are two other important aspects specific to RTOSs. A set of global registers with fast access is used for the implementation of the event mechanism. The ERF is composed of a set of registers of $(2n + k + 1)$ bits, shown in Fig. 5. These registers store the event state on the highest position bit. The following *n* bits store the ID of the source task that activated the event; the next *n* bits hold the ID of the destination task to which the event is addressed; finally, the last *k* bits are used as a message field available for any purpose. The ERF can be used with a certain behavior, when an event is activated, and with another, when the destination task reads the event in order to discover eventually where and what message has been sent. As a consequence, two different instructions are used. When a task wants to activate an event, it prepares the information that is part of an ER and executes a *wait* instruction. This action takes place without the need to specify the event address from the ERF.

The hardware block, shown in Fig. 6, generates automatically the address (starting from 0) of the first free event. Furthermore, this hardware block signals if all events are

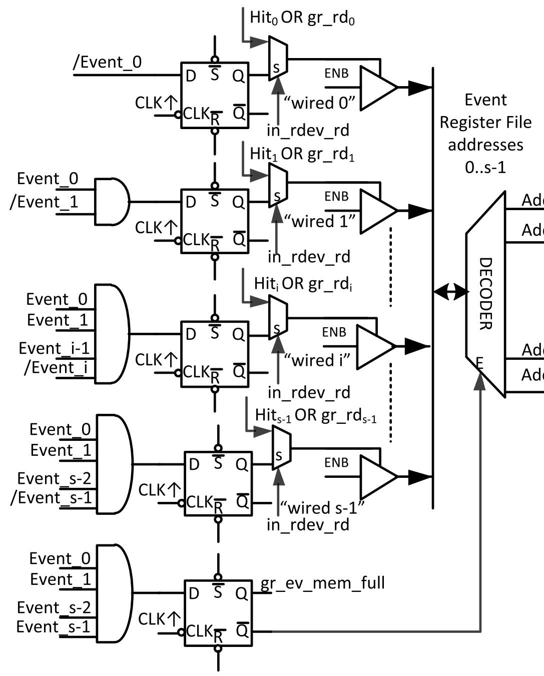

Fig. 6. Hardware for automatic generation of the next free event.

active (set to value 1). After the activation of a writing event instruction, the signal *in_rdev_rd* is 0, and the multiplexor is activated with the D flip-flop outputs, as represented in the schematic (Fig. 6). If *Event_0* (value stored in *ER0* on bit $2n+k$) is zero, then the signal */Event_0* has the logic value 1 and the write takes place in the D flip-flop associated with *Event_0*. If *Event_0* is set to 0, then */Event_0* has the logic value 1, and the write in all others D flip-flops is inhibited (Fig. 6). The */Event_0* value (1 logic value) is stored in the D flip-flop. This signal will pass through the multiplexor and will activate the three state-buffers. The 0 value is then written on the DEMUX input which activates the 0 address. If *Event_0* has the logic value 1, the first D flip-flop will write 0 after passing the multiplexor, will inhibit the three state-buffers, and disable the address 0. Moving on to the second D flip-flop, a logical value of 1 for the *Event_0* will validate the analysis of the *Event_1*. The same approach is used for all signals until event $s-1$.

In case all events are active, the AND gate from the last D flip-flop will generate a valid signal. This means that no free event is left and, in this case, a *gr_en_mem_full* is generated. The *gr_en_mem_full* is also used by the READ and WRITE instructions (as shown in Fig. 7) in order to signal that the event bits are busy. This signal is accessible through a global register accessible to all *sCPUi.*

Fig. 7 shows all generic signals needed to WRITE (*in_wrev_wr, Address_i* and */gr_ev_mem_full* signals) the event *i* corresponding to the D flip-flop and also the other bits (*bit_ij*) of the *ERi* register. A READ operation will automatically reset the D flip-flop that is associated with an event, by activating the *in_wrev_wr, Address_i* and hit signals. The validation of the three state-gates also allows reading the content of the other *bit_ij* bits that are part of the *ERi* register.

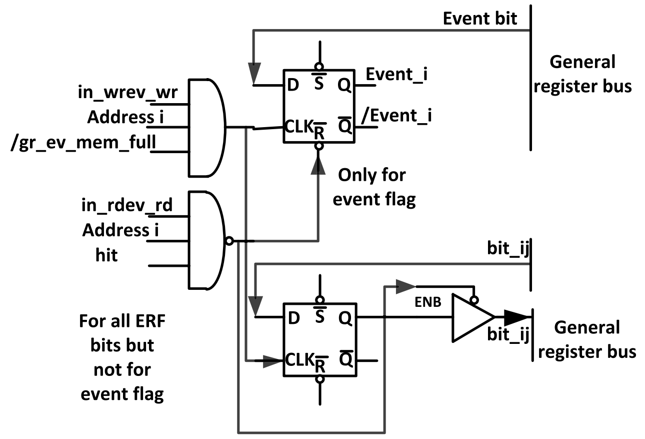

Fig. 7. READ and WRITE from ERF.

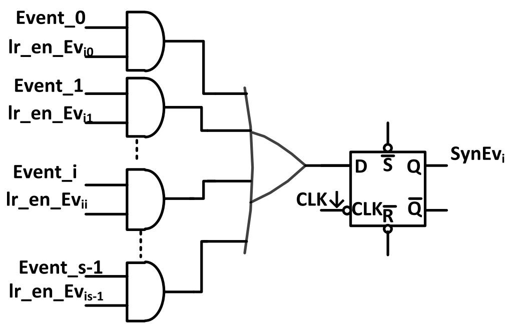

Fig. 8. *sCPUi* event enabling/disabling.

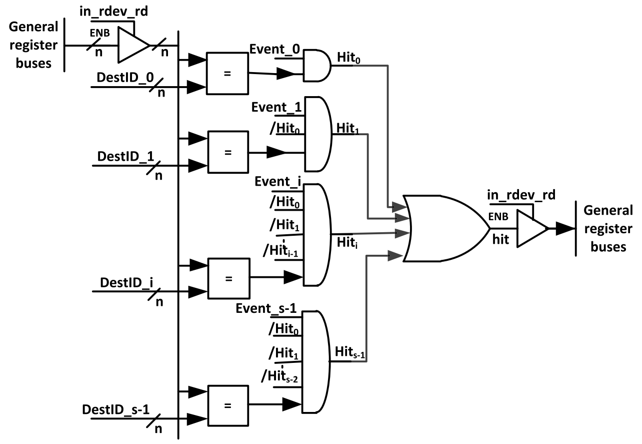

Fig. 9. ERi content read is based on a CAM principle.

This event activation method eliminates entirely the time needed to search for a free event.

The ERF registers can be accessed from any *sCPUi*; therefore, they are shared resources for all *sCPUi* units. Each *sCPUi* has a hardware block in nHSE (as shown in Fig. 8), which is used to generate *SynEvi* signals [Fig. 2(b)] every time a waited event becomes active. With the help of the $lr\_en\_Evi0, \ldots, lr\_en\_Evis - 1$ signals, each *sCPUi* can decide which event is taken into consideration. These signals are stored in the enable event register (EER) local registers. There can be one or several *EERi* registers, depending on the number of events implemented in the ERF. The D flip-flop is used for synchronization with the CPU clock. The information is stored on the rising edge of the CPU clock.

If a task i (*sCPUi*) is awakened by an event, it is important to identify the source of that event. Searching the ERF to find the source may take an unacceptable period of time. To avoid this situation, whenever the READ instruction is executed to read the *ERi* register from the ERF, the search is done based on a CAM principle, as shown in Fig. 9. The search starts with address 0 and ends on the first address for which there is a match between the destination ID task and the current task ID. A READ instruction can identify the match, detect the source task of the event, and find which message has been sent, simply by reading the contents of the ERF register.

The inputs of the comparator blocks contain the ID of the task that executes the instruction (this is done by activating the *in_rdev_rd* signal) and the destination values $DestID\_0, \ldots, DestID\_s - 1$. If there is a match, and the event is active, the subsequent hardware gates are blocked for all events with bigger index numbers (this approach is similar to the one shown in Fig. 6). In Fig. 6, the *in_rdev_rd* signal is used to force the multiplexor to use the value *Hiti OR gr_rdi* (global register read). Except for this addendum, the schematic view shown in Fig. 6 is working as explained previously. The explanations related to Fig. 7 must also be followed.

For working with events, the *event Ri instruction* can be used (*Ri* contains the source task ID, destination task ID, and the message that is stored in the last k bits). After the execution, *Ri* contains the value *gr_en_mem_full* that, before the execution of the instruction, was on the lowest bit position. If this bit has the value 1, the activation of the event fails. *clrev Ri*—after the execution of the instruction, *Ri* holds the content of the first ERF register for which a match is identified. If the event bit has the logical value 0, it means that the event is not active. The event list can be scanned recursively until all event sources have been cleared. This is the case for all tasks that are handling multiple events. The following instructions: *mover Rd, Rs* (move ER, where *Rs* contains the address of the associated event—when *gr_rdi* gets activated—and *Rd* contains the value of the selected ERF register—without affecting its value), *movcr EERi, Ri* (it writes the *Ri* in the *EERi*) and *movcr Ri, EERi* (it writes the *ERRi* in the *Ri*), are available for all *sCPUi.* Together with the *wait* instruction, they can ensure the synchronization, access to events, and message passing between tasks.

G. Further Considerations

After RESET, at boot time, nMPRA activates *sCPU0* and deactivates all others *sCPUi*. The *sCPU0* contains all initialization software and the startup sequences for all others *sCPUi* ($i = 1, \ldots, n - 1$). The *sCPU0* can treat all nMPRA functionality related to the hardware interrupts that may lead to fatal errors. The PCs for each *sCPUi* will be loaded with the address of a trap loop where the control is transferred. From this point, the program can jump to various code places in order to execute the software and enable the functionality

TABLE I
MEMORY REQUIREMENTS FOR REPLICATED RESOURCES OF THE nMPRA PROCESSOR

Number of tasks n	*Memory required for pipeline registers*	*Memory required for general registers*	*Total memory for replicated resources*
8	0.59kB	256B	0.84kB
16	1.18kB	512B	1.68kB
32	2.36kB	1024B	3.36kB

TABLE II
WORKING MEMORY FOR SOME MICROCONTROLLER FROM THE HIGH-PERFORMANCE CATEGORY

Architecture / Producer / Model	*Working RAM memory*
ARM Cortex M4/ ST Microelectronics/ STM32F427IG	256kB
Tricore / Infineon / Aurix 29X	780kB
SuperH / Renesas / R5S726A1P216FP	1366kB
SuperH / Renesas / R5S72681W266FP	2624kB

specific to each *sCPUi*. Each *sCPUi* can have access to both a local code and data memory, and to an external one.

IV. IMPLEMENTATION AND VALIDATION OF THE nMPRA ARCHITECTURE

This section describes operations that were performed to test and validate the nMPRA architecture. The architecture has been implemented and validated on a Virtex-6 FPGA ML605 Evaluation Kit—Xilinx. The code of the processor was described in standard very high speed integrated circuit hardware description language (VHDL). The nMPRA processor was implemented for a working frequency of 50 MHz. The nMPRA architecture uses four pipeline registers (including PC register) totaling 606 bits. The replication of these resources to 16 tasks requires 1.18 KB of RAM.

Although nMPRA is an architecture that involves shared resources, its implementation costs are more effective than those of other commercial architectures. It should be noted that such an implementation is suitable for a reasonable number of tasks. For a large number, the frequency will significantly decrease, due to a synthesis of logic with unreasonably high propagation times.

Table I presents the memory requirements for possible implementations of 8, 16, and 32 *sCPUs*. The values in the table have the purpose to illustrate the memory consumption for extreme implementation versions. Considering the data from Tables I and II, and given that nowadays microcontrollers use hundreds KB of RAM, we may definitely admit that the memory requirements needed to implement the nMPRA processor are more than acceptable. We adopted the example of Renesas SuperH R5S72681W266FP microcontroller that has 2.6-MB RAM for general use, without taking into account the memory required to implement the processor and other hardware components integrated into the chip. On the Virtex-6 FPGA ML605, the number of slice registers is 14 334 (out of 301 440: 5%) and the number of slice LUTs is 17 749 (out of 150 720: 12%) for 8 *sCPU* and the number of slice registers

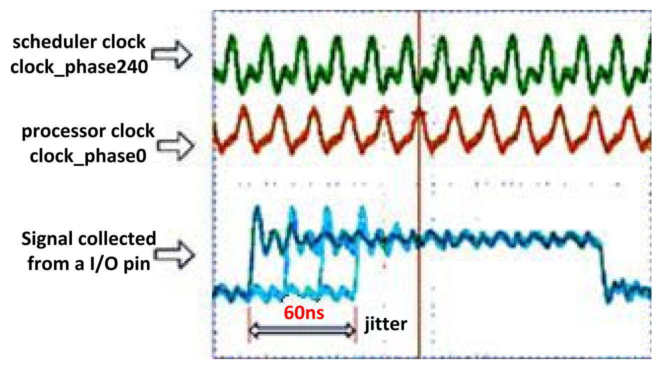

Fig. 10. Jitter of the highest priority task highlighted in relation with the event occurrence and execution start of the highest priority task (trigger on falling signal).

is 27 374 (out of 301 440: 9%) and the number of slice LUTs is 33 571 (out of 150 720: 22%) for 16 *sCPU*.

In order to be able to test the performance of this processor, we developed an assembler translator. This application also uses the VHDL files to validate the opcode of the instructions. This tool optimizes the time used by the translator in the process of adding new instructions to the VHDL files. In the current situation, a code may be written only in the assembler, while the translator will generate the machine-code as a VHDL written file. The development of a new application was necessary due to the fact that the proposed architecture extends the instruction set of the MIPS processor. After testing the functionalities of this processor, traditional MIPS compilation tools can be used to develop real-time applications.

If data must be exchanged via the common memory area, the monitoring of the presence of the load and store instructions on the assembly line must be performed. Only in this case, and during the execution of these instructions, the hardware scheduler engine (HSE) unit is not allowed to perform context switching operations (thus, the data consistency is ensured). In this context, we present the following test which underlines this situation, leading to the highest switching cycle—three clock cycles.

The test application consists of two tasks noted as *task0* and *task1*. *Task0* is initialized, sets an I/O pin on 1 and waits an event (a periodic interrupt generated by a timer). When the event occurs, *task0* sets the I/O pin with the value 0 and after some no operation (NO) instructions; it sets the pin with the value 1, and gets back in the wait state for the timer generated event. *Task1*, of less priority, is initialized and enters in a loop which includes a sw-type instruction.

There is a high probability of engaging in the locking state, which actually explains the jitter from Fig. 10 (a maximum delay of 60 ns—three machine cycles). As shown in Fig. 10, the maximum delay of the scheduler is 60 ns when MPRA works at a frequency of 50 MHz. Frequency is recorded in figure with $1/\Delta t$, while the period of the system clock is $\Delta t =$ 20 ns. The clock from the channel 1 is *clock_phase0* and is used to synchronize memory and pipeline registers; the clock from the channel 4 is used to synchronize the HSE scheduler (*clock_phase240*).

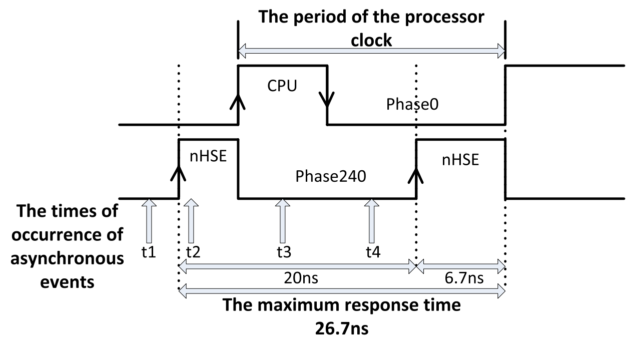
The period of the processor clock
CPU
Phase0
nHSE
Phase240
nHSE
The times of occurrence of asynchronous events
t1
t2
t3
t4
20ns
6.7ns
The maximum response time 26.7ns

Fig. 11. Scheduler response depending on the occurrence of an asynchronous event.

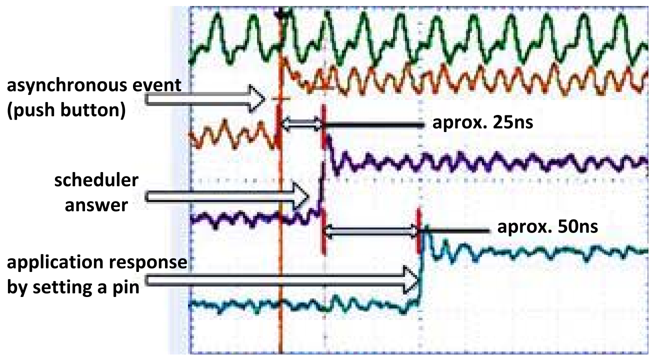

Fig. 12. HSE scheduler and software application response to an external asynchronous event.

Fig. 11 shows the response time of the system to an external interrupt. In this application, the registers used to set a pin, are periodically set or reset. Fig. 12 shows the asynchronous external event on channel 1 of the oscilloscope. In this case, the event is produced by pressing a button on the ML60 board. The scheduler response, which is a result of task switching, is indicated on channel 3 (Fig. 12) while the response of the software application, which sets the pin, is indicated on channel 2. At a frequency of 50 MHz, the scheduler response to an external asynchronous event can be up to 26.7 ns, depending on the occurrence time of the event. The internal logic block of the HSE requires no more than 6.7 ns to achieve the scheduling and remapping sequence of the contexts (Fig. 12).

The response of the software application comes at ~50 ns after task switching. It is noteworthy that 40 ns are required for the execution of a SW and OR instruction. Furthermore, to monitor the application's response, we use a registered port type which uses an internal D flip-flop to store the state. This D flip-flop is written on the falling edge of the *clock_ phase0* clock. This is a second reason for the delay. In order to monitor the clock, the asynchronous interrupt signal, and the response scheduler, we used the unregistered ports P7, P6, and P5. This means that these ports do not use any additional internal D flip-flops to store the state and that they will directly generate the output associated with the logical values that are written to the port. We used this type of port to achieve a realistic assessment of the scheduler response and to eliminate any delays caused by synchronization with the system clock in writing the port.

V. Related Work

This section provides a brief analysis of the results obtained within the last two decades on accelerating the schedulers and algorithm implementation of real-time kernel primitives in hardware. We start with the FASTCHART project proposed in [17] in 1991. The sources of nondeterminism in real-time embedded systems are given by the variation of the instruction execution cycle due to the presence of the pipeline, cache, asynchronous external interrupts, variable execution time of RTOS operations, based on the number of tasks and resources. All these can be reduced by moving the RTOS operations in hardware. This is the basic concept of FASTCHART. The FASTCHART manages 64 tasks with eight different priorities. There is no support for resources (mutexes, semaphores, etc.). The implementation was done as an RISC processor, without interrupt, pipeline, instruction, or data cache; therefore, the instruction cycle became deterministic. FASTHARD [9] is based on the previous FASTCHART project [17], but it is a hardware kernel that supports general-purpose processors. It uses a standard memory mapped address/data bus scheme. It supports facilities, such as rendezvous, external interrupts, periodic start of tasks, and activation and termination of tasks, without CPU intervention. Since FASTHARD [9], Adomat *et al.* [18] described a real-time unit (RTU) as a hardware-based real-time kernel. It supports 64 tasks, eight priority levels, periodic tasks with relative delays, binary semaphores, event flags, watchdogs, and interrupts, and it is managed simultaneously by three homogeneous processors connected to a Versa Module Europa (VME) bus.

Lindh *et al.* [19] describe a scalable architecture for real-time application (SARA), which can be used with RTU. The SARA is an extensible system, which uses individual processor cards with memory and bus controller which is interconnected with a motherboard using compact PCI. Lee *et al.* [20] integrate the RTU into the δ-framework [21] for SoC/RTOS codesign. Nordstrom *et al.* [22] adapt the software core of the μC/OS-II RTOS to a single processor version of the RTU, to improve performance, and in [23] they add the configuration of RTU for a single processor version.

The aim of the silicon TRON (STRON) [8] is to speed up the basic RTOS system calls and to reduce jitter in order to provide accurate timing predictions. The development is based on the TRON project, especially the μITRON specification for RTOSs, whose calls and functionalities have been implemented in a hardware core called STRON. The STRON coprocessor contains task management, flags, semaphores and timers, and external interrupts management. It is mapped to the external memory and it is seen as a coprocessor. The SPRING kernel is developed and described in [24] and has a completely different approach to the normal scheduling of an RTOS. The SPRING kernel uses dynamic and speculative scheduling based on a tree search and heuristic algorithms. Considering the execution time of tasks, deadlines, resources, and precedence constraints, the scheduling algorithms build a schedule plan with the help of which all tasks meet their deadline without ever getting blocked while waiting for resources. In [25], a hardware implementation is described as a spring scheduling coprocessor for the software kernel proposed in [24]. The dynamic scheduling algorithms have a major impact on the RTOSs overhead. To reduce the computation time of priorities when selecting a task and to provide time independence on the number of tasks, a dynamic scheduling coprocessor was implemented in FPGA, as described in [26] and [27]. It was developed as a scheduling accelerator, using the advantage of hardware parallelism. It can be configured for many algorithms of which the most advanced is enhanced least laxity first (ELLF), described in [26]. Vetromille *et al.* [10] make a relevant performance test between the systems that use a single processor, coprocessor, or an external dedicated hardware unit for implementing the scheduler. The testing parameters were the CPU utilization, the number of deadline misses, and the number of context switches. The authors also discuss additional overhead due to processor-coprocessor communication specific to software schedulers that are implemented in a separate CPU. Using the behavioral synthesis tool, Chandra *et al.* [28] describe how to automatically generate HW-RTOS from the POSIX API sources of eCos RTOS, thus increasing the system performance as a whole. H-Kernel [29] uses a specially designed CPU that has a set of registers for each task and a dedicated hardware RTOS. It achieves a 60% increase in performance for a specific application, such as multichannel real-time audio processing. H-Kernel implements tasks based on priority management, interrupt management, event blocks, queues, and time management. Song *et al.* [29] describe a processor-based solution which can be tailored to achieve an improved performance by using the modern FPGA codesign of HW/SW for a specific application. Advanced real-time multithreaded processor architecture (ARPA-MT) is described in [30] and uses modern FPGA technologies to achieve a predictable and customizable processor. The processor is accompanied by two hardware coprocessors Cop0-MEC and Cop2-OSC. The first one manages memory exceptions and interrupts, while the second implements all standard features of an RTOS. Orek is a real-time kernel written in C++. In [31], the motivation, design, and performance results obtained after porting all kernel functionality to hardware are presented. The aim is to improve determinism and performance. ARTESSO [32] is an RTOS implemented in hardware, plus some specific TCP/IP modules. ARTESSO is motivated by the fact that a high-performance embedded processor is required if the application needs to obtain data throughput for speeds of 100 Mb/s or more. The proposed architecture moves the kernel, checksum calculation, memory copying, and the TCP/IP header rearranging to hardware.

An interesting architecture is proposed in [5], one which uses a thread interleaved pipeline, scratch path memories, and a DRAM controller having compassable and predictable memory performances. This architecture allows concurrent programming, preserving at the same time their timing proprieties. Kuacharoen *et al.* [11] show a configurable hardware scheduler for real-time systems. The configurable hardware scheduler supports three scheduling algorithms: 1) property-based; 2) rate monotonic; and 3) earliest deadline first. The hardware scheduler also implements a specific controller for interrupts. It does not implement instructions, such as mutexes, semaphores, and so on, as they are taken over by the software part of the implementation.

Andrews *et al.* [33] describe hthread, which is a hardware/software codesigned multithreaded RTOS kernel. This kernel is part of a hybrid thread programming model being developed for hybrid systems which are comprised of both a software resident and a hardware resident concurrently executing threads. The kernel supports up to 256 active software threads, 256 active hardware threads, 64 blocking semaphores, 64 binary spin-lock semaphores, preemptive priority, round robin, and FIFO scheduling algorithm. The kernel contains a new interrupt controller named bypass interrupt scheduler which transforms asynchronous interrupt semantics into synchronous, controllable, and priority-based thread scheduling requests. Applications access the components of the hardware based operating system through familiar APIs, such as create, join, and exit.

We conclude that the main reasons for implementing RTOS, or parts of it in hardware are: reduction of memory footprint, decrease of execution overhead given by the basic functions (scheduling and context switching) and RTOS function call, jitter reduction or elimination, interrupts response time improvement and the possibility to better control their behavior (interrupts as tasks). One can also notice that there are two trends in hardware-based RTOS design and scheduling accelerators: a minor one, based on the design of specialized processors: the FASTCHART [17], H-Kernel [29], and ARPA [30] projects, and a major one, using the coprocessors, such as FASTHARD [9], RTU [18]–[20], [22], [23], STRON [8], δ-Framework [21], HW-eCos [28], OReK_ CoP [31], AR-TESSO [32], hthreads [33], or scheduling accelerators, such as ELLF_ SCoP [26], [27] or Kuacharoen [11].

In the following paragraphs, we present some comparisons between the solutions proposed in this paper and the ones described in Section IV.

The first comparison, as shown in Table III, is based on resource replication criteria, task communication speed and hardware or hardware–software implementation. Although it may seem like a simple solution, the nMPRA is the only unit ensuring the replication of the pipeline registers with functional effects on task switching speed. The ARPA-MT architecture alone carries out the IF and ID replication of the pipeline registers for each instanced task. However, it enters into competition for the last three pipeline levels—EX, MA, and RB. Although the solution of replicating the file containing the registers is an expensive hardware solution, it is also a new convergent factor toward reaching a switching speed between 1 and 3 machine cycles, reality which can only be met at the nMPRA. As can be clearly seen in Table III, even for a 50-MHz frequency processor, the switching speed of tasks is 60 ns. At the same time, the nMPRA is simply a hardware implementation, unlike the other implementations which are a hardware–software combination, inferentially leading to a longer switching time interval. It is noteworthy that hthreads

TABLE III
PROCESSOR AND RTOS ARCHITECTURE

Features	***nMPRA***	***hthreads [33]***	***ARPA-MT [30]***	***Kuacharoen [11]***
General register replicating	Yes	No	No	No
Pipeline register replicating	Yes	No	IF and ID	No
Tasks switching speed	1-3 processor cycles	1.7μs – 3.3μs 525-990 cycles (300 MHz)	3μs (72 cycles at 24 MHz)	125 cycles
Coprocessors	No	Yes	Yes	Yes -coupled on external processor bus
HW or HW/SW implementation	HW	HW/SW	HW/SW	HW/SW
Real parallel execution?	No	Yes for hardware threads	No	No

TABLE IV
COMPARING SCHEDULERS

Features	***nMPRA***	***hthreads [33]***	***ARPA-MT [30]***	***Kuacharoen [11]***
Support for static scheduling	Yes, HW	Yes, HW	Yes, Cop2-OSC	Yes, HW
Support for dynamic scheduling	Yes, HW	-	Yes, Cop2-OSC	Yes, HW
Algorithms for static scheduling	round robin, priority based scheduling algorithms	FIFO, round robin, priority based scheduling algorithms	Fixed priority policy based on period/ minimum inter-arrival time (RM-Cop2-OSC)	Priority based, RM
Algorithms for dynamic scheduling	Yes, SW	-	Dynamic priority policy based on absolute deadline (EDF Cop2-OSC)	EDF
Scheduler activation	HW - distributed	external FPGA, HW, SW	Coprocessor - Cop2-OSC	FPGA external, HW,SW
The number of scheduled tasks	32	256 SW, 256 HW	128	-

TABLE V
COMPARING INTERRUPT CONTROLLERS

Features	***nMPRA***	***hthreads [33]***	***ARPA-MT [30]***	***Kuacharoen [11]***
Specialized or distributed controller?	Distributed	Bypass Interrupt Scheduler(CBIS) - specialized	Coprocessor (Cop0-MEC)	Specialized
Interrupts as threads?	Yes	Yes	-	Yes
Interrupts can be attached to any task?	Yes	Yes, but a task can have attached a single interrupt	-	Yes
Interrupts inherit the task priority?	Yes	Yes	-	Yes
Interrupts affect pipeline?	No	Yes	-	Yes
Interrupt requires software context saving?	No	Yes	-	Yes

architecture allows the concurrent execution of hardware tasks and a software task.

The second comparison is based on the characteristics of the schedulers and is described in Table IV. The compared solutions refer to hardware-implemented static and dynamic schedulers, except for the hthreads. The nMPRA is the only one having the scheduler implemented at processor level and being able to gather the scheduling information directly from the hardware, without affecting a possible bus communication as is the case with the other solutions taken into consideration. All solutions propose static scheduling algorithms. Since scheduling performs the hardware switch toward the highest priority thread, we can admit that nMPRA is the fastest (it switches from one task to another with a delay of 1–3 machine cycles). As far as the implementation of dynamic scheduling algorithms is concerned, the ARPA-MT [30] and Kuacharoen *et al.* [11] solutions are more advanced, as they are hardware implemented. At present, the nMPRA does not have a hardware-implemented dynamic algorithm, as it is prepared to support dynamic priorities using the dynamic scheduler. As we have already indicated in Section III-B, we can implement some of the solutions proposed in the specialized literature. The scheduler activity is allocated and implemented to the nMPRA through the hardware. As for the other solutions, scheduler activation is attributed to the co-processor, requiring bus transfer cycles and even software intervention.

The third comparison refers to the interrupt system and is presented in Table V. We have found the concept of interrupt of threads in [11] and [33]. As for ARPA-MT [30], there is no explicit reference to interrupt management. In [11], the interrupt controller is briefly described, indicating that it supports eight interrupt levels. Each interrupt can be associated with a task, delegated to manage the associated levels of interrupts. Each interrupt may be set up as a fast interrupt (the interrupt handler suspends the current task) or a slow interrupt (handling task is inserted at the end of the priority line). Unlike the other implementations, the nMPRA interrupt system is completely allocated, so that an interrupt could be attached to one task only, while a task could have attached more interrupts (even all of them). A major advantage of the nMPRA is represented by the idea that it does neither affect the pipeline, NOR does it require register saving. In addition, interrupts are dealt in a unitary manner, just as the other events in the system.

The last comparison is shown in Table VI and refers to the implementation of the synchronization and communication primitives among tasks. The nMPRA is the only architecture, among the ones already presented, which implements these primitives in hardware. If an event occurs, like the ones described in Section III-F, and if it is associated with a task of a higher priority than the current task, then the task switch occurs with a delay of 1–3 cycles. The nMPRA may simultaneously synchronize with all supported events, using only one *wait* instruction. Any of the expected events which activate first shall initiate the task when it becomes the highest

TABLE VI
COMPARING MECHANISMS FOR TASKS SYNCHRONIZATION AND COMMUNICATION

Features	***nMPRA***	***hthreads [33]***	***ARPA-MT [30]***	***Kuacharoen [11]***
Hardware implementations	HW	HW/SW	Coprocessor (Cop2-OSC)	SW
Mutexes, semaphores?	Mutex HW	Semaphores HW/SW	Semaphores HW/SW	-
Events	Yes, HW	-	-	-
Messages	Yes, HW	-	-	-
How control is transferred to the scheduler?	HW	HW/SW	SW	-
Quick search in lists	HW (CAM)	-	-	-

priority task. As a conclusion, we can state that nMPRA does not implement queues for task states NOR task control block structures.

VI. CONCLUSION

This paper extends the basic idea presented in [6] and [7], defining original new functionalities for the nMPRA and nHSE architectures. The scheduler architecture has been defined with support for events (timer, watchdog timer, deadline, interrupt, mutex, and event). Two scheduling structures have been proposed: a static one, in which the ID and the task priority are equal to 0, as the highest priority, and $n-1$, as the lowest, and a dynamic one where the priority and ID are not necessarily the same. Here, the priority can be programmed inside each *sCPU* unit. This allows the implementation of scheduling algorithms with dynamic priorities.

This paper proposes an innovative solution for the interrupt mechanism. In this context, the interrupts inherit the priority and the behavior of the tasks to which they are attached. By using this approach, the behavior of an interrupt is more predictable in the context of a real time application (a task can be interrupted only by the interrupts that are attached to higher priority tasks). The proposed architecture has some interesting and powerful features described in Section III-C.

A set of mutexes implemented in hardware have been included in the architecture so as to restrict the access to the shared resources of the *sCPUs*. The mutex access is done in a single CPU cycle and it is atomic. The proposed solution allows a task that is waiting on a hardware mutex to auto-suspend itself until the mutex is freed without any overhead from the software or from RTOS. A set of events implemented as an ERF has been included in the architecture, in order to be used as part of an integrated intertask communication and synchronization mechanism. The ERF is also regarded as a shared resource for all *sCPUi*. The access for setting an event does not use as reference the event address (there are no search operations to find a free event).

As presented in Section III-A, the wait instruction is very powerful since it allows synchronization with several events. These events can be treated and cleared under software control, following a given priority which is imposed by the *sCPU* tasks.

Finally, we can conclude that the proposed nMPRA architecture is a very powerful one mainly because of the following aspects.

1) The switch between tasks is done usually in one clock cycle and in maximum three clock cycles when the CPU works with the global memory [7].
2) The system's reaction to an external event will not exceed 1.5 clock cycles if the event is attached to a higher priority task than the current task.
3) The pipeline is not reset; there is no need for context saving/restoration. Subroutine calls are accelerated through automatic parameter copying and register set remapping.
4) Internal high speed memory-based stack.
5) Powerful instructions for resource sharing, synchronization, and intertask communication.

As future work, we intend to create specific bench tests (currently, the existing bench tests cannot run on the proposed architecture because we introduced new instructions to use the nHSE) for new architecture in order to compare it with regular processors. Furthermore, we intend to apply the concepts presented in this paper on the advanced RISC machines (ARM) Cortex M architectures.

REFERENCES

[1] J. Shawash and D. R. Selviah, "Real-time nonlinear parameter estimation using the Levenberg–Marquardt algorithm on field programmable gate arrays," *IEEE Trans. Ind. Electron.*, vol. 60, no. 1, pp. 170–176, Jan. 2013.

[2] M. Shahbazi, P. Poure, S. Saadate, and M. R. Zolghadri, "FPGA-based reconfigurable control for fault-tolerant back-to-back converter without redundancy," *IEEE Trans. Ind. Electron.*, vol. 60, no. 8, pp. 3360–3371, Aug. 2013.

[3] M. Shahbazi, P. Poure, S. Saadate, and M. R. Zolghadri, "Fault-tolerant five-leg converter topology with FPGA-based reconfigurable control," *IEEE Trans. Ind. Electron.*, vol. 60, no. 6, pp. 2284–2294, Jun. 2013.

[4] T. T. Phuong, K. Ohishi, Y. Yokokura, and C. Mitsantisuk, "FPGA-based high-performance force control system with friction-free and noise-free force observation," *IEEE Trans. Ind. Electron.*, vol. 61, no. 2, pp. 994–1008, Feb. 2014.

[5] I. Liu, J. Reineke, and E. A. Lee, "A PRET architecture supporting concurrent programs with composable timing properties," in *Proc. Conf. Rec. 44th Asilomar Conf. Signals, Syst. Comput. (ASILOMAR)*, Pacific Grove, CA, USA, Nov. 2010 pp. 2111–2115.

[6] E. Dodiu, V. G. Gaitan, and A. Graur, "Custom designed CPU architecture based on a hardware scheduler and independent pipeline registers—Architecture description," in *Proc. IEEE 35th Jubilee Int. Conv. Inf. Commun. Technol., Electron. Microelectron.*, Opatija, Croatia, May 2012, pp. 859–864.

[7] E. Dodiu and V. G. Gaitan, "Custom designed CPU architecture based on a hardware scheduler and independent pipeline registers—Concept and theory of operation," in *Proc. IEEE Int. Conf. Electro-Inf. Technol. (EIT)*, Indianapolis, IN, USA, May 2012, pp. 1–5.

[8] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, and M. Imai, "Hardware implementation of a real-time operating system," in *Proc. 12th TRON Project Int. Symp.*, Tokyo, Japan, 1995, pp. 34–42.

[9] L. Lindh, "Fasthard—A fast time deterministic hardware based realtime kernel," in *Proc. 4th Euromicro Workshop Real-Time Syst.*, Athens, Greece, 1992, pp. 21–25.

[10] M. Vetromille, L. Ost, C. A. M. Marcon, C. Reif, and F. Hessel, "RTOS scheduler implementation in hardware and software for real time applications," in *Proc. 17th IEEE Int. Workshop Rapid Syst. Prototyping*, Chania, Crete, Jun. 2006, pp. 163–168.

[11] P. Kuacharoen, M. Shalan, and V. J. Mooney, III, "A configurable hardware scheduler for real-time systems," in *Proc. Eng. Reconfigurable Syst. Algorithms*, Brno, Czech Republic, 2003, pp. 95–101.

[12] P. Kohout, B. Ganesh, and B. Jacob, "Hardware support for real-time operating systems," in *Proc. 1st IEEE/ACM/IFIP Int. Conf. Hardw./Softw. Codesign Syst. Synthesis*, Newport Beach, CA, USA, Oct. 2003, pp. 45–51.

[13] G. Bloom, G. Parmer, B. Narahari, and R. Simha, "Real-time scheduling with hardware data structures," in *Proc. IEEE Real-Time Syst. Symp.*, Dec. 2010, pp. 1–4.

[14] H. Jamal and Z. A. Khan, "Hardware IP for scheduling of periodic tasks in multiprocessor systems," *WSEAS Trans. Comput. Res.*, vol. 3, no. 3, pp. 131–134, Mar. 2008.

[15] J. Tarrillo, L. Bolzani, and F. Vargas, "A hardware-scheduler for fault detection in RTOS-based embedded systems," in *Proc. 12th Euromicro Conf. Digital Syst. Design/Archit., Methods Tools*, Patras, Greece, Aug. 2009, pp. 341–347.

[16] S. Kelinman and J. Eykholt, "Interrupts as threads," *ACM SIGOPS Operating Syst. Rev.*, vol. 29, no. 2, pp. 21–26, Apr. 1995.

[17] L. Lindh, "Fastchart—A fast time deterministic CPU and hardware based real-time-kernel," in *Proc. Euromicro Workshop Real Time Syst.*, Paris-Orsay, France, Jun. 1991, pp. 36–40.

[18] J. Adomat, J. Furunas, L. Lindh, and J. Starner, "Real-time kernel in hardware RTU: A step towards deterministic and high-performance real-time systems," in *Proc. 8th Euromicro Workshop Real-Time Syst.*, L'Aquila, Italy, 1996, pp. 164–168.

[19] L. Lindh, T. Klevin, L. L. T. Klevin, and J. Furunäs, "Scalable architecture for real-time applications—SARA" in *Proc. 6th Int. Conf. Comput. Aided Design Comput. Graph. (CAD/CG)*, 1999, pp. 208–211.

[20] J. Lee, V. J. Mooney, III, A. Daleby, K. Ingstrom, T. Klevin, and L. Lindh, "A comparison of the RTU hardware RTOS with a hardware/software RTOS," in *Proc. Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Kitakyushu, Japan, 2003, pp. 683–688.

[21] V. J. Mooney, III and D. M. Blough, "A hardware-software real-time operating system framework for SoCs," *IEEE Des. Test. Comput.*, vol. 19, no. 6, pp. 44–51, Nov./Dec. 2002.

[22] S. Nordstrom, L. Lindh, L. Johansson, and T. Skoglund, "Application specific real-time microkernel in hardware," in *Proc. 14th IEEE-NPSS Real Time Conf.*, Stockholm, Sweden, Jun. 2005 pp. 333–337.

[23] S. Nordstrom and L. Asplund, "Configurable hardware/software support for single processor real-time kernels," in *Proc. Int. Symp. Syst.-Chip*, Tampere, Finland, Nov. 2007, pp. 1–4.

[24] J. Stankovic and K. Ramamritham, "The Spring kernel: A new paradigm for real-time systems," *IEEE Softw.*, vol. 8, no. 3, pp. 62–72, May 1991.

[25] W. Burleson *et al.*, "The spring scheduling coprocessor: A scheduling accelerator," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 7, no. 1, pp. 38–47, Mar. 1999.

[26] J. Hildebrandt, F. Golatowski, and D. Timmermann, "Scheduling coprocessor for enhanced least-laxity-first scheduling in hard real-time systems," in *Proc. 11th Euromicro Conf. Real-Time Syst. (EMRTS)*, York, U.K., 1999, pp. 208–215.

[27] J. Hildebrandt and D. Timmermann, "An FPGA based scheduling coprocessor for dynamic priority scheduling in hard real-time systems," in *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing* (Lecture Notes in Computer Science), vol. 1896, R. Hartenstein and H. Grünbacher, Eds. Berlin, Germany: Springer-Verlag, 2000, pp. 777–780, doi: 10.1007/3-540-44614-1_83.

[28] S. Chandra, F. Regazzoni, and M. Lajolo, "Hardware/software partitioning of operating systems: A behavioral synthesis approach," in *Proc. 16th ACM Great Lakes Symp. VLSI (GLSVLSI)*, New York, NY, USA, 2006, pp. 324–329.

[29] M. Song, S. H. Hong, and Y. Chung, "Reducing the overhead of real-time operating system through reconfigurable hardware," in *Proc. 10th Euromicro Conf. Digital Syst. Design Archit., Methods Tools, (DSD)*, Lubeck, 2007, pp. 311–316.

[30] A. S. R. Oliveira, L. Almeida, and A. B. Ferrari, "The ARPA-MT embedded SMT processor and its RTOS hardware accelerator," *IEEE Trans. Ind. Electron.*, vol. 59, no. 3, pp. 890–904, Mar. 2011.

[31] N. Silva, A. Oliveira, R. Santos, and L. Almeida, "The OReK real-time micro kernel for FPGA-based systems-on-chip," in *Proc. IEEE/ACM/IFIP Workshop Embedded Syst. Real-Time Multimedia*, Atlanta, GA, USA, Oct. 2008, pp. 75–80.

[32] N. Maruyama, T. Ishihara, and H. Yasuura, "An RTOS in hardware for energy efficient software-based TCP/IP processing," in *Proc. IEEE 8th Symp. Appl. Specific Processors (SASP)*, Anaheim, CA, USA, Jun. 2010, pp. 58–63.

[33] D. Andrews *et al.*, "hthreads: A hardware/software co-designed multithreaded RTOS kernel," in *Proc. 10th IEEE Conf. Emerg. Technol. Factory Autom.*, Catania, Italy, Sep. 2005, pp. 331–338.

Vasile Gheorghita Gaitan (M'08) received the M.S. and Ph.D. degrees from the Gheorghe Asachi Technical University of Iasi, Iasi, Romania, in 1984 and 1997, respectively.

He is currently a Professor with the Department of Computers, Electronics and Automation, Stefan cel Mare University of Suceava, Suceava, Romania. His current research interests include real-time scheduling, embedded middleware, digital systems design with FPGAs, fieldbuses, and embedded system application.

Prof. Gaitan is a member of the IEEE Computer Society.

Nicoleta Cristina Gaitan (M'14) received the M.S. and Ph.D. degrees in computer science from the Stefan cel Mare University of Suceava, Suceava, Romania, in 2008 and 2010, respectively.

She is currently a Lecturer with the Department of Computers, Electronics and Automation, Stefan cel Mare University of Suceava. Her current research interests include digital systems design with FPGAs, microprocessors and microcontrollers systems, fieldbuses, real-time systems, and distributed data acquisition systems.

Mrs. Gaitan is a member of the IEEE Computer Society.

Ioan Ungurean (M'09) received the M.S. and Ph.D. degrees in computer science from the Stefan cel Mare University of Suceava, Suceava, Romania, in 2008 and 2011, respectively.

He is currently a Lecturer with the Department of Computers, Electronics and Automation, Stefan cel Mare University of Suceava. His current research interests include fieldbuses, real-time systems, and distributed data acquisition systems.

Mr. Ungurean is a member of the IEEE Computer Society.