DE GRUYTER OPEN
RIGA TECHNICAL UNIVERSITY
Electrical, Control and Communication Engineering
ISSN 2255-9159 (online)
ISSN 2255-9140 (print)
July 2016, vol. 10, pp. 13–22
doi: 10.1515/ecce-2016-0002
https://www.degruyter.com/view/j/ecce

Improving the Performance of CPU Architectures by Reducing the Operating System Overhead (Extended Version)

Ionel Zagan (*Doctoral student, Stefan Cel Mare University of Suceava, Integrated Center for Research, Development and Innovation in Advanced Materials, Nanotechnologies, and Distributed Systems for Fabrication and Control (MANSiD)*),
Vasile Gheorghita Gaitan (*Professor, Stefan Cel Mare University of Suceava, Integrated Center for Research, Development and Innovation in Advanced Materials, Nanotechnologies, and Distributed Systems for Fabrication and Control (MANSiD)*)

***Abstract* – The predictable CPU architectures that run hard real-time tasks must be executed with isolation in order to provide a timing-analyzable execution for real-time systems. The major problems for real-time operating systems are determined by an excessive jitter, introduced mainly through task switching. This can alter deadline requirements, and, consequently, the predictability of hard real-time tasks. New requirements also arise for a real-time operating system used in mixed-criticality systems, when the executions of hard real-time applications require timing predictability. The present article discusses several solutions to improve the performance of CPU architectures and eventually overcome the Operating Systems overhead inconveniences. This paper focuses on the innovative CPU implementation named nMPRA-MT, designed for small real-time applications. This implementation uses the replication and remapping techniques for the program counter, general purpose registers and pipeline registers, enabling multiple threads to share a single pipeline assembly line. In order to increase predictability, the proposed architecture partially removes the hazard situation at the expense of larger execution latency per one instruction.**

***Keywords* – Jitter; Multithreading; Pipeline processing; Real-time systems; Scheduling.**

I. Introduction

Nowadays, specialized CPU architectures are among the most adopted solution for obtaining high performance, especially in embedded systems. Currently, a part of the available CPU implementations is not feasible to be used in mixed-criticality systems with hard real-time requirements. Such systems are really critical in terms of hard real-time execution, and the spatial and temporal isolation and timing predictability of tasks represent defining characteristics even in distributed systems.

In addition, real-time systems are used in all embedded applications within the economic and social areas, including public institutions [1]. We can say that there is no area without one or more microprocessors, and thus research in this field has increased, achieving considerable improvements and also providing Quality of Service (QoS) for hard real-time applications.

The current trend in mixed-criticality systems is represented by the predictable execution with hardware-based isolation of a large number of software tasks in different contexts, using limited hardware resources. Thus, in order to meet the appropriate deadlines, a single or multi-core processor must execute multiple types of tasks, according to their priorities in different situations. For this to be obtained, the field-programmable gate array (FPGA) devices with a high capacity in logic gates, available today at acceptable prices, represents a hardware support for the development of embedded real-time operating systems [2], [3].

In order to eliminate or reduce this inconvenience, in the last few years, studies have been carried out on software schedulers for mixed-criticality system, where software isolation was ensured by the real-time operating system (RTOS). Spatial isolation may be obtained by moving each task on a separate computational component. In this way, multithreaded processors use the hardware support in order to share a pipeline among more threads. With every cycle on the assembly line, fine-grained multithreaded processors alternate instructions from multiple threads, to the detriment of the long-latency created by context switch. Some fine-grained multithreaded processors could achieve isolation to the detriment of an immutable scheduling algorithm.

Achieving a hardware-based isolation using different multiprocessor solutions leads to a non-analyzable timing behavior and inefficient use of hardware resources. On the other hand, in mixed-criticality systems, the predictable behavior of all concurrent tasks can be obtained by scheduling each task on a distinct execution component, such as the cores in multi-core processors.

In order to obtain a competitive processor, we focus on the Multi Pipeline Register Architecture (nMPRA) [4], [5], as a RTOS developed in hardware, based on a Hardware Scheduler Engine (nHSE).

The proposed nMPRA-MT (Multi Pipeline Register Architecture – Fine-grained Multithreading) project fulfills the requirements for the time-bounded execution of parallel hard real-time tasks, being focused on multithreading execution of different types of threads. Although this implementation has reduced costs, the RTOS still has to be checked and validated.

©2016 Ionel Zagan, Vasile Gheorghita Gaitan.
This is an open access article licensed under the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0), in the manner agreed with De Gruyter Open.

This article is an extended version of the work published in [6]; in this paper, we provide a schedulability analysis of the already existing scheduling algorithms and a detailed description of the experimental results obtained during the tests performed on the nMPRA-MT CPU architecture.

This paper is structured as follows: the brief introduction in section I is followed by the related work in section II. Subsequently, section III presents the performance of several scheduling algorithms and section IV gives an overview on the nMPRA and nMPRA-MT architectures. The validation of the proposals including the experimental results achieved during the tests, is presented in section V, while section VI concludes the paper.

II. Related Work

This section presents a brief description of some single-core, multi-core architecture and scheduler implementations, regarding the development of real-time kernel primitives in hardware, focusing on reducing the operating system overhead.

We will start with the XMOS project. The processor presented by May in [7] has a 32 bit scalable architecture, and can therefore use the entire central processing unit, although there are less than four active execution threads. The new XMOS architecture allows systems designers to build interconnected multiple Xcore systems. Communication between Xcore processor cores from the same or from different chips is conducted through messages sent by point-to-point communication links. Cores interact with other external devices through integrated ports, ensuring the predictable execution of concurrent programs.

Therefore, XMOS architecture can be successfully used in multi-core systems, dedicated boards, or distributed systems.

The Merasa project [8], [9] was developed in order to obtain a processor architecture which can be successfully used in hard real-time embedded systems. The main characteristics of this project are the predictability of task execution and the efficient WCET (Worst Case Execution Time) analysis for each task. The MERASA project is better suited for mixed-criticality systems but is focused on the multi-core level. The proposed architecture is based on the SMT (Simultaneous Multithreading) technique, able to execute both hard real-time (HRT) and non real-time (NHRT) threads.

Each core is made up of two five-stage pipeline assembly lines. The first one is dedicated to one HRT execution thread, and the second one – to the NHRT execution threads. As seen in the example used by the authors, in a quad-core model, each core is composed of four hardware slots. The proposed architecture can therefore execute simultaneously one HRT thread and three NHRT threads. The HRT thread has the highest priority, as it is isolated by the real-time scheduler from other NHRT threads from the core. To reduce the task interferences to minimum, the authors propose the use of an AMC (analyzable real-time memory controller). In order to manage shared resources and critical sections belonging to the execution threads, the proposed architecture offers synchronization and inter-task communication mechanisms, such as spinlock, conditional variables, or barriers. To validate the architecture, the authors have calculated WCET for various configurations, using OTAWA and RapiTime benchmarks, based both on real parameters.

Zimmer et al. proposed a new research in the field of precision timed infrastructure. The FlexPRET project [10] is an innovative solution for mixed-criticality systems where time is a decisive factor of correctness. The basic idea of this concept is to ensure predictability and hardware isolation for hard real-time threads, while allowing the soft real-time threads to efficiently use the CPU, in order to increase the overall processor throughput (total number of instructions processed on all threads).

For these new processors called FlexPRET, a multitude of challenges appear regarding the multithreading techniques, single-core and multi-core architectures, scheduling algorithms, memory hierarchy, software components technologies, and programming languages.

The aim of the ARPRET project [11] is to ensure the development and verification of large safety-critical applications, providing thread-safe communication via shared memory access by projecting a particular soft-core coupled with a hardware accelerator.

The hthread project was presented for the first time by Andrews et al. in [12]. The hardware/software implementation of this multithreading architecture represents an innovative operating system for the embedded systems. The authors used Round-Robin or FIFO scheduling algorithm for its 256 active hardware threads, 256 active software threads, 64 binary spin-lock semaphores, and 64 common semaphores.

In [13], this structure is significantly improved. The scheduler is designed as a finite state machine (FSM), implemented in the hardware. In order to provide the necessary services for real-time requirements, the authors organize the operating system into four separate hardware cores: Thread Manager, Synchronization Manager, Scheduler and Condition Variables. Nevertheless, the architecture can schedule 256 active threads with an average delay of 1.9 μs and a jitter of 1.4 μs.

The disadvantage of this architecture is rendered by the jitter emerged from conducting communication among the cores through the system bus. These clock cycles are very important to a hard real-time architecture, even for the use of scheduling coprocessors or accelerators.

In [15], the authors proposed a new processor called Predator. This architectural approach characterized by predictability may be successfully used for embedded systems, even in real-time applications. The reason is that the Predator project uses the simple cores with predictable behavior, analyzable caches, compiler-controlled memory management and predictable kernel mechanisms. In order to obtain a real-time multi-core architecture, the authors use crossbars to implement communication among cores, shared cache, and memory. Thus, allocating the code and data of different threads to a shared memory, the architecture needs an accurate scheduler and memory controller, in order to ensure exclusive access and noninterference.

El-Haj-Mahmoud et al. in [16] have proposed an architecture which can be divided into a set of virtual processors. The execution times of these processors are independent of each other, providing the tasks executed on virtual processors with a composable time [17]. The architecture proposed by the authors presents its partitioning either into a few higher-performance processors, more low-performance processors, or a combination of the two extremes.

The data in Table I represents the main characteristics of the most representative implementations described in the present section, as well as the differences between them. The issues involved refer to: coprocessor, replication of resources (program counter, register file or pipeline registers), scheduler implementation, assembly line (number of stages for every pipeline), synchronization and communication mechanisms, type of scheduler, and implementation. These processor architectures are generally scalable, depending on the FPGA characteristics used and on the type of the implemented processor. Hardware execution of specialized processors, coprocessors or schedulers is a novelty and a challenge in the area of real-time systems.

III. Periodic and Aperiodic Task Scheduling for Real-Time Systems

The present section will present various algorithms for real-time periodic and aperiodic task scheduling. Taking into account the restrictions for each set of tasks, each algorithm represents a scheduling solution. When rigorous scheduling restrictions are not applied, the complexity of implementation can be reduced by using basic algorithms. Thus, although the resulting scheduling scheme is not an optimal solution, the feasibility of the system is ensured for a wide range of situations [1].

A. Aperiodic Task Scheduling

The algorithms described in this section can be used for scheduling aperiodic tasks running on single-processor systems; they can also be applied to multiprocessor systems or distributed architectures with complex tasks.

In non-preemptive scheduling, a CPU runs until completion a task that has entered execution. In this case, all necessary operations are performed in order to complete the current task. Therefore, the executed task will transfer control to the scheduler only when it completes his execution, even if there are tasks with a higher priority ready for execution. Therefore, the impossibility to guarantee the determinism of starting the task execution represents the disadvantage of non-preemptive scheduling. This type of scheduler is not preferred by commercial real-time schedulers because the control transfer towards the scheduler is not deterministic.

An example in this sense is Bratley's algorithm, proposed by Bratley et al. in 1971 [18]. This implementation was proposed in order to find a scheduling scheme for a certain number of non-preemptive and aperiodic tasks.

The Spring non-preemptive algorithm was first adopted in the hard real-time kernel called Spring kernel, designed by Stankovic and Ramamritham [19]. The kernel has been implemented for critical control applications in dynamic systems, the objective of the algorithm being to find a feasible scheduling for a set of tasks with precedence constraints, shared resources, aperiodic arrival, and non-preemptive properties.

In certain applications, the scheduling of a set of tasks cannot be performed randomly, because the compliance of some precedence relationships defined at the design stage is mandatory.

TABLE I

The Main Characteristics of the RTOS Presented in Section II

RTOS	Copro-cessor	Scheduler implementation	Resource replication	Pipeline	Synchronization and communication mechanisms	Scheduling algorithm	Implementation type
Hthread [12], [13]	Yes	HW (individual RTOS core)	No	No	Yes (implemented in hardware)	Static (FIFO, round-robin, priority based)	Single-core
FASTCHART [33], [34]	No	HW	No	No	Yes (introduced the next version [34])	Static (rate monotonic)	Single-core
nMPRA [5]	No	HW	Yes (general and pipeline registers)	5-stage pipeline	Yes (implemented in hardware)	Static (round-robin, priority based) and dynamic (SW)	Single-core
PRET [32]	No	HW	No	5-stage pipeline	Yes	Static (round-robin)	Single-core
FlexPRET [10]	No	HW	No	5-stage pipeline	Yes (PRET-C)	Static and dynamic (earliest deadline first, rate-monotonic)	Single-core
JOP [35] and JOP-Plus [36]	No	SW	No	3-stage pipeline	Yes	Dynamic	Single-core
Merasa [8], [9]	No	HW + SW scheduler for optimization	Yes (general registers)	Two 5-stage pipeline / core	Yes (single-core and multi-core)	Dynamic	Multi-core
XMOS [7]	No	HW	Yes (general registers)	4-stage pipeline/ core	Yes (single-core and multi-core)	Dynamic	Multi-core
Komodo [14]	No	SW	Yes	4-stage pipeline	No	Dynamic	Single-core

These precedence relationships are usually described through acyclic directed graphs, where the tasks are represented through nodes and the precedence relationships through arrows. The precedence graphs introduce a partial order for the set of tasks subject to scheduling. This scheduling method was used in two implementations using the following precedence constraints: Latest Deadline First (LDF), and Earliest Deadline First (EDF). The LDF algorithm was presented by Lawler [20] in 1973, and it can be applied on a set of aperiodic tasks with simultaneous arrival and a precedence relationship.

Chetto et al. [21] presented an algorithm for scheduling a set of aperiodic tasks with precedence constraints and dynamic activation. This implementation is achievable provided that the tasks are preemptive and by transforming the set of dependent tasks in a set of independent ones by modifying the time parameters. If these steps have been completed, aperiodic tasks can be scheduled further using the EDF algorithm.

The Earliest Due Date algorithm (EDD) was presented by Jackson in 1955 [22]. This algorithm schedules a set of aperiodic tasks on a single core, minimizing the maximum delays. Tasks have a synchronized occurrence, different deadlines and periods of execution; they are also independent, without a precedence relation and shared resources. The complexity of the EDD algorithm to achieve optimal scheduling is rendered by the procedure of sorting tasks in the ascending order of the deadline. The scheduling of this algorithm ensures that, in the worst case, all tasks complete execution before the deadline.

The Earliest Deadline First algorithm, proposed by Horn in 1974 [23], is a solution for scheduling an independent set of preemptive and aperiodic tasks executed on a single core system.

In the case aperiodic tasks are not synchronized (these tasks can be dynamically activated during execution), preemptivety becomes an important factor. In general, the issues raised by preemptive schedulers are simpler than those raised by the non-preemptive ones.

In the case of non-preemptive schedulers, the emergence of a new task ready for execution will not interrupt the task being executed in order to meet its deadline. When preemptivety is possible, any task can enter execution if its deadline is lower than that of the one being executed.

For exemplification, we propose in Table II a set of five tasks with their relative parameters, where a_i is the arrival time of τ_i. Each task τ_i is characterized by a WCET noted with C_i, a deadline D_i, and period T_i. A deadline model is defined, compelling a D_i smaller or equal to T_i.

Fig. 1 shows an example of scheduling a set of five tasks using the EDF algorithm. At moment $t = 0$, task τ_1 enters execution, and at moment $t = 1$, task τ_2 cannot interrupt τ_1 because $D_1 < D_2$. Task τ_1 completes execution at time moment $t = 2$, and at moment $t = 4$, when τ_2 is being executed, task τ_3 interrupts τ_2 because $D_3 < D_2$.

To be noted that at time moment $t = 7$, task τ_4 does not interrupt τ_3 because $D_3 < D_4$.

TABLE II

THE PARAMETERS OF A SET OF FIVE TASKS

	a_i	C_i	D_i
τ_5	12	5	18
τ_4	7	8	24
τ_3	4	4	10
τ_2	1	3	11
τ_1	0	2	4

When τ_3 completes execution, the CPU is assigned to task τ_2. Task τ_4 is executed at moment $t = 9$, but it is interrupted at $t = 12$ by τ_5, because the last one has a lower deadline. Task τ_4 re-enters in execution at moment $t = 17$, when τ_5 completes its own.

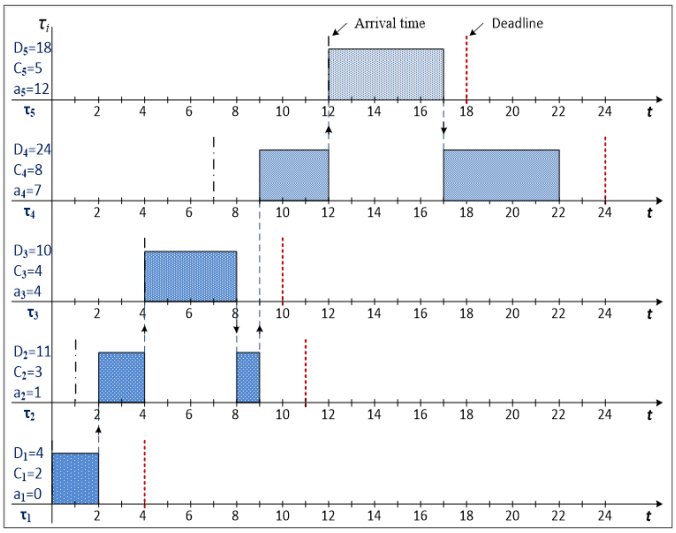

Fig. 1. A scheduling example using the EDF algorithm.

B. Periodic Task Scheduling

In most real-time applications, periodic activities are the system's major computing necessity. Periodic tasks come from the control loops, system monitoring, or sensory data acquisition. These activities need to be performed cyclically with a certain rate specified by the application requirements.

For a control application with a set of competing periodic tasks, each having different time constraints, the real time operating system must guarantee that every periodic instance is regularly activated and completed until the limit imposed by the deadline [1].

The basic algorithms used in periodic task scheduling are the following:

- Timeline Scheduling;
- Rate Monotonic;
- Deadline Monotonic;
- Earliest Deadline First.

The Timeline Scheduling algorithm (TS) is one of the most common approaches for scheduling periodic tasks in the control of traffic and military systems [1]. The algorithm is called Cyclic Executive, and it consists in dividing the time axis in equal intervals during which one or more tasks can be scheduled for execution.

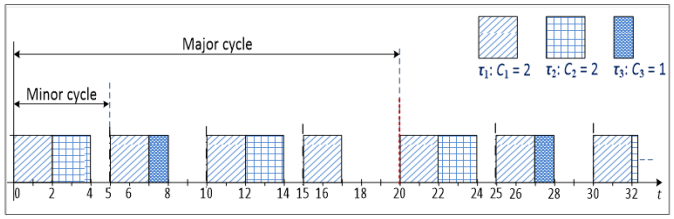

Fig. 2. An example of scheduling three tasks using the TS algorithm.

In order to guarantee the individual scheduling frequency for every task, a timer synchronizes their activation at the beginning of each time frame.

Fig. 2 shows an example of how this scheduling algorithm is used.

Three tasks are considered: τ_1, τ_2, and τ_3; these tasks have to be executed at a time frame of $T_1 = 5$ ms, $T_2 = 10$ ms, and $T_3 = 20$ ms. A possible scheduling for these tasks is represented in Fig. 2. In this case, it is easy to verify that the optimal time length is 5 ms; this period is called Greatest Common Divizor of execution periods. Therefore, task τ_1 must be executed at each interval, task τ_2 at every two intervals, and task τ_3 at every four intervals. The duration of these intervals is called Minor Cycle, while the time frame in which the scheduling of all tasks is repeated is called Major Cycle. In order to ensure the feasibility of the scheduling scheme, it suffices to know the WCET for each task and to check the fact that the sum of execution periods from each interval is less than or equal to the Minor Cycle. Therefore, for the aforementioned example, the following relations must be satisfied: $C_1 + C_2 \leq 5$ ms, and $C_1 + C_3 \leq 5$ ms.

The major advantage of the TS algorithm is its simplicity. The method can be implemented by scheduling an interrupt at a time frame equal to the Minor Cycle, by writing the main program for calling tasks in the order given by the Major Cycle, and by inserting a synchronization point at the beginning of each Minor Cycle. The major disadvantage of this algorithm is that scheduling is fragile at overload.

The scheduling Rate Monotonic (RM) algorithm is based on a simple rule that attaches priorities to tasks, depending on the execution rates. Therefore, the tasks with a higher execution rate will have higher priorities P_i; therefore, RM becomes an intrinsic preemptive algorithm, because the task being executed can be interrupted by the occurrence of a task with a higher priority. If the execution rates are constant, the algorithm assigns fixed priorities P_i before the execution of the tasks that are not changed over time [1].

Fig. 3.a shows that the time response of task τ_k is delayed by the occurrence of task τ_i with a higher priority. Fig. 3.b shows how task τ_i delays even further the execution of task τ_k, the response time being constantly influenced by the number of interrupts generated by task τ_i.

Considering a set of tasks scheduled with RM algorithm, the worst response time of a task τ_i is that when all tasks with higher priority are executed simultaneously. Liu and Layland in [24] proved that, for a set of tasks with unique execution periods, there is a feasible scheduling before the deadline, provided that the CPU usage is less than a certain limit.

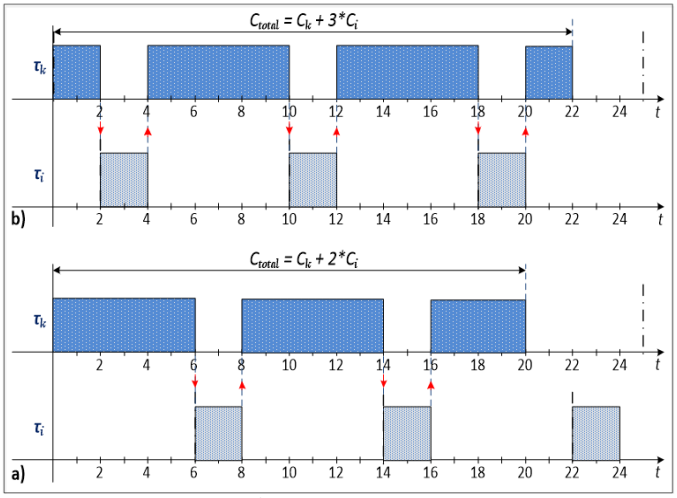

Fig. 3. An example for scheduling using the RM algorithm.

The advantage of this algorithm is that the scheduling scheme can be easily checked at all critical instants.

According to the Deadline Monotonic (DM) algorithm, each task has attached a fixed priority P_i inversely proportional with its relative deadline D_i. Therefore, the task with the lowest deadline is scheduled at any moment of execution, and if the relative deadlines are constant, the DM algorithm is one with constant priorities.

This algorithm was proposed in 1982 by Leung and Whitehead [25] as an extension for the RM algorithm. The feasibility test using the DM algorithm can be performed using the following formula:

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{1/n} - 1). \quad (1)$$

The DM and RM algorithms are used especially in full-preemptive scheduling models, because the executed task can be interrupted by the occurrence of another task with a relatively lower deadline or with a higher execution rate.

The EDF algorithm is a dynamic scheduling method that selects tasks according to their absolute deadline. Higher priorities will be dynamically assigned to tasks with a closer deadline. Moreover, the algorithm is executed preemptively so that the execution of a task can be interrupted by the occurrence of one with a lower deadline. The EDF algorithm does not refer to the frequency of tasks, so it can be used for scheduling both periodic and aperiodic tasks. Checking the scheduling of a set of periodic tasks using the EDF algorithm can be performed through the CPU usage factor (2). This is stated in the following theorem [26]:

Theorem 1: A periodic set of n task can be scheduled with the EDF algorithm only if:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1. \quad (2)$$

The EDF algorithm implies the fact that a task can be interrupted only once in the same interval T_i. Therefore, the

small number of interrupts is the result of dynamically assigning the priorities P_i.

IV. Overview of the nMPRA and nMPRA-MT Architecture

The nMPRA architecture proposed by Gaitan et al. in [5] has been specially designed to reduce the scheduler overhead and the switch time of the task context, with the purpose to minimize the unacceptable jitter present in the current RTOS.

A. nMPRA Architecture

nMPRA is a hardware design that represents a custom CPU architecture based on replication of resources, such as program counter, general purpose registers, and pipeline registers.

As it can be seen in Fig. 4, the authors use a register file and a set of four pipeline registers for each task, in order to hold the individual running state information. All the tasks running on the CPU use the same data path, control unit, ALU, Hazard Detection Unit, and Forward Unit.

The pipeline registers used by nMPRA architecture are the following: IF/ID (instruction fetch/instruction decode), ID/EX (instruction decode/execute), EX/MEM (execute/memory), MEM/WB (memory/write back), and also PC (Program Counter) which is not a pipeline register, but it is managed by the nHSE in the same manner.

This implementation allows a very fast context switching, which is possible due to the remapping of the active running task context with the scheduled task; the jitter is minimized in order to provide an accurate predictability behavior. In other words, the nMPRA architecture replaces the classical stack-saving methods with a remapping technique, allowing us to execute a new task in an average of one clock cycle and maximum three in the case of working with memory instructions.

The original design is based on a traditional MIPS architecture that was specially modified to support instructions dedicated to the hardware scheduler, part of the CPU itself.

The nHSE is task-oriented, in order to increase the throughput of execution and to avoid the excessive use of resources. The entire nHSE is disabled when the processor is connected to a power supply, the only one active being the high priority HT0.

In order to prove the performance of the new processor concept, we used our assembler translator. This component proves useful in validating the opcode of the new instructions added to control the nHSE.

B. Proposed nMPRA-MT Architecture

The nMPRA-MT architecture presented in [27] is a fine-grained multithreaded processor based on the original nMPRA concept, designed to support architectural requirements for hard real-time systems.

The development of a new application has been imposed by 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 easily used to develop real-time applications.

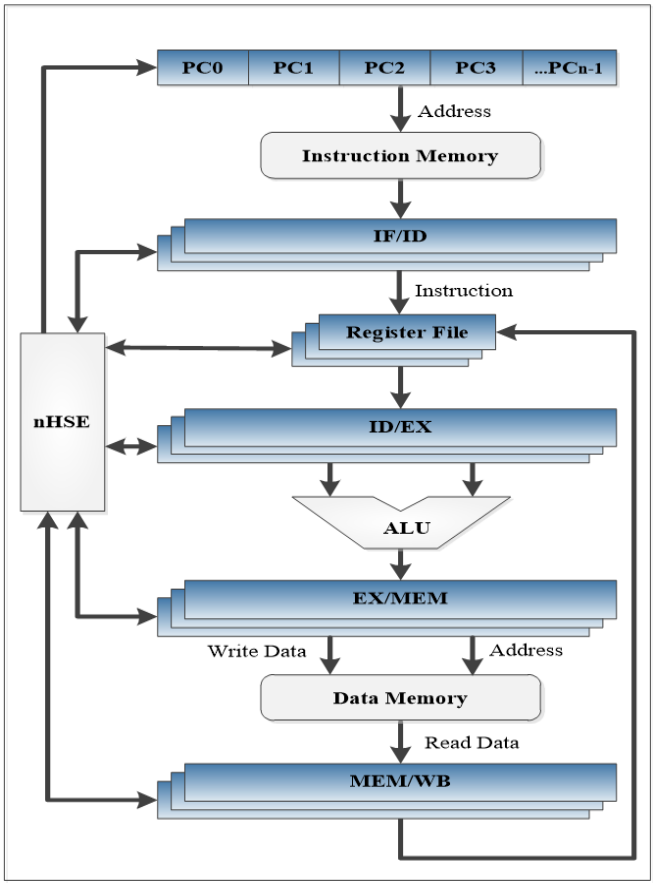

Fig. 4. nMPRA CPU architecture.

The purpose of this paper is to describe and present the implementation results of a predictable scheduler that controls different types of tasks interlacing with the pipeline levels. In order to do this, hard real-time tasks are classified and become Hard Threads-HT, whereas low priority tasks become Soft Threads-ST. Thus, HTs represent the tasks where a missing deadline generates critical effects, and STs represent useful tools for completing the system (we will use the terms "thread" and "task" interchangeably). Using these notations, the implementation of the nMPRA-MT processor is designed to support hardware-based isolation for HTs, at the same time allowing STs to use the unallocated cycles.

The nMPRA-MT is a multithreaded processor based on a hardware scheduler and independent pipeline registers, designed to support architectural requirements for hard real-time systems. The main reason for implementing the nMPRA-MT project is the decrease of execution overhead given by the scheduling and context switching operation.

Although, nMPRA-MT is an architecture that involves multiplication of resources, its pipeline modification improves the related costs that are more effective than those of other commercial CPU architectures using resource replication. Such CPU implementations are recommended for a reasonable number of tasks designed for small hard real-time applications. For a large number of tasks, due to a synthesis of logic with unreasonably high propagation times, the frequency will significantly decrease.

Every thread has its own ID and STATE registers; the thread with the highest priority has the ID equal to 0 and the lowest priority corresponds to $n - 1$.

In order to maintain the performance of pipeline processing, the authors use a five-stage assembly line to allow the execution of multiple instructions from different threads in the pipeline levels. Because nMPRA-MT architecture uses CPU working registers and resource remapping techniques for the pipeline registers, nHSE interleave different threads into the pipeline assembly line, without losing clock cycles due to contexts switching operation.

At the expense of two cycle clock latencies per one instruction from the same HT thread, when the pipeline is full, the efficiency is equal to one instruction for every clock cycle. When data hazard situations are detected, a new forwarding unit is implemented in order to solve data dependencies within the same thread. When a HT thread is scheduled to be executed every two clock cycles, it is no longer possible to stall an instruction already fetched if it is dependent on data hazards. This turns the nMPRA-MT project into a predictable architecture, designed to compute faster; only in exceptional cases, instructions from the pipeline are flushed.

Concerning the interrupt system, the nMPRA-MT architecture preserves the algorithm used by nMPRA. Thus, an interrupt could be assigned to one task only, HT or ST, inheriting its priority and behavior. This interrupt system is completely allocated, so that an important advantage is represented by the fact that interrupts do not affect the pipeline assembly line. By doing this, the proposed nMPRA-MT is able to manage periodic or aperiodic events, such as a time event, watchdog, or deadline events.

nHSE has been designed to implement dynamic scheduling algorithms for HT and ST threads, in order to allow certain ST threads to execute multiple tasks. By doing this, every thread has its own ID and STATE registers. Therefore, the ID register indicates the priority and the type of threads that can be HT or ST. The STATE register memorizes the state of each thread, which can either be active, idle or sleeping. The ID register identifies the thread when an event appears, attached by the scheduler. When HT threads are in the sleeping state, the STs are scheduled to consume the available processor cycles.

C. Pipeline and Thread Management

This paper extends the basic idea presented in [4] and [5], proposing an original implementation based on the nHSE concept.

The *PC_IF_i* signal connected to the simple 32-bit adder is used to provide the program counter (*PC_IF_0*, *PC_IF_1*, …, *PC_IF_n−1*) available in the next clock cycles.

nHSE supports an arbitrary interleaving of threads by using a new innovative Forward Unit. In order to prevent the stalls, the data or control hazards that occur when the HTs are scheduled every two clock cycles are treated by the new Hazard Detection Unit described in subsection *E*.

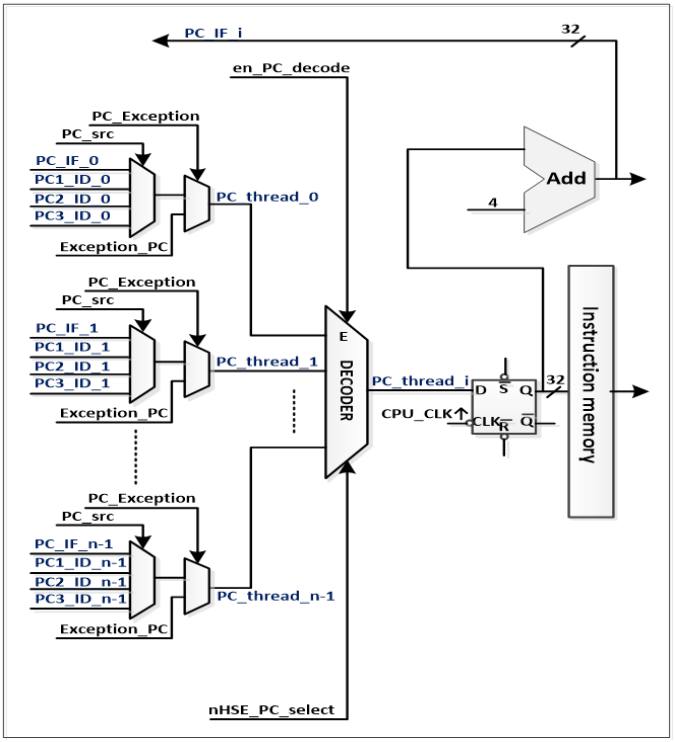

Fig. 5. Hardware for selecting PC corresponding to the scheduled thread.

Due to the fact that each thread has its own program counter, a set of pipeline registers and working registers for general purpose, the next program counter corresponding to the scheduled thread is selected by using *en_PC_decode* and *nHSE_PC_select* signals, as shown in Fig. 5.

However, it is difficult to predict how many cycles it would require for HT threads to be executed, because pipeline spacing between them can vary unpredictably.

D. Events and Resource Management

The model for interrupt handling proposed in this paper is similar to the solution presented in [28]. This is based on the unification of threads and interrupts into a single model; the interrupts are converted into threads using a limited overhead.

This model allows the implementation of periodic, aperiodic or sporadic events related to CPU operation, eliminating the interrupts of the tasks [29]. The interrupts are treated as events attached to HTs or STs, and therefore inheriting their priority. By using this approach, a task can be suspended only by the interrupts that are attached to higher priority tasks; the behavior of the system is more predictable in the context of a small real-time application.

As shown in Fig. 6, there are three types of events: periodic time events, watchdog timer, and two deadline events that are equal to an alarm and/or a fault [30].

E. Exceptions and Hazard Situations

The innovative Hazard Detection Unit and Forward Unit allow the fine-grained pipelined processor to operate efficiently and correctly in the presence of data, structural, and control hazards in various situations.

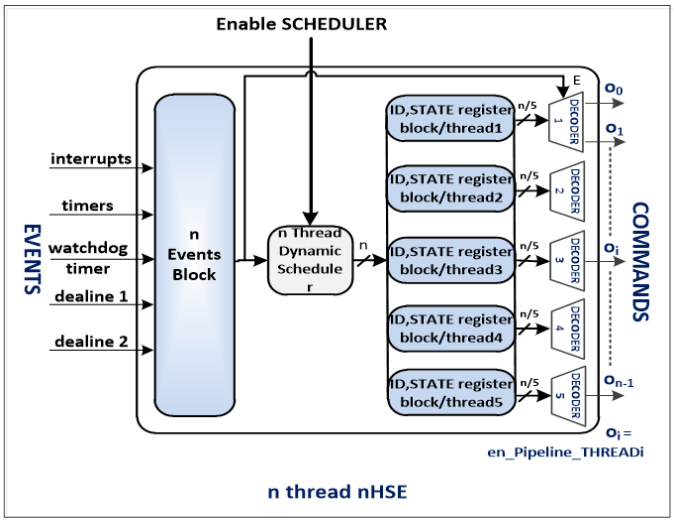

Fig. 6. Architecture of hardware scheduler engine with enable oi signals and interrupt handling model [30].

Data redirecting units play a significant role in improving system performance. We have already mentioned in the previous sections that there are various hazard situations when the data consumed by an instruction is not yet produced by the previous instruction.

When a different number of threads is active on the pipeline assembly line, Hazard Unit detects that the required data is still in the pipeline and whether that data may be forwarded by one of the Forward Units presented in Table III.

For each situation, depending on the number of HT and ST threads, the nMPRA-MT uses the appropriate Forward Unit in order to meet deadlines; therefore, the latency effect is reduced to a minimum. If the scheduler fetches instructions from the same HT thread every two clock cycles, the worst cases of assembly line stalling are avoided, at the expense of a wider execution latency.

As shown in Table III, by way of example, we consider two HTs that are scheduled simultaneously at the expense of two clock cycles latency per one instruction from the same thread. In this case, there will be no wasted clock cycles due to unsolved hazard situations, and the UFW2 Forward Unit does not affect system predictability.

When nHSE fetches continuously and unpredictably the instructions from a scheduled ST thread, using UFW1 Forward Unit, it is possible to stall the pipeline assembly line. In another case, when the instructions executed in the pipeline belong to four different HT or ST threads, no hazard situations are possible and NO FW Forward Unit is used.

TABLE III

HAZARD DETECTION AND FORWARD UNIT CONFIGURATION

Active threads in the pipeline	HT	Latency (HT)	ST	Latency (ST)	Forward Unit
1	0	0	1	1	UFW1
2	2	2	0	0	UFW2
2	1	2	1	2	UFW3
3	1	2	2	4	UFW4
2	0	0	2	2	UFW2
4	2	4	2	4	NO FW
4	4	4	0	0	NO FW
4	0	0	4	4	NO FW

V. VALIDATION OF THE NMPRA-MT ARCHITECTURE

In this section, we focus on the validation of the presented concept, introducing the experimental results of the implemented FPGA prototype.

The project has been tested on a Virtex-6 FPGA ML605 Evaluation Kit from Xilinx, and the code of the processor was developed in standard Verilog. Nevertheless, in the testing and validation procedure, we took into account the influence of the signal propagation time on the number of independent sets of pipeline registers and register files [31]. Every thread uses a PC register, a register file, and a set of pipeline registers, while the replication of these resources for eight active threads requires 8.64 kB of RAM.

A. The Impact of Different Configuration Models on FPGA Resources

In order to evaluate the area cost associated to different requirement models, several nMPRA and nMPRA-MT configurations were validated on a Xilinx FPGA. The register file block has a fixed size, computed at the level of compilation. This is large enough for six nesting levels, so that at a given time, different Register Files can be accessed, according to the nHSE configuration.

Between the original nMPRA project presented in [5] and the proposed nMPRA-MT implementation, the cost increase is by 6 % in LUTs (lookup-table) and 35 % in FFs (flip-flop). This is caused by the fine-grained multithreading, nHSE, Hazard Detection Unit, and Forward Unit. Although the nMPRA-MT removes the stalls from the pipeline levels, the architecture logic requires more multiplexing based on ID and STATE thread registers which must be stored for each thread.

Fig. 7 shows the resource differences between the original nMPRA implementation presented in [5] and the nMPRA-MT implementation.

The implementation proposed in the present paper is a deterministic architecture, as compared to SMT processors, which may expose additional overcontrol, if the program does not expose an ILP (Instruction Level Parallelism). Taking into consideration these data, we can state that the amount of memory needed for the implementation of the nMPRA-MT processor is more than acceptable, provided that the total number of tasks may be 8, 16, or 32.

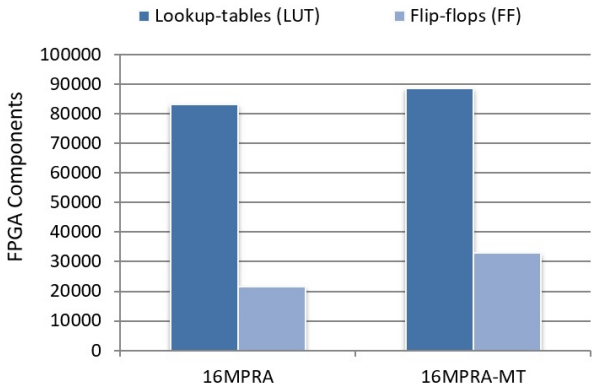

Fig. 7. FPGA resource usage corresponding to the nMPRA and nMPRA-MT processor configurations, with 16 threads each.

B. Experimental Results

As it is seen in Fig. 8, the clock on channel 1 is used for the synchronization of memories and pipeline registers, while the clock on channel 2 is used for the synchronization of the nHSE scheduler. Channel 4 marks the answer of the application to an asynchronous external event (channel 3 signal), resulted from pushing a button from the ML605 board.

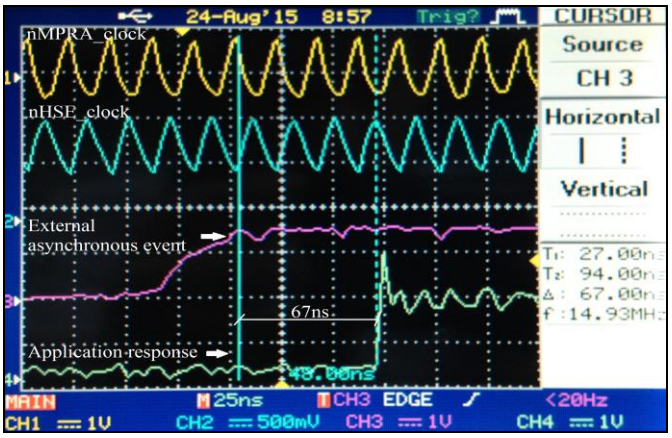

Fig. 8. Application response in relation with an external asynchronous event when the program executes *sw* instructions.

For the case in which the program executes special instructions used for external memory accesses at an operating frequency of 50 MHz, the scheduler response to an asynchronous external event can reach 67 ns, depending on the occurrence time of the event. This working frequency is not the maximum operating frequency of the processor. In order to monitor the clock, for the asynchronous interrupt signal and scheduler answer, we used unregistered ports.

C. WCET Analysis

In order to improve WCET analysis, specific algorithms have been enhanced and developed to ensure the nMPRA-MT features, including a fine-grained pipeline implementation, a particular threaded scheduler, and an original Forward Unit.

Being a fine-grained multithreading implementation, the nMPRA-MT scheduler had to be able to execute simultaneously instructions from different threads. When the pipeline assembly line is occupied by two concurrent HTs, the execution report is finally an instruction for two clock cycles per thread. This means that, at any given time, there may be multiple instructions in the pipeline levels, without encountering unsolved hazard situations. Therefore, to guarantee hardware-based isolation and timing predictability, a constant scheduling frequency is required for HTs. For mixed-criticality systems, the upper bound is statically guaranteed at the compile time. The WCET analysis techniques are applied in order to confirm the safe upper bounds of HTs, but they are not necessarily required for STs.

VI. Conclusion and Future Work

The proposed nMPRA-MT is an innovative project mainly because it provides a minimum time switch between the tasks that are normally accomplished in a single machine cycle.

The response of the system to an external asynchronous event will not exceed 1.5 clock cycles or a maximum of three clock cycles when the CPU accesses the global memory, as seen in Fig. 8. The main reasons for implementing the nMPRA-MT project are: the reduction of memory footprint, the jitter reduction, and the improvement of the response time for external events.

In conclusion, we can say that the use of nMPRA-MT architecture with 16 tasks is fully justified by the benefits it brings; moreover, the implementation performance/cost indicator is very good. The resource differences between the original nMPRA implementation presented in [5] and the nMPRA-MT implementation may increase.

As future work, we will present the experimental WCET analysis obtained using complete benchmarks, considering a feasible set of tasks executed on the nMPRA-MT architecture, which guarantees the predictability and hardware-based isolation for HTs.

However, the processor is opened to new improvements such as:

- implementation in hardware of a memory controller for an explicit model of memory hierarchy;
- computing of the WCET using complete benchmarks.

Acknowledgment

This work was partially supported from the project "Integrated Center for research, development and innovation in Advanced Materials, Nanotechnologies, and Distributed Systems for fabrication and control", Contract No. 671/09.04.2015, Sectoral Operational Program for Increase of the Economic Competitiveness co-funded from the European Regional Development Fund.

This paper has been prepared with the financial support of the project "Quality European Doctorate – EURODOC", Contract No. POSDRU/187/1.5/S/155450, project co-financed by the European Social Fund through the Sectoral Operational Programme "Human Resources Development" 2007–2013.

References

[1] G. C. Buttazzo, *Hard Real-Time Computing Systems – Predictable Scheduling Algorithms and Applications* (Real-Time Systems Series 24). 3rd ed., Springer US, 2011. ISBN 978-1-4614-0675-4. https://doi.org/10.1007/978-1-4614-0676-1

[2] B. Kumthekar, L. Benini, E. Macii and F. Somenzi, "Power optimisation of FPGA-based designs without rewiring," in *IEE Proc. – Comput. and Digital Techniques*, vol. 147, no. 3, pp. 167–174, May 2000. https://doi.org/10.1049/ip-cdt:20000497

[3] 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. on Industrial Electronics*, vol. 60, no. 8, pp. 3360–3371, Aug. 2013. https://doi.org/10.1109/TIE.2012.2200214

[4] 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 *2012 IEEE Int. Conf. on Electro/Information Technology*, Indianapolis, IN, USA, May 2012, pp. 1–5. https://doi.org/10.1109/EIT.2012.6220705

[5] V. G. Gaitan, N. C. Gaitan and I. Ungurean, "CPU Architecture Based on a Hardware Scheduler and Independent Pipeline Registers," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 9, pp. 1661–1674, Sep. 2015. https://doi.org/10.1109/TVLSI.2014.2346542

[6] I. Zagan, "Improving the performance of CPU architectures by reducing the Operating System overhead," in *2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, Riga, Nov. 2015, pp. 1–6. https://doi.org/10.1109/AIEEE.2015.7367279

[7] D. May, "The XMOS Architecture and XS1 Chips," *IEEE Micro*, vol. 32, no. 6, pp. 28–37, Nov.–Dec. 2012. https://doi.org/10.1109/MM.2012.87

[8] T. Ungerer et al., "Merasa: Multicore execution of hard real-time applications supporting analyzability," *IEEE Micro*, vol. 30, no. 5, pp. 66–75, 2010. https://doi.org/10.1109/MM.2010.78

[9] J. Wolf, M. Gerdes, F. Kluge, S. Uhrig, J. Mische, S. Metzlaff, C. Rochange, H. Cassé, P. Sainrat and T. Ungerer, "RTOS Support for Parallel Execution of Hard Real-Time Applications on the MERASA Multi-core Processor," in *2010 13th IEEE Int. Symp. on Object/Component/Service-Oriented Real-Time Distributed Computing*, Carmona, Seville, May 2010, pp. 193–201. https://doi.org/10.1109/ISORC.2010.31

[10] M. Zimmer, D. Broman, C. Shaver and E. A. Lee, "FlexPRET: A processor platform for mixed-criticality systems," in *2014 IEEE 20th Real-Time and Embedded Technology and Applicat. Symp. (RTAS)*, Berlin, 2014, pp. 101–110. https://doi.org/10.1109/RTAS.2014.6925994

[11] S. Andalam, "Predictable platforms for safety-critical embedded systems," Thesis, The University of Auckland, 2013.

[12] D. Andrews et al., "hthreads: A hardware/software co-designed multithreaded RTOS kernel," in *2005 10th IEEE Conference on Emerging Technol. and Factory Autom*., Catania, Italy, Sep. 2005, pp. 331–338. https://doi.org/10.1109/ETFA.2005.1612697

[13] J. Agron, D. Andrews, "Hardware Microkernels for Heterogeneous Manycore Systems," in *2009 Int. Conf. on Parallel Processing Workshops* (*ICPPW '09*), Vienna, 2009, pp. 19–26. https://doi.org/10.1109/ICPPW.2009.21

[14] J. Kreuzinger, R. Marston, T. Ungerer, U. Brinkschulte and C. Krakowski, "The Komodo project: thread-based event handling supported by a multithreaded Java microcontroller," in *Proc. 25th EUROMICRO Conf. Informatics: Theory and Practice for the New Millennium*, Milan, 1999, vol. 2, pp. 122–128. https://doi.org/10.1109/EURMIC.1999.794770

[15] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister and C. Ferdinand, "Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 7, pp. 966–978, July 2009. https://doi.org/10.1109/TCAD.2009.2013287

[16] A. El-Haj-Mahmoud, A. S. Al-Zawawi, A. Anantaraman, and E. Rotenberg, "Virtual multiprocessor: an analyzable, highperformance architecture for real-time computing," in *Proc. of the 2005 int. conf. on Compilers, architectures and synthesis for embedded systems, CASES '05.* San Francisco, 2005, pp. 213–224. https://doi.org/10.1145/1086297.1086326

[17] A. El-Haj-Mahmoud and E. Rotenberg, "Safely Exploiting Multithreaded Processors to Tolerate Memory Latency in Real-Time Systems," in *Proc. of the 2004 int. conf. on Compilers, architecture, and synthesis for embedded systems*, Washington, 2004, pp. 2–13. https://doi.org/10.1145/1023833.1023837

[18] P. Bratley, M. Florian, and P. Robillard, "Scheduling with earliest start and due date constraints," *Naval Research Quarterly*, vol. 18, no. 4, 1971. https://doi.org/10.1002/nav.3800180410

[19] J. Stankovic and K. Ramamritham, "The design of the spring kernel," in *Proc. of the IEEE Real-Time Systems Symp.*, Dec. 1987.

[20] E. L. Lawler, "Optimal sequencing of a single machine subject to precedence constraints," *Managements Science*, vol. 19, no. 5, pp. 544–546, 1973. https://doi.org/10.1287/mnsc.19.5.544

[21] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic scheduling of realtime tasks under precedence constraints," *J. of Real-Time Systems*, vol. 2, no. 3, pp. 181–194, Sep. 1990. https://doi.org/10.1007/BF00365326

[22] J. R. Jackson, "Scheduling a production line to minimize maximum tardiness," *Management Science Research*, vol. 43, 1955.

[23] W. Horn, "Some simple scheduling algorithms," *Naval Research Logistics Quarterly*, vol. 21, no. 1, Mar. 1974. https://doi.org/10.1002/nav.3800210113

[24] C. L. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," Journal of the ACM (JACM), vol. 20, no. 1, pp. 46–61, 1973. https://doi.org/10.1145/321738.321743

[25] J. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic real-time tasks," *Performance Evaluation*, vol. 2, no. 4, pp. 237–250, 1982. https://doi.org/10.1016/0166-5316(82)90024-4

[26] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. of the Association for Computing Machinery*, vol. 20, no. 1, 1973. https://doi.org/10.1145/321738.321743

[27] N. C. Gaitan, I. Zagan and V. G. Gaitan, "Predictable CPU Architecture Designed for Small Real-Time Application - Concept and Theory of Operation," *Int. J. of Advanced Computer Science and Applications*, vol. 6, no. 4, pp. 47–52, 2015. https://doi.org/10.14569/IJACSA.2015.060406

[28] S. Kelinman and J. Eykholt, "Interrupts as threads," *ACM SIGOPS Operating Syst. Rev.*, vol. 29, no. 2, pp. 21–26, Apr. 1995. https://doi.org/10.1145/202213.202217

[29] N. C. Gaitan, V. G. Gaitan, I. Ungurean and I. Zagan, "Methods to Improve the Performances of the Real-Time Operating Systems for Small Microcontrollers," in *2015 20th Int. Conf. on Control Systems and Computer Science*, Bucharest, 2015, pp. 261–266. https://doi.org/10.1109/CSCS.2015.10

[30] N. C. Gaitan, I. Zagan and V. G. Gaitan, "Improving the Predictability of nMPRA and nHSE Architecture," *Bulletin of the Polytechnic Institute of Iasi, Automatic Control and Computer Science Section, fasc. 1/2015*, pp. 27–38, 2015, ISSN 1220-2169,

[31] E. Dodiu, "Real-Time Hardware Scheduler for FPGA Based Embedded Systems," Ph.D. dissertation, University Stefan cel Mare of Suceava, Romania, 2013.

[32] S. A. Edwards and E. A. Lee, "The Case for the Precision Timed (PRET) Machine," in *Proc. of the 44th annu. Design Automation Conf. DAC '07*, San Diego, 2007, pp. 264–265. https://doi.org/10.1145/1278480.1278545

[33] L. Lindh, "Fastchart – A fast time deterministic CPU and hardware based real-time-kernel," in *Proc. EUROMICRO `91 Workshop on Real-Time Syst*., Paris-Orsay, 1991, pp. 36–40. https://doi.org/10.1109/EMWRT.1991.144077

[34] F. Stanischewski, "FASTCHART – Performance, Benefits and Disadvantages of the Architecture," in *Proc. Fifth Euromicro Workshop on Real-Time Syst*., 1993, pp. 246–250. https://doi.org/10.1109/EMWRT.1993.639104

[35] M. Schoeberl, "A time predictable Java processor," in *Proc. of the Design Automation & Test in Europe Conference, DATE'06*, Munich, 2006, pp. 1–6. https://doi.org/10.1109/DATE.2006.244146

[36] M. Nadeem, M. Biglari-Abhari and Z. Salcic, "JOP-plus - A processor for efficient execution of java programs extended with GALS concurrency," in *2012 17th Asia and South Pacific Design Automation Conf., ASP-DAC*, 2012, pp. 17–22. https://doi.org/10.1109/ASPDAC.2012.6164940

Ionel Zagan received the M.Sc. degree in computer science from the Stefan cel Mare University of Suceava, Suceava, Romania, in 2005. He is currently a Ph.D. student at the Department of Computers of Stefan cel Mare University of Suceava. His research interests include real-time systems, microcontrollers and pipeline processors with parallel execution of tasks. Mr. Zagan is a member of the IEEE Computer Society.
Address: Str. Universitatii nr. 13, 720229, Suceava, Romania.
E-mail: zagan@eed.usv.ro

Vasile Gheorghita Gaitan received the M.Sc. and Ph.D. degree from the Gheorghe Asachi Technical University of Iasi, Romania, in 1984 and 1997, respectively. He is currently a Professor at the Department of Computers of Stefan cel Mare University of Suceava, Romania. His main research interests include real time scheduling, embedded middleware, digital systems design with FPGAs, fieldbuses and embedded system application. He is a member of the IEEE, and a member of the IEEE Computer Society.
Address: Str. Universitatii nr. 13, 720229, Suceava, Romania.
E-mail: vgaitan@usm.ro