IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 26 March 2023, accepted 6 April 2023, date of publication 10 April 2023, date of current version 14 April 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3266150

RESEARCH ARTICLE

Custom Soft-Core RISC Processor Validation Based on Real-Time Event Handling Scheduler FPGA Implementation

IONEL ZAGAN, (Member, IEEE), AND VASILE GHEORGHIŢĂ GĂITAN, (Member, IEEE)
Faculty of Electrical Engineering and Computer Science, Stefan cel Mare University of Suceava, 720229 Suceava, Romania
Integrated Center for Research, Development and Innovation in Advanced Materials, Nanotechnologies, and Distributed Systems for Fabrication and Control (MANSiD), Stefan cel Mare University of Suceava, 720229 Suceava, Romania

Corresponding authors: Ionel Zagan (zagan@usm.ro) and Vasile Gheorghiţă Găitan (vgaitan@usm.ro)

This research was funded by the project "119722/Centru pentru transferul de cunoştinţe către întreprinderi din domeniul ICT—CENTRIC—Contract subsidiar 21773/04.10.2022/DIGI-TOUCH/Fragar Trading", contract no. 5/AXA 1/1.2.3/G/13.06.2018, cod SMIS 2014 + 119722 (ID P_40_305), using the infrastructure 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.

ABSTRACT In dynamic real-time systems (RTS), the synchronous communication model is a source of unpredictable behaviors caused by the difficulty of estimating the maximum lockdown time in a process. Inter-task communication is a critical issue in RTS, even in the case of uniprocessor architectures. Using an FPGA-based development platform, through an SoC project, the implementation of HW_nMPRA_RTOS (a unified acronym for multi pipeline register architecture (nMPRA) where *n* is the degree of datapath resource multiplication, hardware scheduler engine (nHSE) for *n* threads and RTOS application programming interface (API)), dedicated processor architecture was developed, simulated and validated. This paper proposes an innovative soft-core implementation to reduce interrupt latencies while maintaining strong spatial and temporal isolation. Among the results which contain relevance and novelty, we can mention: rapid tasks context switching (1 clock cycle); The implementation of a distributed and versatile interrupt system that allows the interrupt attachment to any task; The implementation of a static scheduler and support for the dynamic tasks scheduling; Rapid response to events of up to 2 clock cycles. We demonstrate the architecture's predictability, scalability, and performance by running a set of benchmark applications on several configurations of HW_nMPRA_RTOS synthesized on a Xilinx 7 Series FPGA.

INDEX TERMS Hardware RTOS, microprocessor, scheduling, field-programmable gate arrays (FPGA), real-time systems.

I. INTRODUCTION

The importance of RTS increases the more they are used in the control of critical applications, where a malfunction of control systems could cause material damage or even incalculable losses. Examples include control systems in the industry, aerospace, robotics, automotive, etc. The scope of RTS is widespread and includes a wide range of other devices where the control system response is valid even after a deadline has passed. Most of these applications using RTS are implemented on a wide variety of hardware devices where embedded systems have the largest share due to their size and scalability. In this context, it can be inferred that efficiency is another characteristic that RTS must fulfill. The schedulers that form the basis of the RTS operation must operate on limited hardware resources, where memory and CPU requirements are significantly lower than in desktop systems. Under these conditions, RTS must be robust and ensure the correct

The associate editor coordinating the review of this manuscript and approving it for publication was Mario Donato Marino.

operation of the system in which they are installed, even when system overloads reach their maximum permissible level. Robustness and fault tolerance are two other characteristic features of RTS [1]. A very important aspect of a scheduling algorithm in an RTS is its ability to predictably schedule system tasks in as short a time as possible. RISC processors are more adaptable for handling asynchronous interrupts because their occurrence is rather verified between CPU elementary execution operations. In the case of complex instruction architectures, interrupt handling is restricted either by the limits of instructions execution or by the need to define specific interruptible points.

Preemptive schedulers introduce fluctuations in task execution times, thus degrading system predictability. Designing a CPU scheduler in hardware requires the introduction of new sequential and combinational elements into the processor that can increase the critical path. A kernel provides many useful services to a scheduler, such as multitasking, interrupt management, communication, and signaling between tasks, resource management, time management, memory partition management, and more [2]. Due to the complexity of industrial and automotive applications and response times [3], issues such as:

−"engine does not run smoothly",

− −"control accuracy of an industrial process is poor",

− − −"poor network performance and slow communication", occur when using RTOS implemented in software. The challenges in real-time embedded systems are very stringent, and in some cases, RTS cannot use RTOS because kernel overheads are too high or do not provide the real-time performance required. These result in the following problems:

- It is not easy to add or modify the software;
- Large-scale software development is difficult;
- Difficult to modularize and update software.

The solution to this problem is given by implementing certain functions or the entire RTOS in HW resulting in the concept of HW_RTOS, Thus, HW_RTOS offers a high level of real-time performance based on fast API execution, and fast interrupt response, low RTOS overhead, small footprint, tick offloading, hardware interrupt service routine (ISR) and much lower CPU resource usage. In classic RTOS, interrupts are the highest priority sections of code that may or may not interact with the thread via RTOS API functions [4]. HW_RTOS can be provided as a hardware intellectual property (IP) block that implements some of the most common functions typically found in a software-based core. A good part of the HW_RTOS registers appear as local memory for the processor, this memory is mapped into the coprocessor 2 (COP2) address space for Microprocessor without Interlocked Pipeline Stage (MIPS32) architecture. This memory can be monitored (i.e. read) to show what HW_RTOS is doing, and writing to, in order to set some of the HW_RTOS operating modes.

Current operating systems (OS) have been redesigned or upgraded to handle multithreading [5], process and thread scheduling, memory management and sharing, multi-core processors, GPU and GPGPU coprocessors, interprocess communication/synchronization, deadlock and starvation, and other real-time and embedded entities that they can exist on modern microprocessor systems. The time required to change contexts is the most significant factor in any RTOS [6]. This is an inherent limitation of the kernel, which does not depend on the scheduling algorithm nor on the structure of the task set. In the case of RTSs, another overhead factor is the time required for the processor to execute the interrupt handling routine.

The main contributions of our work can be summarised as follows. HW_nMPRA_RTOS is a validated implementation of the processor whose initial idea was partially described in [7]. In the initial implementation, only a time event-driven scheduler and deadline events were implemented [7]. A preemptive scheduler based on an interrupts system, mutexes, message events, and deadlines has been added at the COP2 level using additional CPU instructions to control HW_nMPRA_RTOS. The solution proposed in this paper is based on the private resources of threads, referred to as HW_thread_i, with $i = 0, \ldots, n-1$. The hardware instances (instPi) of a thread execute and allocate only one HW_thread_i type resource at a given time. However, the thread context (HW_thread_i) incorporates the CPU registers, instPi environment block, and kernel stack, as well as a user stack associated with the process. Compared to the original implementation, the concept proposed in this paper is compliant with MIPS32 Release 1 instruction set architecture (ISA) which enabled us to use COP2 for the real-time event handling module (Chapter III-D). The HW_nMPRA_RTOS project presented by the authors is based on the concept of multiplication of CPU resources patented in [8]. Issues related to missing factors in existing research projects include non-deterministic execution associated with ISA, task context switch time, and real-time response. The task context switch operation, the inter-task synchronization and communication mechanisms, as well as the jitter that occurred in treating aperiodic events, are crucial factors in implementing RTOS. In practice and literature, several solutions can be identified for improving the response speed and performance of RTSs. Software implementations of RTOS-specific functions can generate significant delays, adversely affecting the deadlines required for certain applications. The HW_nMPRA_RTOS performs the same software RTOS function (selects the next task to be executed) except that this is performed much faster by using hardwired logic. It can be asserted that the chosen solution provides a unitary management approach to interrupts and events. The predictability issues for single/multi-core design-based real-time systems must include RTOS aspects and CPU pipeline structure. Many studies on scheduling algorithms and CPU architectures are proposed in the literature without reference to the proper hardware implementation of the related blocks, the impact on critical paths in the CPU, jitter in the case of RTS, the delay introduced by task scheduling and event handling, and core power consumption.

The proposed method solves these problems by implementing RTOS functions in hardware, adding robust support for events (time, external interrupt, deadline, mutex, messages), and minimizing the time for task context switch due to the separate implementation of thread contexts (HW_thread_i). The proposed design differs from the existing studies because it implements in the FPGA the multiplication of pipeline resources for each task (instPi), and proposes an event-based preemptive scheduler that selects for execution the highest priority task.

The rest of this paper is organized as follows. Chapter II reviews the current research in the field of real-time systems and FPGA designs and Chapter III introduces the RTOS overhead and hardware-accelerated RTOS based on a hardware scheduler and real-time event-handling module. Chapter IV shows the synthesis, and implementation results, and Chapter V provides the discussions and conclusions.

II. RELATED WORK

Reconfigurable devices used in real-time embedded systems and research on hybrid fault-tolerant scheduling are a real challenge to improve the reliability of real-time independent periodic hardware tasks. The technique proposed in [9] schedules tasks according to the EDF-NP policy, and candidate spare tasks are selected to load free processing periods. In the next phase, an event-triggered dispatcher decides which candidate spare tasks should be configured and executed on the FPGA at runtime. Experimental results show that the hybrid scheduling technique proposed in [9] improves the mean time to system failure by an average factor of 1.22 compared to other scheduling algorithms. In mixed-criticality systems, software fault-tolerance techniques are available to mitigate hardware failures, but adapting them to real-time systems is challenging due to the overhead introduced. The paper [10] proposes an extension of traditional scheduling theory for mixed-criticality systems to implement fault-tolerant strategies against transient failures, to meet both failure and timing requirements. In particular, the authors introduce dropout relations that generalize the concept of criticality and allow, on the one hand, to improve the scheduling analysis, on the other hand, to control the dependency between tasks satisfying certification requirements. The simulation study shows an improvement in the scheduling ratio of 20-30% compared to classical scheduling while maintaining compliance with failure requirements. In the case of embedded real-time microkernels, the use of hardware to perform part of the CPU processing functions is a common practice that produces good results in terms of power and performance when applied in embedded systems and beyond.

The paper [11] describes the integration of microkernel functions to increase the performance of task-based systems. This is because the CPU overhead is caused by scheduling algorithms and thread context switches. Therefore, microkernel functions have been implemented by hardware to run in parallel with the CPU, thus reducing the task dispatch time. In the paper [12], the authors address the problem of shared resource arbitration at the OS level and propose a new basic OS design platform centered on a scratchpad. In the proposed concept, the predictable use of shared resources across multiple cores is a central goal during the design process. Therefore, the authors demonstrate how conflict-free execution of real-time tasks on scratchpad-based architectures can be achieved, i.e., how separation of application logic and I/O operations in the time domain can be enforced. In [13] the authors present a new computing system architectural concept, instruction overloading, which can support block ciphers in a RISC CPU core without ISA modification. In the proposed concept, the extended core can perform different context-dependent operations, such as the address of source operands, for a single instruction overhead. Consequently, the proposed RISC core not only provides complicated cryptographic operations but also leads to power analysis resistance with complex masked operations without additional custom instructions. To reduce the real run-time overhead, the authors propose in [14] a quasi-sharing-based scheduling algorithm, called qHS-2S-RTS, which can achieve equivalent scheduling performance to HS-2S-RTS. A new control algorithm for HS-2S-RTS and qHS-2S-RTS, called HSAC-2S-RTS, is also designed. Simulation results validate theoretical analysis that qHS-2S-RTS and HS-2S-RTS can satisfy all feasible firm deadline constraints while improving the schedulability of soft tasks. In the context of real-time tasks, EDF scheduling on multiple cores, paper [15] presents a new migration algorithm for multi-core systems. This implementation is based on the idea of migrating tasks only when strictly necessary to respect their timing constraints in accordance with the EDF algorithm. The proposed adaptive migration algorithm is evaluated by an extensive set of simulations validating the obtained performance. In some embedded control systems, energy efficiency required performance, and scalability is affected by power constraints imposed on heterogeneous processors [16]. Memory wall and communication constraints will continue to increase the gap between the performance of an ideal processor and that of a practical processor. This is because new concepts such as dynamic scheduling, out-of-order execution, or HW integration of scheduling and RTOS APIs will continue to be included in future processors.

The paper [17] proposes and validates an adapted EDF variant and tests that it is at least near-optimal soft real-time. However, the authors present simulation results for random task systems that guarantee that the proposed EDF variant is optimal for real-time soft. In the paper [18] Hybrid-SIMD, a modular BvNC coprocessor architecture reliable to decrease memory access and improve energy efficiency in a classical von Neumann architecture, was proposed. An advantage of the Hybrid-SIMD concept is modularity, as the intelligent row structure can be modified according to the algorithm. However, the user can enter the row interfaces required by a single algorithm, in an application-specific approach, or by a given set of benchmarks, such as in a generic approach. Technological development, the increasing complexity of user applications, the emergence of IoT concepts, and the

amount of data to be processed, all contribute to increasing performance requirements of computing devices [19].

In the paper [20], the authors aim to improve the schedulability of deep neural network tasks in real-time while leveraging heterogeneous resources. Thus a new system abstraction is proposed, which allows transparent and flexible CPU/GPU scheduling of individual deep neural network layers. In [21] a new admissibility test for dynamically partitioned constant bandwidth server reservations on multi-core systems is presented. The concept fundamentally preserves the simplicity of utilization-based testing but leverages knowledge about the future evolution of active utilization within each CPU due to tasks that have recently been dropped, either due to termination or migration, which is tracked by any proper scheduler implementation. Analysis of the timing properties of an application is of particular importance for real-time systems. Only a time-predictable platform can allow the computation of safe but tight worst-case execution bounds [22]. Therefore, real-time systems need time-predictable processors. Real-time applications can also compete for processor execution time, which can lead to deadline misses [23]. Thus, isolation between software components is necessary. This includes spatial isolation in which memory is segregated between protection domains to ensure integrity and temporal isolation in which all execution is prioritized appropriately and the impact of even malicious or errant high-priority executions is limited. In [24] it is shown that the parallelism of instructions in the Ariane kernel does not prevent the imposition of predictability in time. This further relaxes the restrictions, allowing a limited amount of speculative execution, proving that the kernel is predictable in time. Experimental results show that performance is reduced by 10% on average compared to the original Ariane core. In [25] the authors present a time-predictable coprocessor called Vicuna. The proposed vector processor can be scaled to meet the performance requirements of massively parallel computational tasks, but its timing behavior can remain simple enough to be efficiently analyzable.

The same idea from existing studies is the improvement or proposal of new scheduling algorithms [9], [14], [15], [17], [18], [20] and their implementation in hardware or in hybrid form for processor acceleration. Another similar idea introduced in articles [10], [11], [12], [16], HW_nMPRA_RTOS, is to integrate operating system functions, schedulers, or external accelerators in hardware, which are then integrated into real-time or even mixed-criticality systems. Table 1 present difference in features between existing, optimized, and proposed concepts, regarding CPU architecture, time-predictable aspects, and scheduling technique, based on different platforms. The difference from existing studies is the architecture, therefore proposed processors use RISC architectures [13], SIMD [18], dual-issue statically scheduled RISC processors [22], multi-core concepts [12], [21], vector processors [25], simultaneous multithreading (SMT) processors or speculative execution-oriented processors and timing predictability [24]. However, temporal predictability

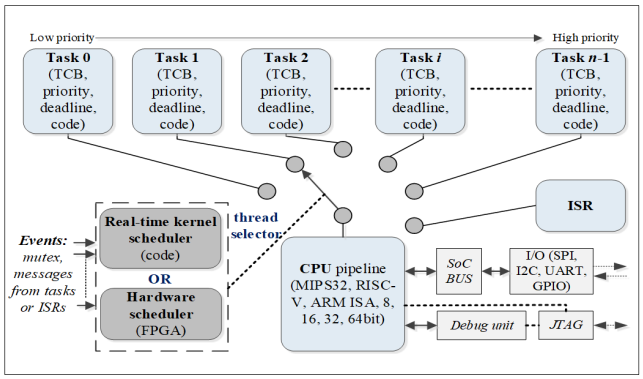

FIGURE 1. Task scheduling with software real-time kernel and hardware RTOS scheduler.

is addressed in different aspects in articles [11], [12], [24], and [25].

III. HARDWARE ACCELERATED RTOS BASED ON PREEMPTIVE SCHEDULER AND REAL-TIME EVENT-HANDLING MODULE

Typically, queues implemented in software use a list structure. In RTOS, TCBs are connected together with a list architecture to implement a queue. There are an enormous number of queues in a classical RTOS. For example, if there are 64 semaphore identifiers, 64 event identifiers, and 16 priority levels, then the total number will be $64 \times 2 \times 16 = 2{,}048$ queues. As a result, even with this relatively small number of objects, plenty of embedded systems resources are required. Thus, the API execution time that accompanies queue management fluctuates dramatically depending on the state of the queues in the RTOS [4]. Also, the resulting overheads are almost as long as the API execution time, so the jitter can also fluctuate dynamically depending on the internal states of the RTOS queue. So when a large number of tasks are attached to queues, queue management can create unexpected overhead, resulting in possible failures in RTSs.

A. RTOS OVERHEAD AND HW-RTOS SCHEDULERS

Multitasking is the process of scheduling and switching the CPU between multiple sequential tasks. Multitasking gives the illusion of multiple CPUs and maximizes CPU utilization, as shown in Figure 1.

With a real-time kernel, application programs are easier to design and maintain. Thus, HW_RTOS does the same function as its software counterpart, i.e. it selects the next task to run, except that it does it much faster using hardwired logic in the FPGA. Also, an HW_RTOS-managed task looks the same as with a software-based core. Independent on pipeline implementation (number of stages), memory system architecture (SoC BUS, cache, fetch queue), debug and scheduling interface (JTAG), and chosen architecture (MIPS32, RISC-V, ARM ISA, 8, 16, 32, 64bit), tasks are scheduled for execution according to the scheme in Figure 1 [4]. Thus, each task relies on its information (datapath context, task control block (TCB), priority, deadline, code) to guarantee safe execution

TABLE 1. The difference in features between related work proposed concepts.

Feature/ Implementation	Hybrid RT scheduling technique [9]	Mixed-Criticality scheduling extension [10]	HM-Plasma [11]	Time-predictable processor [22]	Vicuna [25]
Proposed technique/contribution	Mean-time to-failure (MTTF)	Dropping relationships (DRs) that generalize the concept of criticality	HM-Plasma (single additional internal register bank that operates in parallel with the CPU)	Time-predictable processor/ compiler / Worst-case execution time (WCET) analysis tool	Vicuna for highly parallel real-time applications
Scheduler type and architecture	Static and dynamic real-time scheduling	DRs (improve the schedulability of real-time tasks when fault tolerance tasks are introduced)	Plasma processor/ Preemptive priority-based hardware scheduler	Dual-issue, statically scheduled RISC processor	Timing-predictable vector coprocessor
Platform/Hardware	XUPV5LX110T	Mc-fault-simulator	Intel FPGA Cyclone V	Altera Cyclone IV FPGA (EP4CE115F29C7)	Xilinx 7 Series FPGA
Comparisons/ Benchmarks	Static FT EDF-NP and Pareto-based FT algorithms	Schedulable and compliant tests	IHM-Plasma Gain	LEON3, MicroBlaze, and NIOS II	VESPA and VEGAS
Optimization and improvement factor	1.22 average factor	20-30% schedulability ratio	Performance is virtually independent of the time slice	Optimized for the WCET instead of the average-case execution time (ACET)	Repeatable timing behavior / Free of timing anomalies

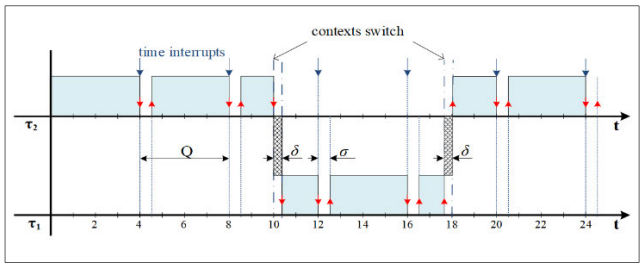

FIGURE 2. RTOS overcontrol effect on task execution based on tick interrupt.

within the deadlines imposed by the RTS, under the stringent conditions imposed by the handling of assigned events (time, deadline, interrupt, mutex, message) and ISR. Figure 2 illustrates the execution intervals σ caused by the routine for handling time interrupts and the intervals δ required for thread context switching.

The effects on periodic task scheduling due to interrupts can be taken into account by adding the factor Ut to the total utilization factor corresponding to the set of tasks [1]. If Q is the system tick and σ is the WCET corresponding to the periodic task, the introduced overhead can be calculated as the utilization factor Ut obtained by equation (1).

$$U_t = \frac{\sigma}{Q} \tag{1}$$

A solution to guarantee system predictability may be to move the primitives and mechanisms of the RTOS, including the scheduler, into hardware (HW_RTOS), thus eliminating the overhead of the basic software-implemented functions and greatly reducing the effect of jitter.

B. HW_nMPRA_RTOS PROJECT DESIGN AND TESTING METHODOLOGY

The design and testing methodology involved the following steps, with overlapping steps in certain stages of testing and reviewing the project specifications. The first step consisted of writing the HW_nMPRA_RTOS concept specification and choosing the ISA (BL_name is MIPS32). Then, using the Vivado DS and VC707 FPGA development kit, the initial XUM design was ported to Virtex-7. Subsequently, the datapath and RTL (Register Transfer Level) for the synthesizable CPU were simulated using the integrated Vivado simulator, in accordance with the MIPS Assembler simulator (Figure 3.a). After this step was successfully completed, the corresponding nHSE module blocks were implemented in Verilog, the multiplication of datapath resources (HW_thread_i) was performed and the HW_nMPRA_RTOS project was simulated in Vivado DS. As can be seen in Figure 3.b, the crTRi and crEVi registers are visualized and the choice of an instPi for execution is made according to the preemptive nHSE scheduler made with a finite-state machine (FSM). For debugging stage we used the customizable Integrated Logic Analyzer (ILA) IP core, so we test synchronously the CPU internal signals.

Finally, the design was integrated into SoC and the proposed scheduler performance was tested in a real-time environment with appropriate comparisons in terms of the achieved requirement/performance ratio. The use of an emulator such as QEMU was not possible because the proposed concept is a hardware extension that relies on a multiplication of the processor datapath for each thread called instPi. Thus, we are considering in the future to develop an emulator for the HW_nMPRA_RTOS concept supporting RISC-V, MIPS32, and ARM ISA. The papers [7], [26], [27] present these simulations at the concept level and do not explain the algorithms underlying the implementation of the real-time scheduler. In [7] the basic concept of nMPRA is presented for the first time, with static Round Robin scheduling. In [26] only a basic solution for handling mutex-type events is presented. In the overview presented in [27] different ISAs for

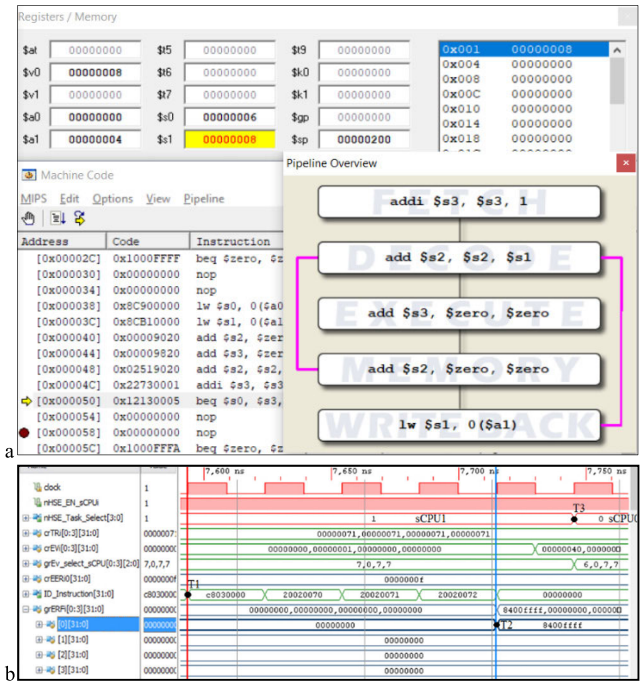

FIGURE 3. (a) Project simulation using five-stage datapath pipeline CPU and data redirection to avoid hazards; (b) HW_nMPRA_RTOS soft-core simulation in Vivado DS developed on Verilog HDL.

nMPRA processor support, namely MIPS32, RISC-V, and ARM, are reviewed. In this paper, the authors describe the dynamic nHSE scheduler with a priority-based preemptive execution for each instPi. Moreover, at the level of each processor instance implemented in hardware, events validated by the crEPRi register can be prioritized. In addition, message events for instPi communication, external interrupt events, and preemptive scheduling mechanisms with flexible scheduling of both instPi priorities and their associated events are presented and tested. Thus, these events inherit the priority (mrPRIsCPUi) of the task to which they are attached, allowing the design of secure applications with predefined deadlines. Based on [7], several practical features were subsequently added, such as FSM for nHSE, event prioritization at each instPi level, hardware implementation of a priority encoder for interrupt events, and realization of the destination task search algorithm for a message using content addressable memory (CAM). This paper adds a qualitative and quantitative study adding the improvements compared to those previous tests so that the basic information for HW_nMPRA_RTOS operation and practical results are presented, including a comparison of power consumption.

C. HW_nMPRA_RTOS CONCEPT AND ARCHITECTURE

In designing and implementing the HW_nMPRA_RTOS architecture, several questions and challenges arise: Which types of registers and functional units should be implemented, and how many are needed for each HW_thread_i? Under what conditions can a functional unit be shared between two or more instPi? How can functional units and multiple registers be linked together so that all register transfer operations are performed correctly?

The nHSE unit has the role of activating only one instance at a time. If one instPi is turned off at some point by nHSE and another instPi is enabled, all program-specific information running on the stopped instPi is preserved due to the multiplication of the PC registers, pipeline registers and RF associated with HW_thread_i hardware resources. The program running on instPi can be a task from a real-time multitasking application. Switching from one instPi to another does not require the saving of general-purpose registers or deletion of the contents of pipeline registers, which causes a very fast context switch. If each instPi runs task *i*, switching from one task to another is also achieved very quickly.

The interrupt management mechanism of an RTOS provides for the management of interrupts that may be generated by peripheral devices such as analog-to-digital converters or sensors. In classical OSs, this mechanism involves the execution of a dedicated interrupt driver routine to transfer data from the peripheral device to the main memory, or vice versa. Thus, application tasks can be interrupted at any time by interrupt routines. In the case of this research project, interrupt-type events are associated with instPi, thus increasing the predictability and performance of RTS. The nHSE integrated hardware scheduler is without a doubt the central block of the HW_nMPRA_RTOS processor. To achieve outstanding performance, nHSE is designed as a constituent part of the processor (Figure 4). In the case of context switching, the preemptive scheduler does not use stack saving of registers as is the case with other architectures, thus eliminating multiple accesses to external memory.

The performance plus is due to the concept of remapping task contexts (HW_thread_i), which is done in practically one clock cycle due to the multiplication of processor resources for each instPi. Even if all instPi use the same BL_name (RISC-V, MIPS32, ARM) datapath, resource multiplication is not an impediment to the practical use of this processor in real-time applications. The nHSE module illustrated in Figure 4 is designed to satisfy the following functions:

- Preemptive scheduling of tasks and therefore interrupts;
- Implementation and management of deadline events;
- Select bank of kernels via nHSE_Task_Select selector;
- Selection of pipeline registers (part of HW_thread_i) corresponding to each instPi and PC.

The project written in Verilog HDL validates both nHSE_Task_Select command signals and CPU datapath signals. Within the proposed CPU datapath, each task has a PC, an RF, and a set of its own pipeline registers, as we can see in Figure 4. The change of task contexts is controlled by the scheduler using the nHSE_Task_Select selector. The execution of a scheduled task is done by simply remapping multiplexed hardware resources (HW_thread_i) related to instPi to be executed due to a higher-priority event.

Table 2 shows the memory map for the local (lr), control (cr), and monitoring (mr) registers. Thus, the

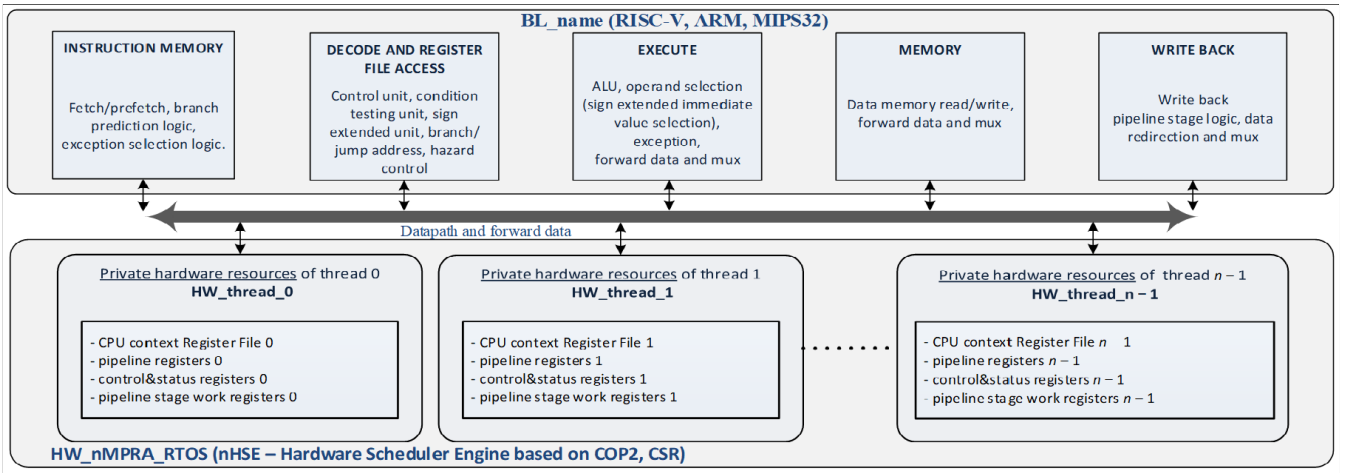

FIGURE 4. Architecture and connections between HW_nMPRA_RTOS blocks based on hardware scheduler registers are defined in Table 2.

HW_nMPRA_RTOS global registers (gr) for a MIPS32 architecture are shown [28]. The following characteristics are highlighted: current number (for GPR only and additional registers), whether cr, lr, mr, or gr, whether mandatory (m) or optional (o), and what it represents. The crTRi register contains one bit for validating time events, such as deadline, interrupt, mutex, and synchronization, respectively one bit for validating program execution on instPi.

The mechanism for attaching interrupts to a specific task is based on a set of grINT_IDi registers used to store the IDs of the tasks to which external interrupts have been attached, NR_INT being the number of interrupts in the processor. By properly associating interrupts with the ID of the attached tasks, an interrupt can be treated as the highest priority interrupt in the system. Through the hardware-implemented address priority encoder, the scheduler activates and launches into execution the task to which the interrupt is attached (Figure 5, interrupt event). The scheduling logic is based on task priorities so that the occurrence of interrupts is not an impediment to meeting the deadlines of the task set. The mutex registers constitute the Mutex Register File (MRF). grMutexi registers select the task ID to which the Mutexi mutex has been attached (bit 31 of this register).

grSMRi registers compose the Message Synchronization Event Register File (SMRF) and store the event status, source, and destination identifier and afferent message. The bank selection in the GPR is independent of the operations performed at the level of the individual instPi. Recall that in the datapath designed at the RTL for the HW_nMPRA_RTOS architecture, the control lines for each bank in the GPR are separate. These bits are implemented at the level of each instPi and are named lr_enTi, lr_enWDi, lr_enD1i, lr_enD2i, lr_enInti, lr_enMutexi, lr_enSMi and lr_run_instPi. The crEVi register is a register that is used to indicate the occurrence (1L) or not (0L) of an event validated by individual bits in crTRi. The bits in the crEVi register, corresponding to the bits in crTRi, are named as follows: TEvi, WDEvi, D1Evi, D2Evi, IntEvi, MutexEvi, SMEvi and lr_run_instPi. Table 3 shows the global registers for external interrupts, mutexes, and message synchronization-related events.

The modification of the registers corresponding to the instPi scheduled by nHSE is done on the positive clock edge, the outputs being of type wire assigned to variables of type registered, depending on the selected instPi. The resource requirement for the implementation of this module is minimal, which is due to the FPGA implementation of the multiplication registers. However, particular importance must be given to the operations of COP2 and providing the necessary information to the nHSE scheduler, control unit, hazard detection, shift register, and register file.

D. nHSE SCHEDULER OPERATION AND EVENT HANDLING LOGIC

Compared to previous work, especially [7], the amount of improvement resulting from a quantitative evaluation is better both in terms of response time to external events and switching thread contexts. This is due to separate hardware contexts for each thread (HW_thread_i), an implemented state machine for nHSE, the use of CAM for RTOS message destination lookup, and the realization of a priority coder for interrupts. The main contributions of this work with respect to the original project concern the proposal of algorithms that minimize context switching time; the design and implementation in FPGA of a flexible and versatile scheme for handling time, mutex, and timing events attached to a task, i.e. these events can be prioritized in turn and at the level of an execution thread; the proposal of an interrupt event handling algorithm implemented in the nHSE dynamic scheduler. The logic for hardware events handled by the nHSE scheduler is illustrated in Figure 5. With independent execution, the scheduler has entries for events such as interrupts and timers (TEvi, WDEvi, D1Evi, D2Evi, IntEvi, MutexEvi, SMEvi). The crTRi register is used to check if the corresponding events are validated (see Table 2 and Table 3), so at the level of each instPi there is a pattern like the one shown in Figure 5. In the integrated fail-safe mechanism implementation, the

TABLE 2. Memory mapping for local (Lr), control (Cr), and monitoring (Mr) nHSE registers.

No.	NAME MIPS32	Type	Name HW_nMPRA_RTOS	Type	m/o	Description
0	Zero	lr	crTRi	cr	m	Enable/disable event (lr_enTi, lr_enWDi, lr_enD1i, lr_enD2i, lr_enInti, lr_enMutexi, lr_enSMi, lr_run_instPi)
1	at	lr	crEVi	cr	m	Read the events state (TEvi, WDEvi, D1Evi, D2Evi, IntEvi, MutexEvi, SMEvi)
2	v0	lr	crEPRi	cr	o	Set the priority of major events at the level of each instPi
3	v1	lr	cr0D1	cr	m	instP0 reads all D1 events from all instPi
4	a0	lr	cr0D2	cr	m	instP0 reads all D2 events from all instPi
5	a1	lr	cr0MSTOP	cr	m	Allows instP0 to stop the execution of all other instPi
6	a2	lr	cr0RESET	cr	m	Reset instPi (access only for instP0)
7	a3	lr	cr0CPUID	cr	m	Identify HW_nMPRA_RTOS (access only for instP0)
8	t0	lr	crEMRi0	cr	m	Validates mutexes (access for instPi)
9	t1	lr	crEMRi1	cr	o	Validates mutexes (access for instPi)
10	t2	lr	crEMRi2	cr	o	Validates mutexes (access for instPi)
11	t3	lr	crEMRi3	cr	o	Validates mutexes (access for instPi)
12	t4	lr	crEERi0	cr	m	Validates synchronization events via messages attached to instPi
13	t5	lr	crEERi1	cr	o	Validates synchronization events via messages attached to instPi
14	t6	lr	crEERi2	cr	o	Validates synchronization events via messages attached to instPi
15	t7	lr	crEERi3	cr	o	Validates synchronization events via messages attached to instPi
16	s0	lr	mrPRIsCPUi	mr	o	Defines dynamic priority for instPi
17	s1	lr	mrTEVi	mr	m	The reload value of the timer attached to instPi
18	s2	lr	mrWDEVi	mr	m	Alarm value for the watchdog attached to instPi
19	s3	lr	mrD1EVi	mr	m	Alarm value for deadline 1 assigned to instPi
20	s4	lr	mrD2EVi	mr	m	Alarm value for deadline 2 assigned to instPi
21	s5	lr	mrCntRun	mr	m	Operating cycles of instPi
22	s6	lr	mrCntSleepi	mr	m	Stationary cycles of instPi
23	s7	lr	mr0CntSleep	mr	m	HW_nMPRA_RTOS stop cycles (is also a cr register)
24	t8	lr	mrCommRegi0	mr	o	Register 0 for intertask communication (access for instPi)
25	t9	lr	mrCommRegi1	mr	o	Register 1 for intertask communication (access for instPi)
26	k0	lr	mrCommRegi2	mr	o	Register 2 for intertask communication (access for instPi)
27	k1	lr	mrCommRegi3	mr	o	Register 3 for intertask communication (access for instPi)
28	gp	lr	mrCommRegi4	mr	o	Register 4 for intertask communication (access for instPi)
29	sp	lr	mrCommRegi5	mr	o	Register 5 for intertask communication (access for instPi)
30	fp	lr	mrCommRegi6	mr	o	Register 6 for intertask communication (access for instPi)
31	ra	lr	mrCommRegi7	mr	o	Register 7 for intertask communication (access for instPi)
…						
j			mrCommRegij	mr	o	Register j for intertask communication (access for instPi)

TABLE 3. nHSE associated global registers.

Address	NAME HW_nMPRA_RTOS	Type	m/o	Description
Base0				
0	grINT_ID0	gr	m	Registers for attaching interrupts to instPi
1	grINT_ID1	gr	m	Registers for attaching interrupts to instPi
…	…	gr	m	Registers for attaching interrupts to instPi
p-1	grINT_IDp-1	gr	m	Registers for attaching interrupts to instPi
Base1 = Base0 + 256				
0	grMutex0	gr	m	The ID of the task to which the mutex is attached
1	grMutex1	gr	m	The ID of the task to which the mutex is attached
…	…	…	…	…
m-1	grMutexm-1	gr	m	The ID of the task to which the mutex is attached
Base2 = Base1+ 256				
0	grSMR0	gr	m	The registry defines an event of type synchronization via messages
1	grSMR1	gr	m	The registry defines an event of type synchronization via messages
…	….	…	…	…
e-1	grSMRe-1	gr	m	The registry defines an event of type synchronization via messages
Base3 = Base 2+256				
0	grSMFG	gr	m	Register (b0 = 1) defines if the message event file is full
Base4 = Base 3 +256				
0	grINT_PR	gr	o	The register stores the interrupt number attached to this priority which is the highest priority if an interrupt priority encoder is implemented

highest priority semiprocessor instP0 is used. Thus, the HW_nMPRA_RTOS processor can enter a stable state of a system instability situation or a critical error. instP0 is the active task after reset and has the highest priority task in the processor, causing any running task to be suspended when a critical error occurs. In the case of an exception, the

processor enters a controlled fail-safe state and executes the exception handler. The nHSE_EN_sCPUi selector is the activation command for tasks to which events are attached. When the scheduler activates the instPi via the cr0MSTOP signals, the block diagram simultaneously decodes the mrPRIsCPUi, crTRi, crEVi, and grINT_IDi registers for the active tasks. At some point in normal execution, only one instPi can be in the RUNNING state. This is made possible by the nHSE scheduler, instPi states, the event block (crTRi[j] & crEVi[j]), and timing signals. If it is desired to measure the delay to activate a digital output (LED), mapped to the address space of the data memory (DataMem_Address[29:26]), then the processor must execute the store word (sw) instruction.

The jitter is only two clock cycles, the time required to identify the external interrupt (ExtIntEv[0]) and to activate the instPi to which the interrupt is attached. Then, additional time is needed to execute the instructions to change the logic level of the FPGA circuit pin. The identifier register block contains an ID for each instPi, a register with the priority (mrPRIsCPUi, see Table 2) attached to each instPi used only by the dynamic scheduler (except for instP0 which is always the highest priority), and a global register containing the identifier of the active instPi.

In the case of mutex event implementation logic in HW_nMPRA_RTOS, the number of mutexes has been denoted by m, so the MRF for mutexes is constituted based on m (Figure 6.a) and instPi. The grMutexi register selects the instPi identifier to which mutex *i* has been attached, after resetting all bits to 0L.

At the level of each instPi there is a pattern like the one illustrated in Figure 6.b that allows a type event to be generated whenever an expected mutex is free. At the level of each instPi it is possible to decide which mutex is considered with the help of the signals lr_en_Mi0, . . . , lr_en_Mim-1. These signals are stored in local registers called enable mutex registers (EMR). There can be one or more EMRi registers depending on the number of mutexes implemented in the MRF. For synchronization with the processor clock, the D-type flip-flop is used which memorizes the information on the rising edge of the processor clock. Mutex_i bit set and reset operations are performed, using the scheme in Figure 6.b, indivisibly in a single processor cycle. A test and set instruction (in_tasm_wr and Address_i signals) will read the old value of the mutex bit developed from the MRF (Mutex_i), and set the Mutex_i bit. If the Mutex_i bit was set to 1L it was used and the last *n* bits of the MRF register read (Address_i sign) domain the mutex owner ID, which is provided by the in_tasm_wr, Address_i and /Mutex_i signals in Figure 6. If the mutex bit is set to 0L it will set the mutex bit to 1L and write the ID of the instPi, from which the instruction was executed, to the common MRF register (Figure 6.a, for a bit in the MRF other than Mutex_i). If the Mutex_i bit read by the instruction was 0L, then the mutex was not found, only the read ID belongs to the owner of the mutex. The delete instruction creates the in_clrm_wr and Address_i signals and sets Mutex_i to 0L.

The algorithm 1 show the selection of processor instances (instPi) that are in the READY state implemented in the nHSE dynamic scheduler. Thus NR_TASKS represents the total number of instPi that have dedicated hardware resources (HW_thread_i). Initially, the cr0MSTOP register (see Table 2) is checked to test if the instPi is not in the stopped state for execution. Then, the algorithm checks if instPi has at least one validated and active event (crTRi[i] & (crEVi[i]) and the lr_run bit is set, which condition causes the corresponding instPi bit in nHSE_sCPUi_Ready to be set. Subsequently, the algorithm implements the encoder for dynamically prioritized tasks that provides the highest priority instPi in the sCPUi_Lower_PRI register based on the priorities stored in mrPRIsCPUi[0:NR_TASKS-1].

It should be specified that this algorithm is transposed in hardware, the FPGA logic contributing to generate a coder that provides in the shortest time (one clock cycle) the instPi ID with the highest priority (lowest value for mrPRIsCPUi). Algorithm 2 generates a message-type event based on SMRF. This checks if there is at least one free input for signals (grSMRFG), then checks the Signal bit in each grSMRi register (grSMRi[i] & 32'h80000000) (Figure 5). The next step in the algorithm is to search the index of the first signal for all instPi, then signaling message events (signals). The effect of this algorithm is to set the SMEvi bits of crEVi, validated by the lr_enSMi bits of component crTRi (crEVi[(((grSMRi[i] & 32'h03E00000) ≫ 21)] <= crEVi[(((grSMRi[i] & 32'h03E00000) ≫ 21)] | Mask1_bit6;). Based on the hardware implementation of the search operation in grSMRi, the setting of a SMEvi event is done in one CPU cycle. The search in SMRF is performed in one processor cycle, based on the CAM principle. Then, Algorithm 1 uses the priority queue to launch instPi into execution and executes a direct jump to the trap cell associated with the event that generated the RUNNING state. The scheduling operation using a preemptive algorithm is based on HW_thread_i resource selection with zero theoretical overhead. The scheduling algorithm can be offline (static) or online (dynamic). The offline variant is rudimentary and assumes that the scheduler's decisions are determined apriori. The next condition checks the occurrence of time events according to the event priorities in the crEPRi register at the level of each instPi. Obviously, this processing type is very rigid, strictly dependent on a specific application architecture, and difficult to use for medium and high-complexity real-time applications. For this reason, current RTOS does not use this kind of approach. Unlike static scheduling, online or dynamic schedulers decide which ''ready to run'' or ''running'' task should receive the active resource, with up-to-date information about task states. Control over the processor is transferred to the higher priority task (instPi), and for tasks on the same priority level, the RR (Round Robin) or FIFO (First In First Out) algorithm can be applied in relation to the arrival time in the list of ready to run tasks. Dynamic scheduling algorithms can work with static or dynamic priorities based on mrPRIsCPUi registers. In the first case, task priorities are

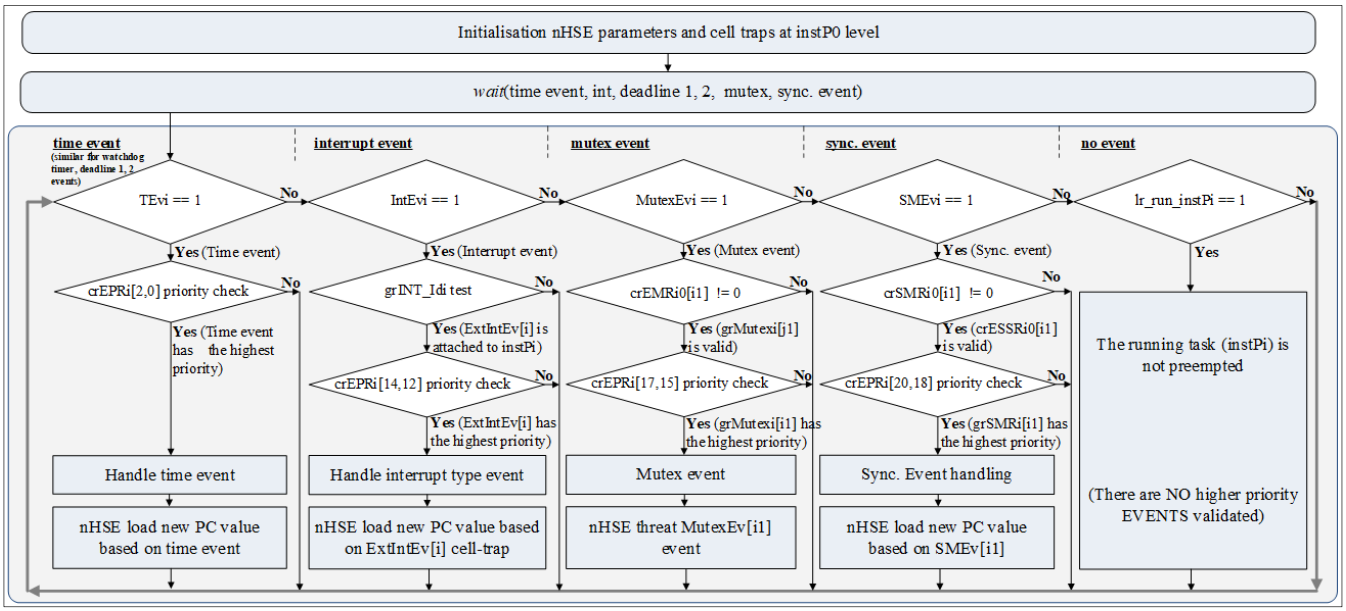

FIGURE 5. The hardware solution for event handling is implemented in the nHSE scheduler (COP2) with effect on HW_thread_i.

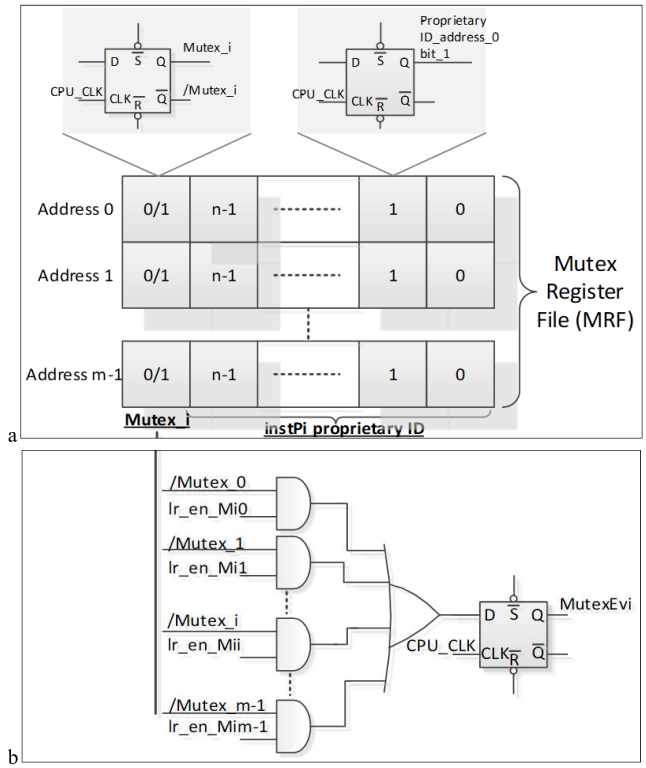

FIGURE 6. (a) MRF logic implementation schemes; (b) MutexEvi event generation logic with MRF and nHSE registers.

not changed by the scheduling algorithm. It should be noted that task priorities may vary during the execution of the application, but these changes are not managed by the scheduler. In the second case, the priorities are also changed by the scheduling algorithm (EDF), taking into account the nHSE-associated time constraints (TEvi, WDEvi, D1Evi, D2Evi).

IV. FPGA SYNTHESIS AND IMPLEMENTATION RESULTS BASED ON HW_nMPRA_RTOS EVALUATION

In this paper, the HW_nMPRA_RTOS architecture validation and the SoC design used for testing the implementation proposed in this paper were presented. By means of practical results obtained using the Vivado simulator and the ChipScope analyzer, the practical implementation of the theoretical approaches presented in Chapter III has been demonstrated.

The proposed project needs to be newly implemented and verified on FPGA because the initial project which included a simple multiplication of the register file and pipeline registers was realized in VHDL, using Xilinx ISE and Virtex-6 environment.

In the new HW_nMPRA_RTOS implementation, Vivado DS environment was used, with Virtex-7 and Verilog hardware description language. The Vivado DS tool allowed rapid project simulation, debugging using ChipScope and hardware testing of the above-mentioned modules.

A. VALIDATION OF THE PROPOSED HARDWARE SCHEDULER IMPLEMENTED IN HW_nMPRA_RTOS

The elements of difficulty of this research appear at the HW_RTOS level, due to embedding its functionality in hardware and highlighting performance improvement through appropriate test programs. Based on FPGA design methodology and optimization techniques, Figure 7 illustrates the results from the design and debugging stage of the HW_nMPRA_RTOS concept at a working frequency of 33 MHz with a corresponding bitstream running on the board. The signals were captured with the PicoScope 6404B oscilloscope on the AM39, AN39, AR37, AT37 VC707 FPGA circuit pins (LEDs mapped in the address space of the data memory of the processor). For implementation,

Algorithm 1 nHSE Dynamic Scheduler: Select Which instPi Are in the Ready State

```
always @(posedge clock) begin
  for i = 0, NR_TASKS do
      if (((cr0MSTOP & Wire_Mask1_bit_i[i])!=32'h00000000) &
          ((crTRi[i] & (crEVi[i]|32'h00000080))!=32'h00000000)) begin
          //If there is a validated and occurred event or if the lr_run bit is set
          nHSE_sCPUi_Ready = nHSE_sCPUi_Ready | Wire_Mask1_bit_i[i];
      else
          nHSE_sCPUi_Ready = nHSE_sCPUi_Ready & Wire_Mask0_bit_i[i];
      end if
  end for
  sCPUi_Lower_PRI = 32'hFFFFFFFF; //The dynamic prioritized instPi encoder providing the highest priority instPi ID
  for i = 0, NR_TASKS do
      if (((nHSE_sCPUi_Ready & Wire_Mask1_bit_i[i])!=32'h00000000) & (mrPRIsCPUi[i] <
          sCPUi_Lower_PRI)) begin
          sCPUi_Lower_PRI = mrPRIsCPUi[i];
      end if
  end for
end always
```

Algorithm 2 Generate a Message Type Event Based on SMRF

```
always @(posedge clock) begin
  grSMR = 32'h00000001;
  for i = 0, NR_EV-1 do//check if it is at least one free input for signals
    if((grSMRi[i] & 32'h80000000) != 32'h80000000) begin  //if the signal bit is 0 the input is free
      grSMRFG = 32'h00000000; //grSMRFF = 0 indicates that the signals register file is not full
    end if
  end for
  for i = NR_EV-1, i >= 0; i– do begin //search the index of the first free signal in the grSMRF
    if ((grSMRi[i] & 32'h80000000) == 32'h00000000) begin
      index_FreeSignal <= i;
    end if
  end for
  for i = 0, NR_TASKS do //initialize crSMRindex registers
    crSMRindex[i] = 32'hFFFFFFFF;
  end for
  for i = NR_EV-1, i>=0; i– do //search the index of the first signal for all instPi
    if ((grSMRi[i] & 32'h80000000) == 32'h80000000) begin //if it's an active signal
      crSMRindex[((grSMRi[i] & 32'h03E00000)≫21)] = i;
    end if
  end for
  for i = 0, NR_EV do //signaling of message events (signals)
    if (((crSMRi0[i] & Wire_Mask1_bit_i[i]) !=
      32'h00000000) && ((grSMRi[i] & 32'h80000000) == 32'h80000000)) begin
      if (((crTRi[((grSMRi[i] & 32'h03E00000)≫21)] & Mask1_bit6) != 32'h00000000)) begin
        crEVi[((grSMRi[i] & 32'h03E00000)≫21)] <= crEVi[((grSMRi[i] & 32'h03E00000)≫21)] | Mask1_bit6;
      end if
    end if
  end for
end always
```

debugging, and testing we used Virtex-7, which has a 200MHz differential oscillator. Based on CPU Harvard architecture, the dual-port on-chip memory was designed with IP Block Memory Generator 8.4 and clocked by IP Clocking Wizard 5.2. The design methodology used to derive the HW_nMPRA_RTOS concept involved writing CPU

specifications, writing Verilog HDL code, Vivado DS RTL simulation, synthesis, implementation, report generation, and analysis to meet timing and power consumption constraints, bitstream generation, FPGA programming, and in-circuit testing using an oscilloscope.

The data in Figure 7 and Figure 8 are obtained from a bitstream running on the Vistex-7 board. Figure 7 shows three tasks scheduled by nHSE by running a benchmark application on instP4 configuration, instP0 being the highest priority, followed by instP1. Oscilloscope signals indicate the time each instPi is running, this is done by setting a pin of the FPGA with *sw* instruction. Thus, preemption points T1 and T2 are indicated, then when instP0 finishes execution, at time point T3, instP1 is selected by the FSM with a jitter of only one clock cycle because the instP1 context has been stored by HW_thread_1. At time point T4 switches task instP3, which is the lowest priority task in the system. An overview shows that saving the state of instPi tasks in their own HW_thread_i hardware space simplifies context switching and improves CPU execution time.

In the second situation illustrated in Figure 7.a, it is observed that instP0 enters execution (T6), but although validated events occur at instP1 and instP3 (arrival time), they are scheduled at time points T7 and T8. A task set is schedulable with nHSE if equation (2) is verified, the preemptive priority-based scheduler providing handling of n instPi, p interrupt events, m mutex events, and s message timing events (3).

$$U = \sum_{i=0}^{n-1} t\_instPi \leq 1 \tag{2}$$

$$\begin{aligned} t_{instPi} &= tTEVi + tWDEvi + tD1Evi + tD2Evi \\ &+ \sum_{i=0}^{p-1} t\_IntEvi + \sum_{i=0}^{m-1} t\_MutexEvi \\ &+ \sum_{i=0}^{s-1} t\_SMEvi \end{aligned} \tag{3}$$

Thus, t_instPi is the time for instPi task execution, and tTEvi, tWDEvi, tD1Evi, and tD2Evi represent the time spent for the execution of time events. It should be mentioned that the nHSE scheduler can run scheduling algorithms such as EDF, as it has a set of timers available for measuring instPi run cycles (mrCntRuni), instPi idle cycles (mrCntSleepi), and HW_nMPRA_RTOS idle cycles (see Table 2).

Figure 7.b shows the preemptive scheduling of events at the instPi level when the oscilloscope is set to persistent mode. It can be verified that instP0 interrupts instP2 and instP3 at time points T1 and T2, thus validating the correct execution of the scheduling scheme, with a dynamic attachment of events to instPi. IntEvi interrupts events borrow the priority (mrPRIsCPUi) of the task to which it is attached so that the search for the source of the interrupt and the jump to the trap cell associated with this event is performed in hardware.

The watchdog component is the only element that monitors the execution of tasks, the integrated hardware scheduler is relieved of calculating execution times. The feasibility study on task set schedule is the offline operation performed at the system design stage, based on the real-time constraints discussed in Chapter III of this paper. Therefore, the computation of the processor load and the predictability of the HW_nMPRA_RTOS processor guarantee successful task set scheduling. To handle multiple events with its own trap cell (interrupts, time, mutexes, and message events), a solution to integrate the priority encoder for interrupts and local events associated with instPi has been proposed. This results in an automatic jump to the handler for the highest priority event associated with instPi. The identifier register block contains an identifier for each instPi, a register with priority attached to each instPi (mrPRIsCPUi) used only by the dynamic scheduler (except instP0 which is always the highest priority), and a global register containing the active instPi identifier. For the implementation of these registers, as well as the combinational logic associated with the scheduler, this paper presents the FPGA circuit resource requirements for the chosen degree of multiplication.

Figure 8.a shows the sequence of events occurring at the FSM-controlled nHSE scheduler. This shows the handling of an asynchronous event (cursor C1 at time moment T1) in a maximum of 2 clock cycles. Subsequently, the priority encoder for TEvi, WDEvi, D1Evi, D2Evi, IntEvi, MutexEvi, and SMEvi events will select the highest priority event based on the priorities of each of the crEPRi. Each mentioned event has 3 priority bits at each instPi level (crEPRi register). Cursor C2 indicates the setting of the evIi bit (T2) in the crEV0 register, the jitter in this case is a maximum of 30.3 ns. This depends on when the ExtIntEv[0] event occurs, with priority crEPR0[14:12] and attached to instP0, and can be a maximum of one processor clock period. Cursor C3 illustrates the moment when nHSE performs context switching at instP0 (T3), and FSM switches the task to the RUNNING state (nHSE_FSM_state = FSM_sCPU0). The condition (crTRi[sCPU0_ID] & Mask1_bits)&&(crEVi[sCPU0_ID] & Mask1_bits) validates the treatment of the type *i* event, so that cursor C4 shows the time when instP0 fetches the first instruction to be executed in the pipeline (T4). As we can see in Figure 8.a the time elapsed from the moment an asynchronous event occurs with the processor to the actual execution of the handler's afferent code is minimal (maximum two clock cycles). Figure 8.b illustrates the jitter of the nHSE scheduler in the case of handling an asynchronous message event (T1). The measured period of 571.7 ns includes the time to send a message from instP3 to instP0, search based on hardware for the SMEvi event, schedule the execution of instP0 (T2), and delete that event.

Unlike timing mechanisms implemented in software, the nHSE scheduler checks for SMEvi events in hardware and based on the destination task in grSMRi places the corresponding task in the READY state. With HW_nMPRA_RTOS, it is easy to implement multiple interrupt-handling modules, by assigning an interrupt priority (instPi, mrPRIsCPUi). In HW ISR, multiple interrupts are implemented according to the priority of the tasks being activated. The performance of a hardware-implemented RTOS comes at a low cost considering that using Synopsys Design

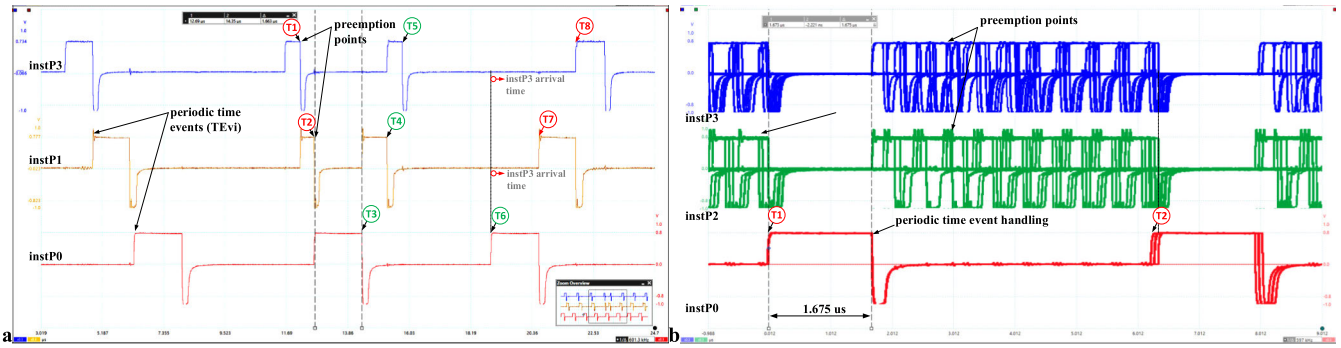

FIGURE 7. HW_nMPRA_RTOS multiple events scheduling validations (instP0 = 1.663 us) by using the PicoScope 6404B oscilloscope by Pico Technology (St Neots, UK), (a) Preemptive event scheduling at instPi level; (b) Execution selection of processor instance instP0 (highest priority) against preemption points (persistent oscilloscope mode).

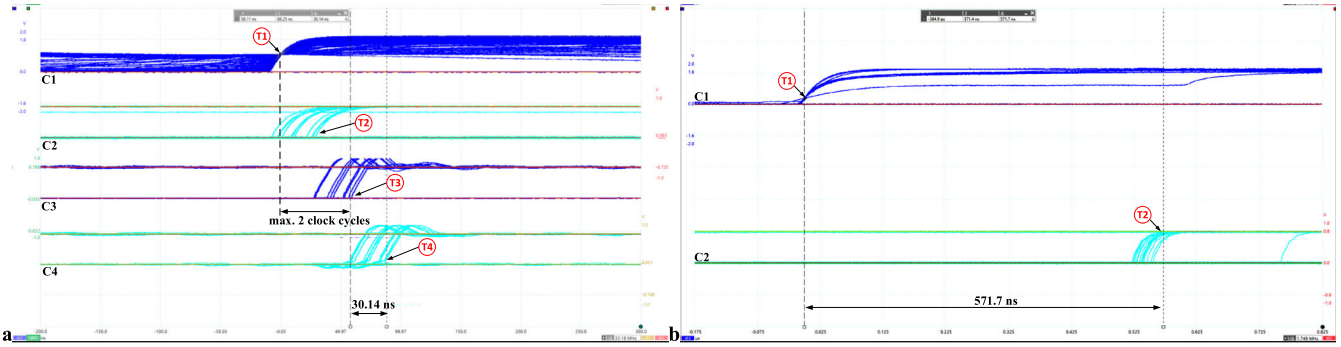

FIGURE 8. (a) HW_nMPRA_RTOS kernel latency to asynchronous event attached to instP0 (T1: Asynchronous external event (ExtIntEv[0]); T2: Setting the evIi bit in the crEVi nHSE register; T3: Context switch to instP0 (FSM state change); T4: instP0 fetch instruction to execute and decode); (b) The jitter of the nHSE scheduler in the case of handling a SMEv0 message-type synchronization event (T1: Ir_enIi external event; T2: inter-thread communication jitter (SMEvi event)).

Compiler, the space occupied by the HW_RTOS is about 8476 LUTs. Integrating a hardware scheduler into the data path requires the design of new combinational and sequential elements that can increase the critical path in the CPU. In the proposed concept architecture, the multiplication of processor units per processor instance (instPi) has a direct effect on the critical path (Table 4).

The maximum response time of the nHSE scheduler, for an event defined in crEvi was 60.6 ns (from T1 to T3 time moment), whereas the response time jitter of the external interrupt event type is about 30.14 ns (due to the time period T1 - T2). This architecture is complete and provides a maximum task-switching time of two processor cycles. HW_RTOS does not have a cyclic management function. However, an equivalent function can be performed using an HW ISR. Furthermore, the startup time of a cyclic activation task using an HW ISR is shorter than that of a cyclic handler running on conventional RTOS software. The procedure is simple: define the input to the HW ISR (ExtIntEv[0]) as the output of the built-in time event (TEvi) of the HW_nMPRA_RTOS. By designing a system that includes multiple interrupt management, you can perform multiple cyclic activation tasks. In fact, it is possible to define three or more cyclic activation tasks. The periods of each task do not have to synchronize with each other. It is also possible to trigger the cyclic activation of an external pin.

B. RESOURCE USAGE AND SYNTHESIS RESULTS

To achieve a predictable and reliable real-time system, a timer block has been designed in the nHSE component to monitor task execution (mrCntRuni). The implemented processor predictability is defined by the ability to change task contexts in one clock cycle. Through a special implementation of the GPR and pipeline registers using programmable logic technology, each instPi corresponds to a bank of registers. The multiplication of pipelined registers guarantees hardware isolation of tasks under the nHSE module direct control. The HW_nMPRA_RTOS architecture provides superior performance in terms of response time to external events and context switching time, and is suitable for small real-time applications due to the resource consumption of register multiplication, as we can see in Figure 9. The performance and stability of the HW_nMPRA_RTOS architecture can be improved by designing a dedicated hardware module to protect against unauthorized memory access. This module is very important because it protects private memory areas belonging to hard real-time tasks, ensuring process stability at maximum performance. Figure 9 presents the Virtex-7 FPGA resource utilization based on different projects, including the HW_nMPRA_RTOS concept based on a dynamic hardware scheduler. For nHSE registers the implementation, as well as the combinational logic associated with the scheduler, experimental soft-core

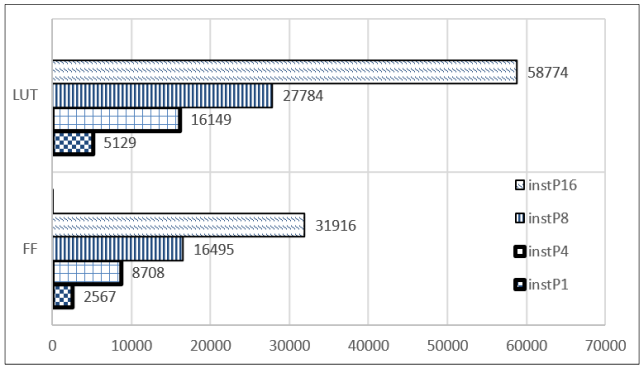

FIGURE 9. FPGA device resource utilization for the proposed HW_RTOS_nMPRA architecture and its configurations.

TABLE 4. FPGA timing report based on different processor implementation.

Timing / CPU	RISC-V core [29]	Cortex-M3 [30]	HW_nMPRA_RTOS	MicroBlaze [31]
Worst negative slack (WNS)	2.106 ns	0.785 ns	3.395 ns	1.857 ns
Total number of endpoints (setup)	3693	74965	35439	4120
Worst Hold Slack (WHS)	0.054 ns	0.078 ns	0.056 ns	0.042 ns
Total number of endpoints (hold)	3693	74907	35439	4120
Worst pulse width slack (WPWS)	1.100 ns	3.000 ns	1.100 ns	3.000 ns
Total number of endpoints (pulse width)	1116	12538	9723	1652

CPU implementations show the FPGA resource requirements according to HW_RTOS_nMPRA multiplication degree and scheduler support, including dynamic priorities for instPi. To test whether all instructions were implemented correctly, another testbench was created that does nothing but operations using a few values loaded from memory.

Some of these instructions have certain dependencies and others do not. Each instPi instruction is stored in different registers, so it was possible to check them using the simulator included in the Vivado environment. So, by checking the final values in memory obtained from the arithmetic and logic operations, it was sufficient to test whether the instructions were implemented correctly. Table 4 shows the information generated by Vivado DS on FPGA implemented timing report, so all the user-specified timing constraints are met. WNS is the most negative of all the paths that failed any constraint (3.395 ns). Again, unless it is put into perspective, it only serves to show how badly the designer missed the timing closure. During Place and Route step, TNS indicates the negative slack sum for all paths that fail any time constraint. TNS (total negative slack) is the sum of all negative slacks, whereas WNS is slack of the critical path. WNS slack can be negative, zero or positive, so by obtaining 2.106 ns we demonstrate that the HW_nMPRA_RTOS project is implementable in FPGA. Thus, obtaining and analyzing the values for WNS, WHS, and WPWS demonstrates that the project meets the time constraints at 33 MHz CPU frequency. WHS is the worst hold slack, which is the worst negative hold value (0.056 ns). Thus, FPGAs have the clock shaft already seated, so on a single clock design designers do not see intra-FF hold.

The nHSE_EN_sCPUi selector represents the activation command for instPi based on the attached events. When the scheduler activates the instPi hardware instance, the nHSE block diagram simultaneously decodes the mrPRIsCPUi, crTRi, crEVi, and grINT_IDi registers. Thus, only one instPi can be in the RUN state. This is possible through the scheduler FSM, task states, and event block, as well as the synchronization signals. The jitter for setting the LED is only nine clock cycles, a period necessary to identify the external interrupt, activate the instPi to which the interrupt is attached, and execute the instructions related to the change in the logical level corresponding to the digital output FPGA circuit. For the WCET calculation, the deadlines for RTS with hard constraints were considered. In HW_nMPRA_RTOS, events are statically attached to instPi and inherit its priority (mrPRIsCPUi), which can be set dynamically by instP0. It is worth mentioning that the CPU pipeline does not contain valid data, so the instructions extracted from the address indicated by the trap cell associated with the interrupt event ExtIntEv[0] must pass through the fetch, decode, execute, memory, and write back pipeline stages. The grINT_PR global register implemented in the nHSE scheduler stores the highest-priority interrupt ID, which is updated by an interrupt priority encoder. This priority register makes it possible to perform dynamic scheduling in software, e.g., the EDF algorithm can be implemented for thread scheduling. Pipeline registers are multiplied to store instruction execution information for each thread. The scheduler and the CPU datapath provide, through the instructions CFC2 (copy control word from COP2), CTC2 (copy control word to COP2), MFC2 (move word from COP2) and MTC2 (move word to COP2), the writing and reading of the scheduler registers, which are mapped to the COP2 address space. Thus, the architecture was intended to be as deterministic as possible, and implementing a cache level could introduce an unpredictability factor in terms of the RTS response time.

In the process of designing the HW_nMPRA_RTOS, designers had to verify the timing report and where this hold violation is happening and see if it can be resolved by an IO timing constraint or by moving the IO register to the IO Pad or some other way. Thus, THS is total hold slack, which is the cumulative negative hold value. The worst of all of the pulse width violations (1.100 ns for HW_nMPRA_RTOS) are reported as the Worst Pulse Width Slack (WPWS). Table 5 presents the power report HW_nMPRA_RTOS post-implementation in the FPGA circuit.

C. COMPARISON BETWEEN DIFFERENT CPU ARCHITECTURES BASED ON FPGA IMPLEMENTATION

Figure 10 presents the Virtex-7 FPGA resource utilization based on different projects [32], [33], [34], [35], [36], [37], including the HW_nMPRA_RTOS project (4 instPi soft-core only) with a dynamic hardware scheduler. Table 6 shows

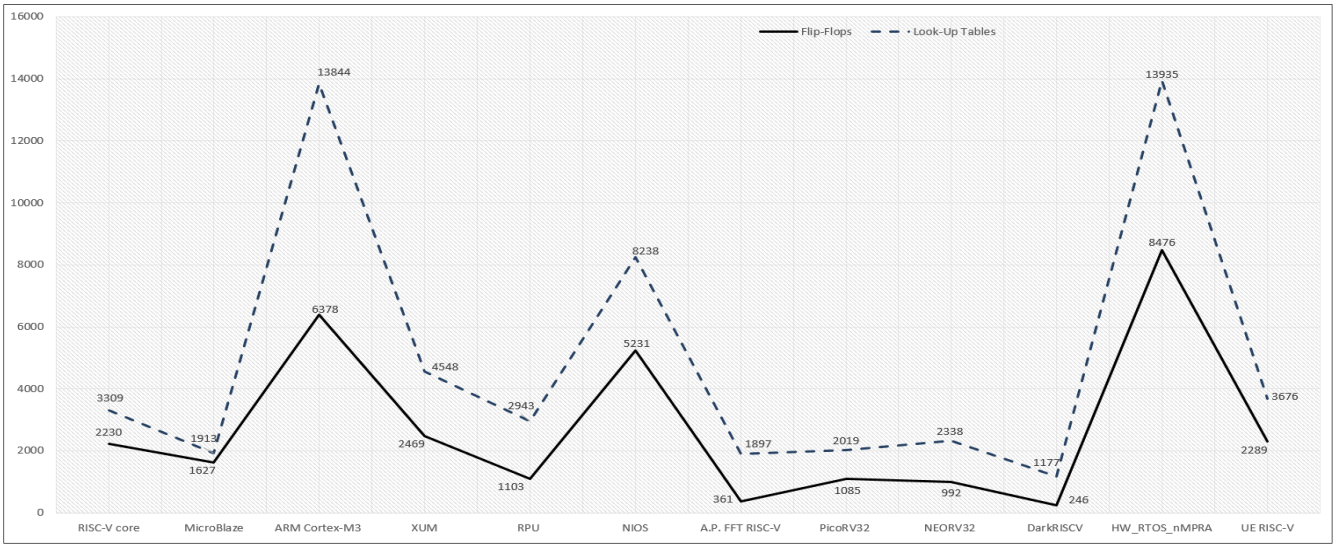

FIGURE 10. FPGA resource requirements comparison among CPU cores (RISC-V core [29], MicroBlaze [31], ARM Cortex-M3 [30], XUM [28], RPU [31], NIOS [32], A.P.FFT.RISC-V [33], PicoRV32 [34], NEOR32 [35], DarkRISC [36], HW_nMPRA_RTOS (soft-core CPU), UE RISC-V [37]).

TABLE 5. Power report post-implementation in FPGA.

On-Chip Power (mW) / CPU	RISC-V core [29]	Cortex-M3 [30]	HW_nMPRA_RTOS	MicroBlaze [31]
Clocks	56	20	17	8
Logic	14	13	3	3
Signals	141	41	9	5
I/O	11	3	4	11
MMCM	128	106	121	106
DSP	4	1	1	2
Device Dynamic	374 (58%)	183 (72%)	161 (39%)	135 (58%)
Device Static	275 (42%)	72 (28%)	252 (61%)	98 (42%)
Total on-chip power	0.649 mW	255 mW	413 mW	233 mW

TABLE 6. Overhead related to RTOS functions and services implementation and other CPU architectures.

RTOS Overhead / CPU	hthreads [38] (µs)	ARPA-MT: OReK-Sw [39] (µs)	HW_nMPRA_RTOS (µs)
Thread context save/restore	0.510	3.200 (72 cycles, 24 MHz)	0.121 (33 MHz)
Task dispatch (scheduler jitter)	1.400	2.800	0.060
Task preemption	-	0.500	0.090
Asynchronous event handling	1.910	2.300	0.268
Boot and Configuration	0.500	39.700	21.800
Task create/destroy	0.100	6.200/7.700	0.151
Periodic task activation and prioritization	0.280	9.200	0.242
Task parameter Read/Write	0.100	2.300	0.151
Mutex / Semaphore Enable/disable	0.030	1.000	0.101
Mutex / Semaphore lock/unlock	0.027/0.0521	2.600	0.212

the overhead results related to RTOS functions and services partially or fully implemented in hardware by the nHSE scheduler and other CPU architectures. The improvement by the difference of the proposed method in the evaluation is given by the thread context save/restore which is only 0.121 us, compared to 0.510 at the hthreads project [38].

The present project relies on both separate hardware resources for each thread and a priority-based preemptive scheduler. Jitter for synchronous event handling is only 0.268 us at HW_nMPRA_RTOS compared to 1.910 us at hthreads and 2.300 us at ARPA-MT [39]. This improvement is mainly because in HW_nMPRA_RTOS, the interrupt event borrows the priority of the instPi to which it is attached, and the handler useful code is executed directly, without searching through software tests for the source that issued the request. The same hardware implementation idea for processing structures as presented in this paper can be found in [23], [38], [39]. Many projects choose to integrate and implement in FPGA established architectures such as MIPS32 [28], HW_nMPRA_RTOS, new architectures such as RISC-V [24], [29], [33], [34], [35], [36], [37], architectures with the most produced ISAs such as ARM [30], or specific architectures of FPGA circuit manufacturers such as MicroBlaze [31], NIOS [32].

In the specialized literature, numerous highly established studies are proposed that implement some features of RTOS in hardware and that are designed for time-predictable systems, such as MERASA [40], PRET [41], and Patmos [22], [42]. Advantages that can be newly realized in this study are of conceptual and practical nature. It can be stated that a new algorithm has been proposed for the nHSE dynamic scheduler that selects which instPi is in the ready state, and generate a message-type event based on SMRF. At the implementation level, the project has been successfully synthesized in Vivado DS and the jitter afferent to the events

associated with the RTOS functions and the context switching times have been measured with the oscilloscope. To perform the appropriate performance measurements, some reg-type spies were introduced into the Verilog HDL software. These flip-flops change the state of some internal wire-type signals, which are then connected to the pins of the Virtex-7 FPGA circuit. Following the presentation, the description of the HW_nMPRA_RTOS architecture, and the analysis of processor architectures with functions implemented in the hardware we can deduce the following new findings associated with the proposed paper. First, the nHSE preemptive scheduler has been implemented in hardware (Figure 4) facilitating practical testing of the HW_nMPRA_RTOS architecture using programmable logic technology, which enhances the state-of-the-art scheduler in the field. Thus, the nHSE unit registers were reorganized to interconnect with the HW_nMPRA_RTOS datapath and the control unit was modeled to meet the requirements of the nHSE scheduler based on COP2 instructions. As a derivative scientific contribution, the mechanism for handling interrupts, mutexes, and messages (Figure 5, Algorithm 2) between processor instances (instPi) with private hardware context was integrated into nHSE (Table 6).

From a safety-critical application point of view, the HW_nMPRA_RTOS architecture represents an innovative, forward-looking, low-cost solution (including RTOS) with better performance than existing systems in automotive, robotics, medical, motion control, Building Internet of Things (BIoT), and industrial process control.

V. CONCLUSION

In this paper, the validation of the HW_nMPRA_RTOS architecture and the SoC design used for testing the implementation proposed in this project was presented. By means of practical results obtained using the Vivado and the Virtex-7 FPGA circuit, the practical implementation of the theoretical approaches presented in Chapter III has been demonstrated. The elements of difficulty of this research appear at the HW_RTOS level, due to embedding its functionality in hardware and highlighting performance improvement through appropriate test programs. The amount of improvement resulting from a quantitative evaluation is about three times better both in terms of response time to external events and switching thread contexts jitter. This is due to separate hardware contexts (HW_thread_i) for each thread (instPi), an implemented FSM for nHSE, the use of CAM for RTOS message destination lookup, and the realization of a priority encoder for interrupts. The relevant performances, required in critical RTS, are represented by the memory consumption for the multiplication in the hardware of the multiplexed datapath resources. This factor depends on thread hardware instances implemented, their number being in the range of 4-16 because the architecture validated in this paper is proposed for small-embedded applications in the industrial, IoT, medical, or automotive sectors.

The implementation proposed in this paper comes with practical results explicitly validated for MIPS32 and tested on the Virtex-7 FPGA development kit produced by Xilinx-AMD. Thus, we can consider that the scientific and practical contribution brought by the authors through this paper is a significant one since the improvement brought to the management scheme of asynchronous external events is an essential one in terms of real-time embedded systems.

Given the application domains and range of capabilities of the nMPRA concept, the economic impact is significant, materializing in the form of a predictable processor and the elimination of costly WCET determination, testing, and certification. The authors plan in the future to implement a multicore processor architecture with RISC-V ISA, where instPi are dynamically associated with pipeline stages from cores, and integrate the HW_nMPRA_RTOS concept into a set of BIoT-based smart switches. These concepts are also complex approaches, generating scalable, extensible architectures with wide applicability in practice.

PATENTS

The central processing unit with pipeline registers is patented in Germany, Munich (DE202012104250U1).

REFERENCES

[1] G. Butazzo and C. Buttazzo, *Hard Real-Time Computing Systems Predictable Scheduling Algorithms and Applications*, 3rd ed. Cham, Switzerland: Springer, 2011.

[2] E. Antolak and A. Pulka, "Energy-efficient task scheduling in design of multithread time predictable real-time systems," *IEEE Access*, vol. 9, pp. 121111–121127, 2021, doi: 10.1109/ACCESS.2021.3108912.

[3] *Issues With Real Time Performance in Conventional RTOS and Performance Improvements through HW-RTOS*, document R70WP0002EJ0100, HW-RTOS, Real Time OS in Hardware, RENESAS, Tokyo, Japan, Sep. 2018.

[4] J. J. Labrosse, *µC/OS-II the Real Time Kernel*, 2nd ed. Oxfordshire, U.K.: Taylor & Francis, 2002.

[5] M. Naghibzadeh, *Contemporary Operating Systems*. Chicago, IL, USA: Independently Published, 2022.

[6] J. Echague, I. Ripoll, and A. Crespo, "Hard real-time preemptively scheduling with high context switch cost," in *Proc. 7th Euromicro Workshop Real-Time Syst.*, Jun. 1995, pp. 14–16, doi: 10.1109/EMWRTS.1995.514310.

[7] V. G. Gaitan, N. C. Gaitan, and I. Ungurean, "CPU architecture based on a hardware scheduler and independent pipeline registers," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 9, pp. 1661–1674, Sep. 2015.

[8] E. Dodiu and V. G. Gaitan, "Central processing unit with combined into a bank pipeline registers," DE Grant DE 202 012 104 250 U1, Feb. 25, 2013.

[9] A. Ghavidel, Y. Sedaghat, and M. Naghibzadeh, "Hybrid scheduling to enhance reliability of real-time tasks running on reconfigurable devices," *J. Supercomput.*, vol. 76, no. 6, pp. 4701–4730, Jun. 2020, doi: 10.1007/s11227-019-02976-6.

[10] F. Reghenzani, Z. Guo, L. Santinelli, and W. Fornaciari, "A mixed-criticality approach to fault tolerance: Integrating schedulability and failure requirements," in *Proc. IEEE 28th Real-Time Embedded Technol. Appl. Symp. (RTAS)*, May 2022, pp. 27–39, doi: 10.1109/RTAS54340.2022.00011.

[11] L. P. Dantas, R. J. de Azevedo, and S. P. Gimenez, "A novel processor architecture with a hardware microkernel to improve the performance of task-based systems," *IEEE Embedded Syst. Lett.*, vol. 11, no. 2, pp. 46–49, Jun. 2019, doi: 10.1109/LES.2018.2864094.

[12] R. Tabish, R. Mancuso, and S. Wasly, "A real-time scratchpad-centric OS with predictable inter/intra-core communication for multi-core embedded systems," *Real-Time Syst.*, vol. 55, pp. 850–888, Oct. 2019, doi: 10.1007/s11241-019-09340-0.

[13] P. Choi, W. Kong, J.-H. Kim, M.-K. Lee, and D. K. Kim, "Architectural supports for block ciphers in a RISC CPU core by instruction overloading," *IEEE Trans. Comput.*, vol. 71, no. 11, pp. 2844–2857, Nov. 2022, doi: 10.1109/TC.2021.3050515.
[14] C. Leng, Y. Qiao, X. S. Hu, and H. Wang, "Co-scheduling aperiodic real-time tasks with end-to-end firm and soft deadlines in two-stage systems," *Real-Time Syst.*, vol. 56, no. 4, pp. 391–451, Oct. 2020, doi: 10.1007/s11241-020-09352-1.
[15] L. Abeni and T. Cucinotta, "EDF scheduling of real-time tasks on multiple cores: Adaptive partitioning vs. Global scheduling," *ACM SIGAPP Appl. Comput. Rev.*, vol. 20, no. 2, pp. 5–18, Jul. 2020, doi: 10.1145/3412816.3412817.
[16] S. Pei, M.-S. Kim, and J.-L. Gaudiot, "Extending Amdahl's law for heterogeneous multicore processor with consideration of the overhead of data preparation," *IEEE Embedded Syst. Lett.*, vol. 8, no. 1, pp. 26–29, Mar. 2016, doi: 10.1109/LES.2016.2519521.
[17] S. Tang, S. Voronov, and J. H. Anderson, "Extending EDF for soft real-time scheduling on unrelated multiprocessors," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, Dec. 2021, pp. 253–265, doi: 10.1109/RTSS52674.2021.00032.
[18] A. Coluccio, "Hybrid-SIMD: A modular and reconfigurable approach to beyond von Neumann computing," *IEEE Trans. Comput.*, vol. 71, no. 9, pp. 2287–2299, Sep. 2022, doi: 10.1109/TC.2021.3127354.
[19] M. Andreozzi, G. Gabrielli, B. Venu, and G. Travaglini, "Industrial challenge 2022: A high-performance real-time case study on arm," in *Proc. 34th Euromicro Conf. Real-Time Syst. (ECRTS)*, vol. 231, 2022, p. 470, doi: 10.4230/LIPIcs.ECRTS.2022.1.
[20] W. Kang, K. Lee, J. Lee, I. Shin, and H. S. Chwa, "LaLaRAND: Flexible layer-by-layer CPU/GPU scheduling for real-time DNN tasks," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, Dec. 2021, pp. 329–341, doi: 10.1109/RTSS52674.2021.00038.
[21] T. Cucinotta and L. Abeni, "Migrating constant bandwidth servers on multi-cores," in *Proc. 29th Int. Conf. Real-Time Netw. Syst.*, Apr. 2021, pp. 155–164, doi: 10.1145/3453417.3453441.
[22] M. Schoeberl, W. Puffitsch, S. Hepp, B. Huber, and D. Prokesch, "Patmos: A time-predictable microprocessor," *Real-Time Syst.*, vol. 54, no. 2, pp. 389–423, Apr. 2018, doi: 10.1007/s11241-018-9300-4.
[23] R. Pan and G. Parmer, "SBIs: Application access to safe, baremetal interrupt latencies," in *Proc. IEEE 28th Real-Time Embedded Technol. Appl. Symp. (RTAS)*, May 2022, pp. 82–94, doi: 10.1109/RTAS54340.2022.00015.
[24] A. Gruin, T. Carle, H. Casse, and C. Rochange, "Speculative execution and timing predictability in an open source RISC-V core," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, Dec. 2021, pp. 393–404, doi: 10.1109/RTSS52674.2021.00043.
[25] D. M. Platzer and P. Puschner, "Vicuna: A timing-predictable RISC-V vector coprocessor for scalable parallel computation," in *Proc. 33rd Euromicro Conf. Real-Time Syst. (ECRTS)*, 2021, p. 370, doi: 10.4230/LIPIcs.ECRTS.2021.1.
[26] I. Zagan and V. G. Găitan, "Designing a custom CPU architecture based on hardware RTOS and dynamic preemptive scheduler," *Mathematics*, vol. 10, no. 15, p. 2637, Jul. 2022, doi: 10.3390/math10152637.
[27] V. G. Găitan and I. Zagan, "An overview of the nMPRA and nHSE microarchitectures for real-time applications," *Sensors*, vol. 21, p. 4500, Jun. 2021, doi: 10.3390/s21134500.
[28] G. Ayers. (2011). *eXtensible Utah Multicore (XUM) Project at the University of Utah*. Accessed: May 2021. [Online]. Available: http://opencores.org/project,mips32r1
[29] *uRV Core*. Accessed: Feb. 15, 2021. [Online]. Available: https://github.com/twlostow/urv-core
[30] J. Yiu, *System-on-Chip Design With Arm Cortex-M Processors*. Cambridge, U.K.: Arm Education Media, 2019.
[31] *RPU*. Accessed: Sep. 2022. [Online]. Available: https://github.com/Domipheus/RPU
[32] W.-T. Zhang, L.-F. Geng, D.-L. Zhang, G.-M. Du, M.-L. Gao, W. Zhang, N. Hou, and Y.-H. Tang, "Design of heterogeneous MPSoC on FPGA," in *Proc. 7th Int. Conf. ASIC*, Oct. 2007, pp. 102–105, doi: 10.1109/ICASIC.2007.4415577.
[33] Z. Zheng, X. Zhu, and H. Qian, "Design and implementation of arbitrary point FFT based on RISC-V SoC," in *Proc. IEEE 5th Adv. Inf. Technol., Electron. Autom. Control Conf. (IAEAC)*, Mar. 2021, pp. 1216–1219, doi: 10.1109/IAEAC50856.2021.9390787.
[34] *PicoRV32 A Size-Optimized RISC-V CPU*. Accessed: Sep. 2022. [Online]. Available: https://github.com/cliffordwolf/picorv32
[35] *The NEORV32 Processor*. Accessed: Sep. 2022. [Online]. Available: https://github.com/stnolting/neorv32
[36] *DarkRISCV*. Accessed: Sep. 2022. [Online]. Available: https://github.com/darklife/darkriscv
[37] *UE RISC-V*. Accessed: Sep. 2022. [Online]. Available: https://github.com/ultraembedded/riscv
[38] J. Agron and D. Andrews, "Hardware microkernels for heterogeneous manycore systems," in *Proc. Int. Conf. Parallel Process. Workshops*, Sep. 2009, pp. 19–26, doi: 10.1109/ICPPW.2009.21.
[39] A. S. R. Oliveira, L. Almeida, and A. D. B. Ferrari, "The ARPA-MT embedded SMT processor and its RTOS hardware accelerator," *IEEE Trans. Ind. Electron.*, vol. 58, no. 3, pp. 890–904, Mar. 2011, doi: 10.1109/TIE.2009.2028359.
[40] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, J. Wolf, H. Casse, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische, "Merasa: Multicore execution of hard real-time applications supporting analyzability," *IEEE Micro*, vol. 30, no. 5, pp. 66–75, Sep. 2010, doi: 10.1109/MM.2010.78.
[41] S. A. Edwards and E. A. Lee, "The case for the precision timed (PRET) machine," in *Proc. 44th ACM/IEEE Design Autom. Conf.*, Jun. 2007, pp. 264–265, doi: 10.1109/DAC.2007.375165.
[42] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, and C. W. Probst, "Towards a time-predictable dual-issue microprocessor: The Patmos approach," in *Bringing Theory to Practice: Predictability and Performance in Embedded Systems* (OpenAccess Series in Informatics), vol. 18, P. Lucas, L. Thiele, B. Triquet, T. Ungerer, and R. Wilhelm, Eds. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011, pp. 11–21. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2011/3077, doi: 10.4230/OASIcs.PPES.2011.11.

IONEL ZAGAN (Member, IEEE) received the M.Sc. degree in computer science from the Stefan cel Mare University of Suceava, Suceava, Romania, in 2005, where he is currently pursuing the Ph.D. degree in engineering.

He is a Lecturer with the Department of Computers, Stefan cel Mare University of Suceava. His research interests include real-time systems, field-programmable gate arrays, microcontrollers, and pipeline processors with parallel execution of tasks. He is a member of the IEEE Computer Society.

VASILE GHEORGHIȚĂ GĂITAN (Member, IEEE) received the M.Sc. and Ph.D. degrees from the Gheorghe Asachi Technical University of Iaşi, Iaşi, Romania, in 1984 and 1997, respectively.

He is currently a Professor with the Department of Computers, Stefan cel Mare University of Suceava, Suceava, Romania. His main research interests include real-time scheduling, embedded middleware, digital systems design with field-programmable gate arrays, fieldbuses, and embedded system applications. He is a member of the IEEE Computer Society.