Hindawi
Wireless Communications and Mobile Computing
Volume 2022, Article ID 5705184, 15 pages
https://doi.org/10.1155/2022/5705184

Research Article

Embedded Microprocessor Extension Design and Optimization for Real-Time Edge Computing

Yubo Wang,[1,2] Xiujia Zhao,[3,4] Ting Chong,[2] and Xianhua Liu[3,4]

[1]*North China Electric Power University, Beijing, China*
[2]*Beijing Smart-Chip Microelectronics Technology Co., Ltd., Beijing, China*
[3]*Department of Computer Science & Technology, Peking University, Beijing, China*
[4]*Engineering Research Center of Microprocessor & System, Peking University, China*

Correspondence should be addressed to Xianhua Liu; liuxianhua@mprc.pku.edu.cn

Received 17 January 2022; Revised 10 February 2022; Accepted 11 February 2022; Published 11 March 2022

Academic Editor: Mohammad Farukh Hashmi

With the development of 5G communication technology, more and more applications could be integrated into one system. Edge computing system and mixed-criticality system may integrate tasks of different criticality levels, which brings better balance in isolation and performance. Such advantages make it gradually become a research hotspot in edge computing and real-time systems with 5G. The important content of designing a mixed-criticality system is how to reduce interference between tasks and how to schedule tasks efficiently to ensure that tasks of different criticality levels can meet time constraints. Instruction extension and hardware software cooperative support may be an effective solution. Based on a fine-grained multithreaded RISC-V processor, this article gives some extensions for real-time operations and proposes a hardware software cooperative real-time scheduling mechanism. Experimental results show that, compared with FlexPRET hardware, the performance of thread scheduling is improved by 22.94% on average. Compared with software scheduling, the performance of scheduling same programs and multiple programs are improved by 15.46% and 26.00%, respectively.

1. Introduction

With the development 5G communication technology, people are more and more inclined to integrate increasingly complex software tasks into a hardware platform to reduce the hardware cost and the size, weight, and power consumption of the system. This requires a single processor to run tasks with different importance or security, resulting in the introduction of mixed criticality system (MCS) [1]. Mixed-criticality system has been widely used in avionics, automobile, industrial automation, medical, and other 5G fields nowadays, which has strong requirements for real-time performance and becoming an emerging design paradigm to improve the resource-efficiency for real-time embedded system. The correctness of real-time system behavior depends not only on the logical result of calculation but also on the time when the result is generated. Compared with real-time systems, the correctness of general computer systems only depends on the input and output of the system, and the timing behavior of programs is only a measure of system performance. The design and implementation of real-time systems are very different from nonreal-time processor design technologies used to improve the average performance of the system, such as branch prediction, cache, and speculative execution. Those improved techniques cannot be simply applied to real-time systems, because these technologies cannot guarantee the execution time of programs in the worst case (worst case execution time, WCET), which damages the real-time performance of the system [2]. The challenge is that each task in the system may require different criticality levels. The criticality level here refers to the assurance level necessary to prevent task or component failure. In different application fields, the number and definition of criticality levels are different, but at least two levels can be defined, usually defined as safety critical and noncritical [3]. For example, in automatic driving standard ISO 26262 [4], the system critical level is defined as four safety levels. Standard DO-178C [5] defines five design assurance

levels (DALs) with different definitions and requirements: (a) catastrophic, (b) dangerous, (c) primary, (d) secondary, and (e) no security impact. Software standard ARINC 653 defines how real-time operating system (RTOS) should separate time and space resources for tasks at different critical levels.

However, the traditional mixed-criticality system is commonly designed under pessimistic constraints on overrun handling and slack utilization. These solutions cannot provide efficient resource utilization when considering complex behaviors of mixed criticality systems [6]. Tasks at different critical levels run on the same hardware platform and share processor time, memory, network, and other resources. For mixed criticality systems, a highly integrated hardware platform needs to ensure the flexibility and efficiency of resource sharing for tasks at different critical levels, meet their respective critical levels, and reduce the interference between tasks caused by resource sharing. Isolation between tasks is very important. If the system ensures isolation, multiple tasks can be verified separately and then integrated into the same system, which is important for modular system design [7]. In the 5G-based IoT applications on the multiprocessor platform, the partitioned scheduling has been widely applied. However, these partitioned scheduling approaches could cause system resource waste and uneven workload among processors [8]. However, how to make a trade-off between isolation and effective utilization of resources is an important issue. The isolation methods include hardware isolation, software isolation, and the combination method. The existing isolation methods still have the problem of insufficient resource utilization.

Vestal [9] put forward the concept of mixed criticality system for the first time, and proposed a fixed priority scheduling method based on preemption. A lot of research work has focused on the scheduling problem of mixed criticality system. However, most of these studies regard the execution time of tasks as a fixed value [10]. In practice, the execution time of a program is affected by many factors, and the worst-case execution time is difficult to define.

For real-time systems, especially mixed criticality systems, not all ILP (instruction level parallelism) or TLP (thread level parallelism) technologies are conducive to the WCET (worst-case execution time) analysis of programs. On the contrary, many optimization technologies based on locality rely on the historical execution information of the program, which makes the WCET difficult to be determined. It requires processor designers to use appropriate technology to make a trade-off between real-time and processor performance under the condition of predictable timing.

FlexPRET [3] (flexible precision timed machine) is a mixed criticality system processor architecture based on RISC-V [11] instruction set architecture proposed by the University of California, Berkeley. It is a part of the precision timed machines project [12]. FlexPRET is a 32-bit 5-stage pipelined fine-grained multithreaded processor written by chisel. It provides a variety of configuration parameters to allow task threads to balance between hardware-based isolation and efficient resource utilization. Although FlexPRET realizes flexible hardware thread scheduling and hardware isolation, its supporting software functions are not perfect. Specifically, the following points need to be improved. First, hardware semantics are not abstracted into software semantics that programmers can easily understand. Second, the existing underlying software functions are not perfect, which require users to write inline assemblies and encapsulate the interfaces. Third, it lacks global consideration of thread scheduling, which not fully utilizes the hardware resources. Finally, its instruction extensions support the running of real-time programs, but there is no way to guarantee WCET for a thread.

2. Related Work

The modeling of a typical mixed criticality system [13] could be described as below. The system consists of a limited set of components, each component contains a critical level L, and a limited set of sporadic tasks. Each task Ti can be defined as a quadruple $Ti = \{Til, Di, Cil, Li\}$, where Til represents the minimum period of Ti under the critical level L, and Di represents the deadline of the task Ti, Cil is the execution time of Ti under the critical level L, and Li represents the critical level. For $\{Til\}$ and $\{Cil\}$, there are as follows.

$$L_1 > L_2 \Longrightarrow T_{l1} \leq T_{l2}\ L_1 > L_2 \Longrightarrow C_{l1} \geq C_{l2} \tag{1}$$

In other words, the higher critical levels, the smaller the task cycle and the longer the worst-case execution time. For higher critical levels, a more conservative verification process is required, and the task execution time will be longer, which is observed in the practical application of Vestal's article [8]. For higher critical levels, more events need to be processed, so the task needs to run in a shorter cycle to meet the needs, which is also discussed in many research articles [14–16].

2.1. Scheduling of Tasks/Threads. Mixed criticality system scheduling methods for single processor can be roughly divided into fixed priority scheduling and dynamic priority scheduling. Fixed priority scheduling can be based on response time analysis (RTA), slack scheduling, or period transformation. Dynamic priority scheduling is mainly based on the idea of earliest deadline first (EDF).

2.1.1. Scheduling Based on Response Time Analysis. The key idea of the scheduling strategy is to assign appropriate priority to tasks based on task response time. Vestal gives a fixed priority scheduling scheme for mixed criticality systems for the first time based on the optimal priority assignment (OPA). Baruah's research further extended Vestal's method. It proposed a scheduling model on a single processor and an associated analysis framework for two key level systems and proposed a response time analysis method for adaptive mixed criticality, which calculates the response time for different situations, further improving resource utilization.

2.1.2. Slack Time Scheduling. Niz et al. [17] proposed the method of slack time scheduling, so that low critical level tasks can be executed in the unused idle time of high critical level tasks. The disadvantage is that it is difficult to recover

the idle time of high-level critical tasks for aperiodic tasks. At the same time, this method may cause high-level critical tasks to exceed the time limit as well when low-level tasks exceed the time limit [18]. It abandons low-level critical tasks when they exceed the time limit.

2.1.3. Period Transformation Scheduling. In Vestal's work, it is also proposed that period transformation protocol can also be used to schedule mixed criticality systems. The period transformation protocol divides the high critical level tasks into N copies, and the execution time and cycle of the converted tasks become $1/N$ of the original. Then, rate monotonic scheduling is applied to improve the relative priority of high critical tasks. This method improves the schedulability as well as the overhead of the system.

2.1.4. EDF-Based Scheduling. Vestal and Baruah [19] first introduced the EDF scheduling method into the scheduling analysis of mixed criticality systems and specified the key level tasks with the earliest deadline by dynamically specifying priority. Most of the subsequent research work is based on the EDF algorithm.

For multicore processors, Anderson et al. [20] first discussed the scheduling problem of mixed criticality systems on multicore platforms. Most of the subsequent work focuses on task division, schedulability analysis, resource allocation, etc.

2.2. Isolation of Tasks and Threads. Isolation is another important method to reduce the interference caused by resource sharing between tasks. Isolation strategies may include hardware isolation, software isolation, and hybrid implementation.

2.2.1. Hardware Isolation. Hardware isolation includes isolation based on multicore processor and multithreaded processor. Proteus [21], SPUMON [22], and RGMP [23] are typical homogeneous multicore isolations. It achieves isolation by running different operating systems on different cores, which requires more resources. Heterogeneous multicore isolation can be improved in energy efficiency, and its overall architecture is similar to the above solution. Such schemes need more than two heterogeneous computing cores to ensure the effective response to real-time tasks, but the isolation of hardware and software makes the resource utilization not high.

The isolation based on multithreaded processor includes FlexPRET [3], PTARM [24], and XMOS [25]. In multithreaded processors, tasks run in different hardware threads and have different thread states such as registers. Therefore, the spatial isolation of multithreaded processor is natural. However, its timing isolation needs the thread scheduling algorithm to ensure. PTARM uses a fixed round-robin scheduling algorithm, and each thread occupies a fixed time slice, even if the thread is in sleep state. XMOS uses an active round robin scheduling algorithm to schedule active hardware threads. Such algorithms cannot achieve complete time isolation, because the scheduling frequency of threads depends on the number of currently active threads.

2.2.2. Software Isolation. Software isolation may be implemented by virtualization or by real-time operating systems. Typical virtualization schemes include VLX [26], OKL4 Microvisor [27], and Xvisor [28]. Since the hypervisor of virtual machine consumes much resources, some studies have proposed lightweight virtualization, which directly runs the real-time operating system on the processor, while the general operating system runs as a task on RTOS. Typical schemes include RTAI [29], RTLinux [30], and Xenomai [31].

In addition to the above two software isolation methods, some scholars propose to realize a certain degree of isolation based on the modification of the real-time operating system. Kron OS [32] achieves isolation by controlling the execution of repeated frame sequences. HIPPEROS [33] is a multicore RTOS used in the field of avionics. It uses one core to make all scheduling decisions and uses an elastic task model to degrade low key tasks in time exceeding scenarios.

2.2.3. Hybrid Mechanism. The combination of software and hardware refers to adding a new mode to the hardware, providing corresponding software support, and using the combination of software and hardware to solve isolation and resource scheduling, such as TrustZone [34], ViMoExpress [35], ARMithril [36], and RTZVisor [37]. It has become a promising research direction in real-time area. Such designs may have a negative impact when computing resources are insufficient. In terms of isolation and performance, more careful consideration must be given to the design.

2.3. Real-Time Semantics in Instruction Set Architecture. Instruction set architecture (ISA) describes the abstract interface of hardware and software systems. For real-time systems and mixed criticality systems, general ISA does not provide instructions which describe the timing behavior of programs. It is large because the constraints on timing behavior depend on the design of processor microstructure and the implementation details of compiler. Hardware models that accurately describe timing behavior are usually unavailable and difficult to implement [4]. Nevertheless, the program can still specify timing behavior. Many processors are equipped with programmable timers. By counting the interrupt caused by the timer, the software gets the timing information, so as to control the timing of the program.

Many programming languages contain the characteristics of timing control. Ada [38] is a programming language originally designed for embedded and real-time computing. The delay keyword is provided in Ada, which can block the running of tasks until a certain time is reached. Real-time Euclid [39] is a programming language specially used to solve reliability and schedulability problems in real-time systems. It stipulates that each structure in the language needs time or space constraints to ensure the schedulability of threads.

Similar to programming language, the realization of timing control in instruction set also needs the support of processor and compiler. The extended timing instructions have a lower level than the programming language, which can provide guidance for hardware design and realize more accurate real-time semantics. Ip and Edwards [40] proposed

a timing instruction to specify the shortest execution time of a piece of code. Bui et al. [41] further proposed four temporal semantics that can be expressed at the instruction level. Liu reduced the first five of the six instructions into four instructions in his dissertation [42] and implemented them in PTARM processor. Antolak and Pulka [43] extended the instruction set and added four timing instructions to a multicore timing predictable processor. Broman et al. [44] introduced their extension of temporal semantics based on llvm.

3. Design and Implementation

This article takes the FlexPRET processor as a reference model for implementation. FlexPRET is a 32-bit fine-grained multithreaded processor based on the RISC-V instruction set. It provides basic 32-bit RV32I and 64-bit RV64I instructions. Other instructions are provided under the standard instruction extension framework, providing a flexible choice space for processor designers. FlexPRET adopts the classical five-stage pipeline design, and its structure is shown in Figure 1.

3.1. Hardware Design and Implementation. As a processor designed for mixed criticality systems, FlexPRET improves the timing predictability of the processor in various ways to support the calculation of WCET and serve real-time programs.

Firstly, FlexPRET supports fine-grained multithreading. Hardware prediction mechanisms such as branch prediction and cache hierarchy make it difficult to accurately predict the execution time of instructions, and removing these prediction mechanisms will damage the performance of the processor. The design of fine-grained multithreading can alleviate these problems to a certain extent. Since fine-grained multithreaded processors can switch threads in each clock cycle, when the pipeline of traditional single-threaded processors needs to stall due to misprediction, the pipeline of fine-grained multithreaded processors can execute the instructions of another thread, so as to improve the overall throughput. Based on this idea, the branch prediction strategy of FlexPRET is to predict that the branch does not taken, that is, the hardware needs not to make additional changes to the branch prediction. Secondly, FlexPRET uses scratch-pad memory instead of cache, so that the delay of memory access instructions will be fixed. It should be noted that the load instruction may have a data hazard with the next instruction. At this time, the pipeline needs to stall for one cycle. In order to simplify the hazard detection and bypass logic, it does not detect the hazard of load instruction, but sets the execution time of the load instruction to two cycles. For fine-grained multithreaded processors, as long as the number of active threads is greater than or equal to two cycles, the delay of load instructions will be hidden. Finally, it adopts thread-independent processing method for interrupt signals, that is, each hardware thread has a separate interrupt signal. When one thread processes an interrupt, it does not affect the timing of other threads.

Although the design of fine-grained multithreading increases the time consumption of a single thread, its predictable timing is very helpful for WCET analysis. Under the condition of fixed scheduling frequency, the number of cycles required for instruction execution is shown in Table 1. Programmers can get the WCET of the program according to the table and combine it with the WCET analysis tool, so as to guide the management of the real-time system.

The hardware thread scheduling pseudocode is shown in Pseudocode 1, with minor modifications from FlexPRET. It adds two control and state registers (CSR) to support the hardware scheduling of threads. *CSR_tmode* stores the statuses (active or sleeping) and attributes (hardware real-time thread, HRTT, or software real-time thread, SRTT) of all hardware threads. There are four kinds of values: HA (active HRTT), SA (active SRTT), HZ (sleeping HRTT), and SZ (sleeping SRTT). *CSR_slot* saves the sequence of threads that the hardware scheduler should schedule. *CSR_slot* divides the 32-bit register into eight 4-bit thread slots. Each thread slot can be specified as a thread number (representing that the thread slot is dedicated to the specified thread).

When the hardware scheduler decides to schedule the thread in the next cycle, it first schedules the thread according to the previous cycle to find the next enabled thread slot. If the thread number is in the thread slot, and the corresponding thread state is active, the corresponding thread is scheduled. If the thread is inactive or the thread slot is dedicated to SRTT, the next SRTT that should be scheduled is found according to the last scheduled SRTT. Generally speaking, the hardware scheduler will cycle through the thread slot in *CSR_slot* to schedule the corresponding threads. When the thread in the thread slot sleeps or the thread slot is specially prepared for SRTT, the SRTT will be schedule in sequence. This scheduling method not only ensures that the HRTT is scheduled at a constant frequency but also utilizes the idle period to schedule the SRTT to improve the throughput.

Different from the previous work, this article adds several extended instructions, which are listed as Table 2. It can express four fundamental temporal semantics proposed by Bui et al. [41] at the instruction set architecture level. Similar with FlexPRET, *set_compare* instruction to set the timing, which may cooperate with *delay_until* and *interrupt_on_ expire*. It also adds *wait_until* instruction which enriches the operations of the threads when the interrupt arrives. Instruction *interrupt_on_expire* is also introduced to trigger soft interrupts for normal program behavior. What emerged as another improvement is the introduction of *mt/fd* instruction pair. It is used to ensure that the code block execution does not exceed the specified time. This constrain is not guaranteed at runtime, but can be remedied in the compilation stage through static analysis. Compiler can analyze the possible execution time of the code block in advance and then check whether the code block meets the time limit. In the hardware, it adds registers *CSR_Clock* and *CSR_Compare* to store the current time and timing time, respectively.

3.2. Software Design and Implementation. In order to provide a friendly programming interface for real-time system users and reduce the porting cost of experienced users, this

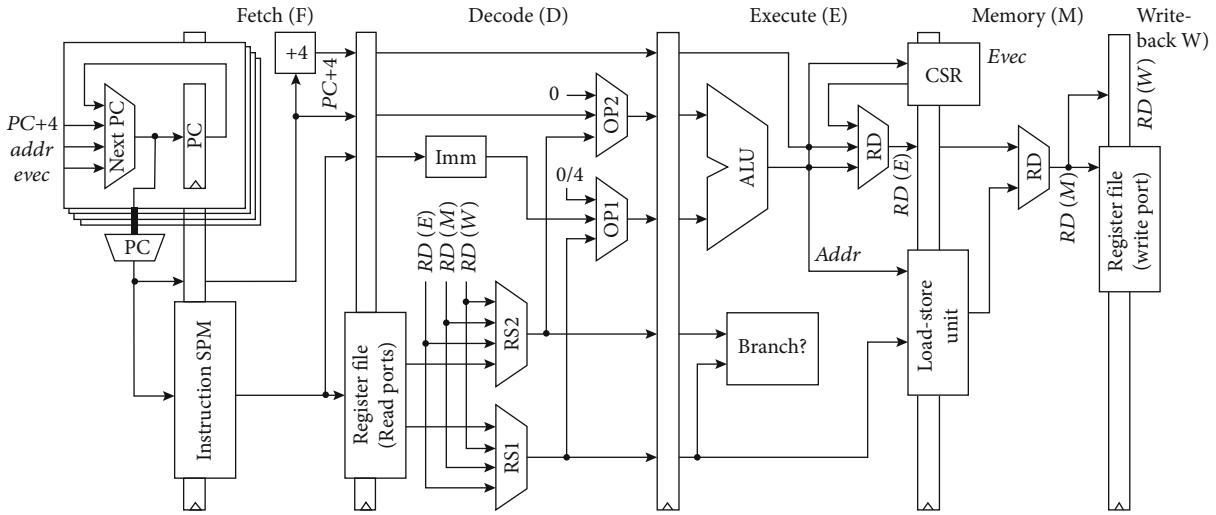

Figure 1: Pipeline structure of FlexPRET processor [3].

Table 1: Cycle count of extended instructions.

RV32I and extended ISAs	Cycles in different frequency f			
	$f = 1$	$f = 1/2$	$f = 1/3$	$f = 1/4$
lui, auipc	1	1	1	1
addi, slti, sltiu, xori, ori, andi, slli, srli, sra	1	1	1	1
add, sub, sll, slt, sltu, xor, srl, sra, or, and	1	1	1	1
jal, jalr	3	2	1	1
beq, bne, blt, bge, bltu, bgeu	1 or 3	1 or 2	1	1
sb, sh, sw	1	1	1	1
lb, lh, lw, lbu, lhu	2	1	1	1
csrrw, csrrs, csrrc, csrrwi,csrrsi, csrrci, sret, fence	1	1	1	1
syscall, fence.i	4	2	2	1
get_time, set_compare	1	1	1	1
delay_until, wait_until	5 or 1	3 or 1	2 or 1	2 or 1
exception_on_expire	1	1	1	1
interrupt_on_expire	1	1	1	1

```
Input: slots_last, tmodes_last
Output: tid, valid, slots_last, tmodes_last
1   valid, s ⟵ NEXT_ENABLED_SLOT(slots_last) // round-robin
2   if valid then
3      slot ⟵ CSR_slot(s)
4      if slot != S and (CSR_tmode(slot) == HA or CSR_tmode(slot) == SA) then
5            tid ⟵ slot
6      else
7         valid, t ⟵ NEXT_ACTIVE_SRTT(tmodes_last)
8         if valid then
9               tid ⟵ t
10            tmodes_last ⟵ t
11        endif
12     endif
13     slots_last ⟵ s
14  endif
```

Pseudocode 1: Pseudocode of hardware thread scheduling.

Table 2: Extended temporal instructions.

Notations	Implementation	Semantics
get_time	csrr rd, CSR_clock	Read CSR_CLOCK to get current time
set_compare	csrw CSR_compare, r1	Cancel the previous timing (if any) and set a new timing
delay_until	custom0 zero, zero, zero, 0	Stall until current time exceeds the value in provided register
wait_until	custom1 zero, zero, zero, 0	Wait until current time exceeds the value in provided register
interrupt_on_expire	custom2 zero, zero, zero, 0	Interrupt program when current time exceeds the value in provided register
exception_on_expire	custom2 zero, zero, zero, 1	Throw exception when current time exceeds the value in provided register
meet_time/register	custom3 0, r1, 0, 0	Label the entry of the code, record the timing limit by value in register
meet_time/immediate	lui zero, imm	Label the entry of the code, record the timing limit by value in immediate field
final_deadline	custom3 zero,zero,zero, 0	Label the end of the code, throw exception when current time exceeds

article selects the general RTOS interface CMSIS-RTOS2 proposed by ARM [45]. CMSIS-RTOS2 defines the standardized RTOS application programming interface and provides many basic functions needed by real-time applications. As the middle layer between the underlying third-party RTOS or other implementations and the upper real-time applications, such interfaces improve the portability of the program and reduce the learning cost.

Based on designed hardware feathers, this article implements the interfaces in the kernel control, thread management, timer management, and mutex management modules defined by CMSIS-RTOS2. At the same time, by using the hardware self-trapping mechanism provided, the processing of exceptions and interrupts is realized. In addition, since the processor adopts the hardware scheduling mode of fine-grained multithreading, the threading model of CMSIS-RTOS2 is quite different. This article provides an extended user-defined interface while being compatible with the CMSIS-RTOS2 threading model as much as possible. It is convenient for programmers to realize richer thread control and scheduling functions through extended interface on the basis of understanding hardware scheduling.

The kernel control module provides an interface for querying kernel information and controlling kernel operation. Table 3 gives the definition and corresponding realization. In CMSIS-RTOS2, the system keeps the system time by ticking count. This method is not accurate enough for time maintenance, which is limited by the frequency of input ticks. The processor maintains the accurate time value of the cycle, and the time accuracy can reach 10 ns at the frequency of 100 MHz. Therefore, the *osKernelGetTickCount()* interface implemented in this article can directly obtain the time value since the system was started, and the unit is nanosecond. At the same time, the *osKernelGetTickFreq()* interface obtains the frequency information preset by the processor. This design allows users to benefit from the precise time control of hardware, independent of the frequency of input beats.

The thread management module provides the most important thread management interface in RTOS. Table 4. gives the relevant definition and realization. Among them, interfaces such as *osThreadNew()*, *osThreadSetpriority()*, *osThreadSuspend()*, *osDelay()*, *osThreadResume()*, *osThread Exit()*, *osThreadTerminate()*, and *osDelayUntil()* will change *CSR_slot* or *CSR_tmode* value of *tmode* to change the scheduling state of the thread. Other interfaces will read information such as thread control blocks to complete the corresponding functions. The content of thread control block in the thread management module realized in this article includes thread entry function address, entry function parameters, thread state, and thread stack address. After the thread is created, the initialization code will read the contents of the thread control block and complete the operation of the thread function. When the thread exits, there is also special code to clean up the thread control block.

In the thread model defined by CMSIS-RTOS2, threads have four states: ready state, blocking state, running state, and terminated state. CMSIS-RTOS2 standard adopts priority-based preemptive task scheduling, which is different from the hardware thread scheduling.

In hardware thread scheduling like FlexPRET, there are only two thread states recorded by hardware: active state and sleeping state, and threads in active state do not always occupy CPU time. In this article, the abstraction of thread state basically follows the definition of FlexPRET. Active state corresponds to ready state and running state in CMSIS-RTOS2, while sleeping state corresponds to blocking state. Terminated state is added for extension, which indicates the state after the thread exits. The state transition of the thread is shown in Figure 2. The solid arrow in the figure indicates the state transition automatically carried out by hardware operation, and the dashed arrow indicates the state transition carried out by software scheduling.

In addition to different thread states, it is also necessary to deal with the priorities of threads. The interface has achieved 48 priorities in this article. Threads with higher priority will seize the running opportunity of the current thread immediately. Hardware defines two key levels, representing hard real-time thread HRTT and soft real-time thread SRTT. In this article, HRTT is mapped to priority of *osPriorityNormal()*, and SRTT is mapped to priority of *osPriorityRealtime()*, which corresponds to the related priority in CMSIS-RTOS2.

Since the hardware processor sets *CSR_slot* to implement the hardware scheduling mechanism. The interface of CMSIS-RTOS2 cannot express relevant semantics of slots. In this article, a set of default slot allocation and recycling principles is established to ensure that users can obtain

Table 3: Interface of accessing kernel information and control modules.

Interface name	CMSIS interface function	Interface implementation in this article
osKernelInitialize()	Initialize kernel	Initialize the default thread control block, timer and mutex, etc.
osKernelGetInfo()	Get kernel version information	Returns the kernel version information string
osKernelGetState()	Get the running state of the kernel	Returns the kernel global state variable
osKernelStart()	Start thread scheduling	Read the existing thread control block status and start thread scheduling
osKernelGetTickCount()	Get system tick count value	Return system time
osKernelGetTickFreq()	Get system tick frequency	Returns the preset frequency

Table 4: Thread management module interface.

Interface name	CMSIS interface function	Interface implementation in this article
osThreadNew()	Create the thread	The thread control block is set by parameters.
osThreadGetName()	Get thread name	Returns the thread name string
osThreadGetId()	Get thread identifier	Returns the thread control block pointer of the corresponding thread
osThreadGetState()	Get thread status	Return thread status
osThreadGetStackSize()	Get thread stack size	Returns the thread stack size in the thread control block
osThreadGetStackSpace()	Get stack free space	Called by the current thread to return the stack space
osThreadSetPriority()	Set thread priority	Set thread priority. 2 priorities, corresponding to HRTT/SRTT, are supported
osThreadGetPriority()	Get thread priority	Return thread priority
osThreadSuspend()	Suspend the thread	Modify CSR_tmode, changes the thread state to suspend
osThreadResume()	Wake up the thread	Modify CSR_tmode, changes the thread state to active
osThreadDetach()	Set the properties to detached	Modify the detached attribute in the thread control block
osThreadJoin()	Wait for the child thread to exit	Check the thread control block of the child to see if it exits
osThreadExit()	Exit the thread	Exit current thread and enter the looping sleep state
osThreadTerminate()	Terminate the thread	Make a thread sleep, but cannot change its control flow
osThreadGetCount()	Get the number of active threads	Returns the number of currently active threads
osThreadEnumerate()	Gets the active thread identifiers	Returns an array of currently active thread identifiers
osDelay()	Thread sleeps for a period	Using set_compare and delay_until instruction to implement the semantics
osDelayUntil()	Thread sleeps to a point in time	Using set_compare and delay_until instruction to implement the semantics

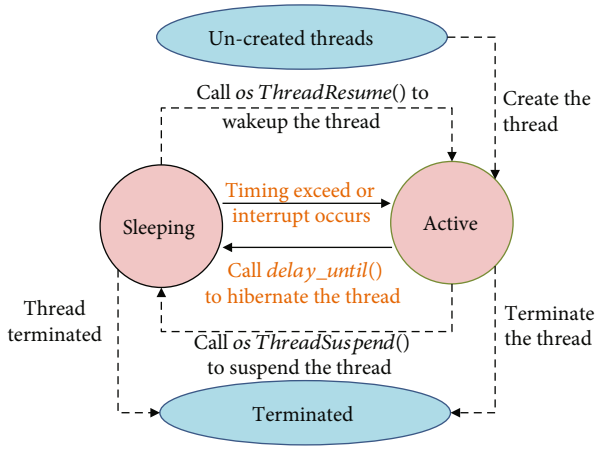

Figure 2: Thread state transition diagram.

better real-time performance when creating threads by default. For users who are familiar with hardware scheduling, this implementation also provides corresponding interfaces to control thread slots more finely, so as to meet the specific needs of users. The default thread slot allocation and recycling principles are as follows.

When creating HRTT with *osPriorityRealtime()*, allocate exclusive thread slots. When using *osPriorityNormal()* to

create SRTT, exclusive thread slots will not be allocated, and one thread slot used by SRTT will be allocated uniformly.

When SRTT modifies priority and changes it to HRTT, if the thread has an exclusive thread slot, no additional thread slot will be allocated, and only *tmode* will be modified. If the thread does not have an exclusive thread slot, allocate a thread slot for it. If this thread was previously the only SRTT, remove the original SRTT-dedicated thread slot (when there is a dedicated SRTT thread slot and there is no active SRTT, the HRTT will not be scheduled, and this thread slot will be wasted).

When HRTT modifies priority and changes it to SRTT, if the thread has no exclusive thread slot, no extra processing will be performed (if it does not exist, it means that the user has specified the thread slot, so there is no interfere). If the thread has an exclusive thread slot, the exclusive thread slot is deleted. If there is no SRTT or SRTT dedicated thread slot, then add an SRTT dedicated thread slot. When thread exits, all exclusive thread slots will be removed. If it is the only SRTT, all SRTT dedicated thread slots will be removed.

This article also makes reasonable transformation of other thread interfaces as follows.

OsThreadGetStackSpace(). Because of the hardware isolation, threads cannot get the current stack frame position of other threads. Therefore, it only supports obtaining the available stack space of the current thread.

OsThreadExit() and *osThreadTerminate()*. These two functions are used to exit the thread. The former is used to exit the thread itself, which is equivalent to the return function of the thread. The latter is to force other threads to exit. According to the characteristics of hardware isolation, the current thread cannot change the control flow of other threads to enter the processing function when the thread exits. Therefore, in this article, thread exiting by calling *osThreadExit()* enables the thread control block to be cleared. The hardware thread can be freed up for new threads. The *osThreadTerminate()* function only cleans up the thread control block, but cannot release the hardware thread for new threads. At this time, the terminated hardware thread will always be in suspend state, and will not be allocated with thread slot, that is, will not occupy processor resources.

The timer in CMSIS-RTOS2 standard is mainly used to complete timing tasks. When creating a new timer, a callback function needs to be specified. When the timer reaches, the callback function is called to realize the corresponding functions. There are two types of timers, one-off timing and periodic timing. The former only counts once, and the timer will automatically stop after the callback function returns. The latter timer will repeat timing at regular intervals. When the callback function returns, the timer will restart the next round of timing. Using the *set_compare* and *interrupt_on expire* extension instruction, the precise time control function is realized. The callback function is implemented as trap handling of the soft interrupt instruction *interrupt_on_expire*.

It should be pointed out that in order to reduce hardware complexity, area, and power consumption, the hardware implementation may only provide one timer for each hardware thread. Once *set_compare* instruction is executed, the previous timing information will be overwritten. Therefore, in this article, each thread can only create one timer, and *osDelay()* and *osDelayUntil()* functions cannot be called when the timer is working. For scenarios that multiple timers need to be created, this article provides an interface to modify the timer callback function. Users can reuse the same timer by modifying the timer callback function, so as to achieve the effect of multiple timers. Of course, the premise is that the timing ranges of a plurality of timers do not overlap. In addition, users can set up multiple timers by creating multiple threads.

Mutually exclusive is a tool widely used in the operating system, which is used to ensure mutually exclusive access of multiple threads to shared resources. When multiple threads need to access the same shared resource at the same time, it is necessary to obtain the corresponding mutex before accessing it. Only the successful thread can access the resource, which makes it possible for only one thread to access the shared resource at the same time, thus ensuring data consistency. In CMSIS_RTOS2, it supports a thread to recursively acquire the mutexes (*osMutexRecursive* attribute needs to be set). In this case, acquiring the same mutex by the same thread multiple times will not cause thread blocking. Instead, the mutex also needs to be released multiple times by decreasing the counter by one. Only when the mutex count is reset to zero will the mutex be truly released and can be acquired by other threads.

The implementation of mutex requires the support of atomic operations provided by the underlying hardware. This article uses the atomic operation instruction of CSR in the 32-bit basic instruction of RISC-V to provide low-level support for mutexes. The *csrrw* instruction in RISC-V atomically exchanges the values in the source operand register and the target CSR, and the old values in the CSR are stored in the target operand register. The function of the entire instruction is equivalent to a test and set assembly primitive. In this article, eight 1-bit registers are added to the hardware as special CSR of mutual exclusion lock, and the logic of reading and writing the corresponding CSR is added to the data path. The algorithm to realize mutual exclusion by using *csrrw* [46] instruction is as follows. Each predefined mutual exclusion corresponds to an added CSR in the hardware. When acquiring mutexes, write "1" to the corresponding CSR, and then check whether the old value in CSR is "0." If it is "0," the mutex is successfully acquired. If it is "1," it indicates that the mutex has been acquired by other threads. After the mutex is released, "0" is written to the corresponding CSR directly.

In this implementation, the kernel reserves a mutex to ensure the consistency of *CSR_slot* access. Although the operation of CSR in RISC-V instruction set architecture is atomic, the processor implementation divides 32-bit *CSR_slot* into eight 4-bit slots. It represents the thread number that should be scheduled by the hardware scheduler in each cycle. The thread needs to read the *CSR_slot* first, then modify one or more slots to meet their needs, and finally, write the results back. This operation is difficult to complete in one instruction. Therefore, it is necessary to ensure the consistency of *CSR_slot* access by each thread.

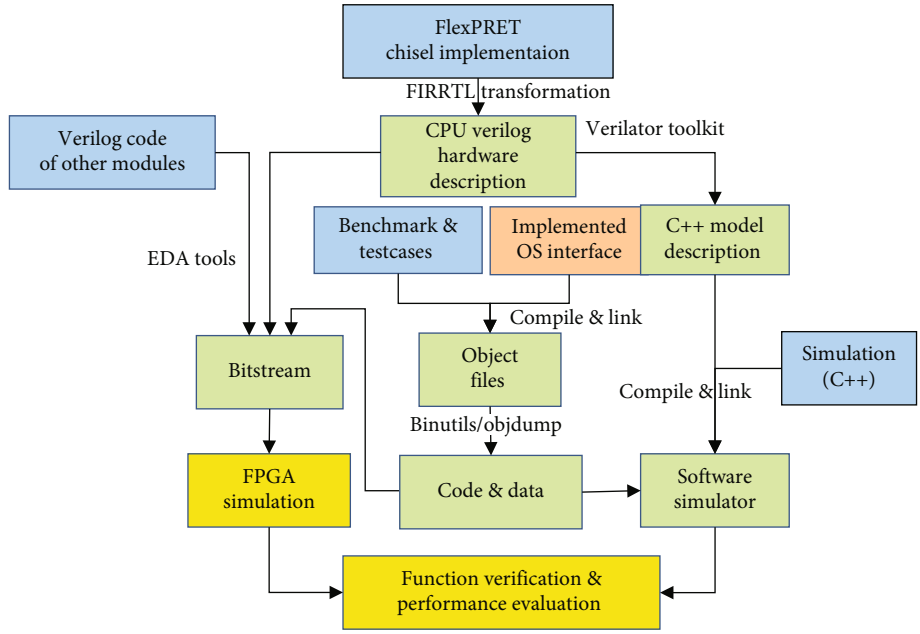

Figure 3: Thread state transition diagram.

Interrupt and exception could be divided into four types, which are interrupt caused by *interrupt_on_expire* instruction, exception thrown by *exception_on_expire* instruction, external interrupts, and internal instruction exceptions. The internal instruction exceptions supported by hardware processor include unknown instructions, system call, misaligned addresses, and memory access errors. Users may call *osThreadGetTrapHandler()* and *osThreadSetTrap()* interfaces to query and set self-trap handling functions, so as to handle different types of interrupts and exceptions in a customized manner.

4. Experimental Results

The configuration in this article adopts 8 hardware threads, 64 KB instruction scratchpad memory and 64 KB data memory, and the running frequency is 100 MHz. The software evaluation environment is based on the C++ simulator generated by chisel (version 2.2.27). The simulator is based on verilator [47] and can accurately simulate the hardware behavior in cycles. In this article, the simulator and FPGA are used to verify the behavior and evaluate the performance. Figure 3 shows the flow of functional verification and performance evaluation. The Verilog hardware description is generated from chisel description, and then, it is simulated by commercial EDA tools (Xilinx Vivado 2020.2). Digilent Artix7 (xc7a100 tcsg324-1) is selected as the evaluation board.

4.1. Functional Verification. The general CMSIS-RTOS2 interface proposed by ARM company is adopted in the software implementation. ARM also provides official test samples for functional verification. In this article, the official test cases for thread scheduling module, timer control module, and mutex management module are selected for function verification. The selected test cases are shown in Table 5.

For the interfaces that do not exist in the official testcases (such as *osThreadJoin()*, *osThreadSuspend()*, and *osThread Resume()* interfaces) and those extended interfaces in this article, some custom testcases are designed, and its behaviors are shown in Table 6.

All the above test cases and CMSIS-RTOS V1 framework are compiled into binary files and exported as the input of the simulator and FPGA evaluation board. Through the evaluation process shown in Figure 3, the results run on the simulator and FPGA show that the behavior of all interfaces meets the expectations.

4.2. Performance Evaluation. Mälardalens [48] is selected as the benchmark for performance evaluation. It is often used for real-time system analysis. Most of the evaluation programs are collected from research institutions and suppliers of WCET analysis tools. Eight typical programs are selected, which include simple operations and data reading. The benchmarks are listed in Table 7.

Three groups of experiments are designed for the evaluation, which are native hardware scheduling, hardware-software cooperative thread scheduling implemented in this article, and RTOS software scheduling. The experimental configurations of three groups are shown in Figure 4.

For the above three groups of experiments, seven child threads are created, respectively, and each child thread runs an instance of the benchmark program. The main thread is responsible for creating the thread and waiting for the child thread. For the hardware scheduling and the hardware software cooperative scheduling implemented in this article, 8 hardware threads are configured as HRTT, and each hardware thread is configured with a thread slot. For RTOS pure software scheduling mode, one hardware HRTT thread is configured to run RTOS itself; other resources are managed by RTOS itself. Eight programs are evaluated in three

Table 5: Description of verification test cases.

(a)

Test case name	Test content (thread management)
TC_ThreadCreate	Create threads with different priorities and parameters
TC_ThreadMultiInstance	Create multiple threads and check the running results
TC_ThreadTerminate	Create a child thread and then terminate it.
TC_ThreadGetId	Create multiple threads and check the thread ID
TC_ThreadPriority	Repeatedly modify the thread priority
TC_ThreadChainedCreate	Create nested child threads and check the child threads
TC_ThreadParam	Pass invalid parameters to all thread control interfaces

(b)

Test case name	Test content (timer control)
TC_TimerOneShot	Create a timer that runs only once and check the result
TC_TimerPeriodic	Create a timer that runs periodically and repeatedly
TC_TimerParam	Pass invalid parameters to all timer control interfaces

(c)

Test case name	Test content (Mutex)
TC_MutexBasic	Create, obtain, and release mutexes, respectively
TC_MutexCheckTimeout	Create mutex, wait for fixed time and infinite time
TC_MutexNestedAcquire	Create a nested mutex to test the behavior is abnormal
TC_MutexParam	Pass invalid parameters to all mutex control interfaces

Table 6: Description of customized test cases.

Test case name	Test content
TC_ThreadJoin	Call osThreadJoin() to wait for the child thread after creating it
TC_ThreadControl	Suspend the child thread, and call osThreadResume() for wakeup
TC_SchedulerSlot	Call osSchedulerSetSlotNum() to set the slots
TC_SchedulerTmode	Call osSchedulerSetTmodes() to set the status
TC_ThreadTrap	Call osThreadSetTrapHandler() to set the trap handler
TC_MutexSpin	Call osMutexSetSpin() to set the mutex spin

Table 7: Description of evaluation benchmarks.

Name	LoCs	Instructions	Description
bs	114	74	Binary lookup
Cover	240	1668	Nested loop with switch
Duff	86	763	Array copy
Fibcall	72	21	Fibonacci calculation
Insertsort	92	328	Insert sort
lcdnum	64	155	Read data and output
Loop3	82	675	Multiple loops
ns	535	4086	Search in multiple arrays

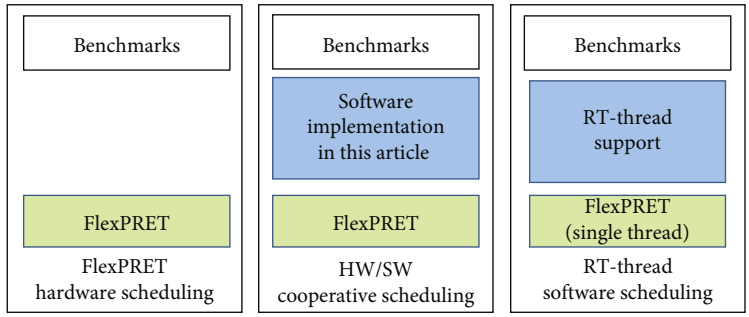

Figure 4: Configuration of three different experiment groups.

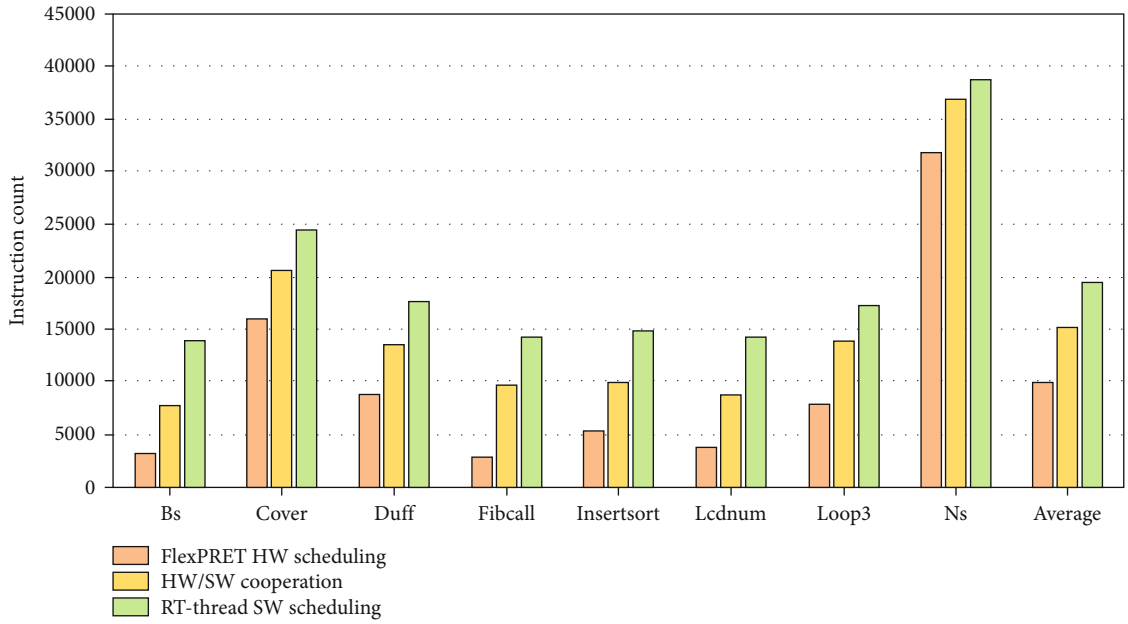

Figure 5: Dynamic executed instructions of different scheduling methods.

scheduling modes, respectively, and the results are shown in Figures 5 and 6.

Experimental results show that the number of dynamic instructions of method in this article is less than that of RTOS software scheduling. Compared with RTOS software scheduling, it reduces 39.22% at most (lcdnum), with an average reduction of 22.51%. This shows that compared with RTOS software scheduling, hardware/software cooperative scheduling in this article runs fewer instructions. When compared with pure hardware scheduling, the number of dynamic instructions increases. The main cost is creation and exiting of threads. The number of cycles reduction is 15.46% in average, indicating that hardware/software cooperative scheduling has a better performance.

In practice, applications running in the same system simultaneously may be different. It is necessary to evaluate the thread scheduling performance when running different test programs. In this article, 8 different mixed cases were built from eight different test programs and then run in three scheduling modes, respectively. The configurations are the same as the previous experiments, and the results are shown in Figure 7.

It can be seen that scheduling in this article has fewer running cycles than RT_Thread software scheduling in 8 groups of experiments, with an average reduction of 26.0%. Compared with FlexPRET native hardware scheduling, it brings an average improvement of 22.94%. Further investigation on the proportion of each program behavior in the total cycles as shown in Figure 8.

Experimental results show that the FlexPRET native hardware scheduling has a large number of unused thread slots, and the RT-Thread scheduling has a high proportion of initialization and scheduling overhead. Hardware software cooperative scheduling implemented in this article can recover the thread slot when the thread exits. The recovered slots can be reused by other threads, to improve the resource utilization, which improves the overall performance.

4.3. Hardware Cost and Power Consumption. Experiments show that the number of LUTs(look-up table) increases by 18.14%, and the number of flip-flop increases by 15.03% after the temporal instructions are introduced. Furthermore, after introducing the MT(meet the time) and FD(final deadline) instructions, the number of LUTs increases by 15.46%,

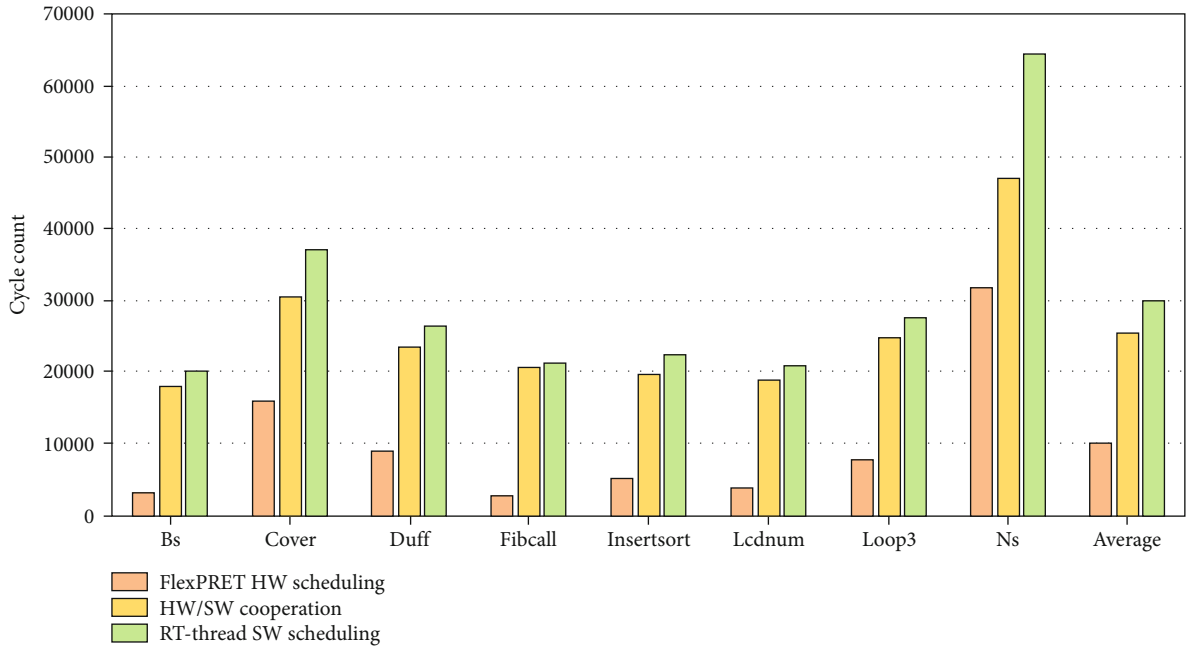

Figure 6: Cycles executed of different scheduling methods.

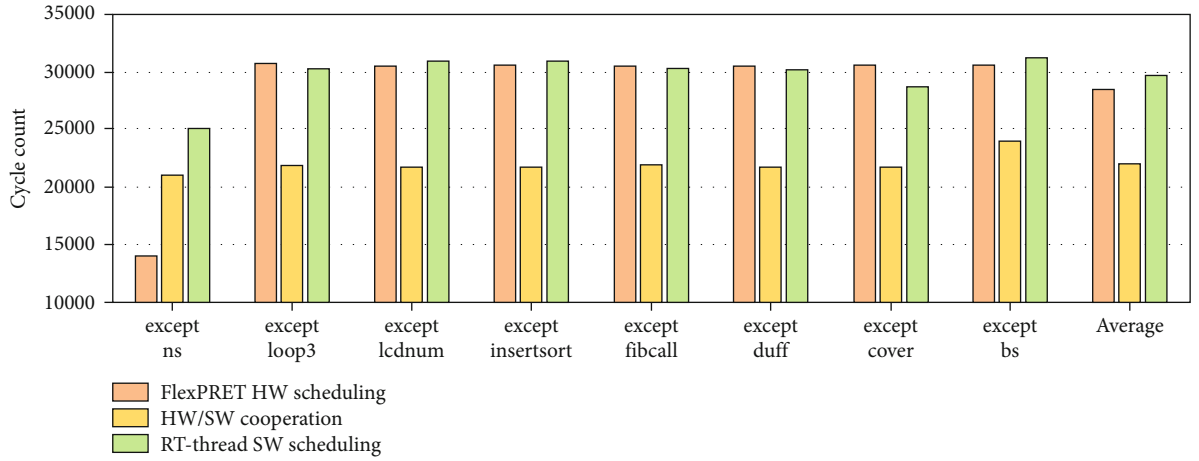

Figure 7: Performance in different mixed threads.

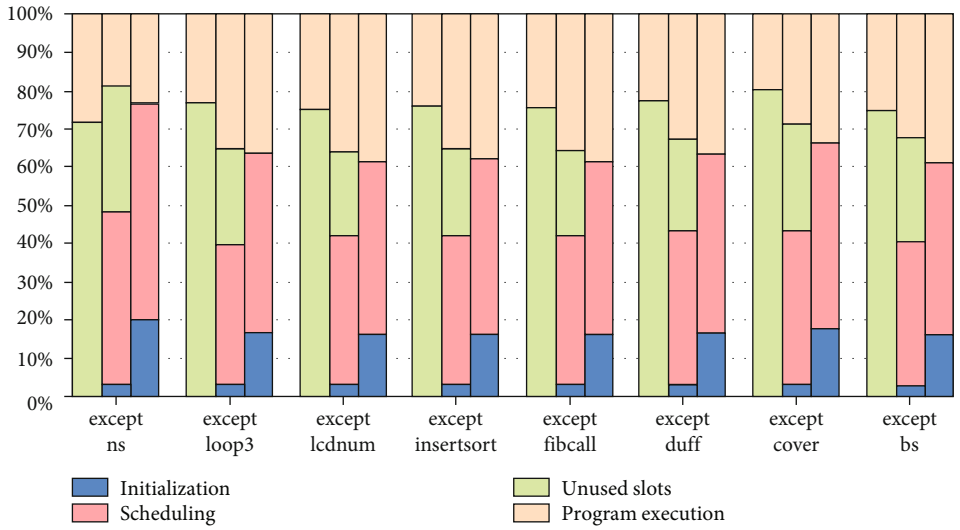

Figure 8: Ratios of execution behaviors in different mixed threads.

and the number of flip-flop increases by 14.12%, respectively. Power evaluation shows that the total on-chip power consumption increases from 0.271 w to 0.286 w after adding six temporal extended instructions at 100 MHz clock frequency, which incurs an increasement of 5.54%. After further extending MT and FD instructions, the total on-chip power consumption changes from 0.286 w to 0.310 w, an increase of 8.39%.

5. Conclusion

Multithreaded processors provide a better balance between thread isolation and performance. This article introduces some architecture extensions for real-time operations and realizes a hardware/software cooperative real-time scheduling mechanism on RISC-V processor, which are compatible with CMSIS-RTOS2. It better utilizes the resources of thread slots in multithreaded processor and greatly improves the performance, which may have many potential applications in edge computing, communication system, intelligence system, and cyber physical system [6–10, 17, 40, 41, 49].

The implementation is verified and evaluated in the simulator and FPGA evaluation board. Experimental results show that in the experiment of mixed scheduling different test programs, the running cycles are reduced by 22.94% and 26.00%, respectively, compared with FlexPRET native hardware scheduling and RT-Thread real-time software scheduling.

Data Availability

The datasets used and/or analyzed during the current study are available from the corresponding author on reasonable request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

Financials supported by the Laboratory Open Fund of Beijing Smart-Chip Microelectronics Technology Co., Ltd. are greatly acknowledged.

References

[1] R. Arbaud, D. Juhász, and A. Jantsch, "Resource management for mixed-criticality systems on multi-core platforms with focus on communication," in *2018 21st Euromicro Conference on Digital System Design (DSD)*, pp. 627–641, Prague, Czech Republic, 2018.

[2] M. Abuteir, "Distributed architecture for developing mixed-criticality systems in multi- core platforms," *Journal of Systems and Software*, vol. 123, pp. 145–159, 2017.

[3] M. P. Zimmer, *Predictable processors for mixed-criticality systems and precision-timed I/O*, UC Berkeley, 2015.

[4] International Standardization Organization, "ISO 26262-1: 2011 road vehicles - functional safety - part 1: vocabulary, international standardization Organization," 2011, https://www.iso.org/standard/43464.html.

[5] *DO178C: Software Considerations in Airborne Systems and Equipment Certification*, RTCA Std., 2012.

[6] X. Dong, G. Chen, M. Lv, W. Pang, and W. Yi, "Flexible mixed-criticality scheduling with dynamic slack management," *Journal of Circuits, Systems and Computers*, vol. 30, no. 10, p. 2150306, 2021.

[7] K. Goossens, A. Azevedo, K. Chandrasekar et al., "Virtual execution platforms for mixed-time-criticality systems," *ACM SIGBED Review*, vol. 10, no. 3, pp. 23–34, 2013.

[8] W. Wang, C. Mao, S. Zhao et al., "A smart semipartitioned real-time scheduling strategy for mixed-criticality systems in 6G-based edge computing," *Wireless Communications and Mobile Computing*, vol. 2021, Article ID 6663199, 11 pages, 2021.

[9] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pp. 239–243, Tucson, AZ, USA, 2007.

[10] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, pp. 1–37, 2018.

[11] A. Waterman, *Design of the RISC-V Instruction Set Architecture*, University of California, Berkeley, 2016, http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.pdf.

[12] "Precision timed (PRET) machines," https://ptolemy.berkeley.edu/projects/chess/pret/.

[13] A. Burns and R. I. Davis, *Mixed Criticality Systems: A Review, Twelfth Edition. Department of Computer Science*, University of York, Tech. Rep, 2019.

[14] S. K. Baruah, "Certification-cognizant scheduling of tasks with pessimistic frequency specification," in *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, pp. 31–38, Karlsruhe, Germany, 2012.

[15] S. K. Baruah, "Schedulability analysis of mixed-criticality systems with multiple frequency specifications," in *2016 International Conference on Embedded Software (EMSOFT)*, Pittsburgh, PA, USA, 2016.

[16] N. Zhang, C. Xu, J. Li, and M. Peng, "A sufficient response-time analysis for mixed criticality systems with pessimistic period," *The Journal of Computer Information Systems*, vol. 11, no. 6, pp. 1955–1964, 2015.

[17] D. de Niz, K. Lakshmanan, and R. Rajkumar, "On the scheduling of mixed-criticality real-time task sets," in *2009 30th IEEE Real-Time Systems Symposium*, pp. 291–300, Washington, DC, USA, 2009.

[18] H.-M. Huang, C. Gill, and C. Lu, "Implementation and evaluation of mixed criticality scheduling approaches for periodic tasks," *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, 2012, pp. 23–32, Beijing, China, 2012.

[19] S. Baruah and S. Vestal, "Schedulability analysis of sporadic tasks with multiple criticality specifications," in *2008 Euromicro Conference on Real-Time Systems*, pp. 147–155, Prague, Czech Republic, 2008.

[20] J. H. Anderson, S. K. Baruah, and B. B. Brandenburg, "Multicore operating-system support for mixed criticality," in *Proceedings of the Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, San Francisco, 2009.

[21] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl, "Proteus: a high-performance parallel-architecture simulator," *ACM SIGMETRICS Performance Evaluation Review*, vol. 20, no. 1, pp. 247-248, 1992.

[22] Y. Kinebuchi, T. Morita, K. Makijima, M. Sugaya, and T. Nakajima, *Analysis, Architectures and Modelling of Embedded Systems. IESS 2009*, IFIP Advances in Information and Communication Technology, A. Rettberg, M. C. Zanella, M. Amann, M. Keckeisen, and F. J. Rammig, Eds., Springer, Berlin, Heidelberg, 2009.

[23] H. Wei, Z. Huang, Q. Yu, M. Liu, Y. Guan, and J. Tan, Eds., "RGMP-ROS: a real-time ROS architecture of hybrid RTOS and GPOS on multi-core processor," *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 2482–2487, Hong Kong, China, 2014.

[24] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee, "A PRET microarchitecture implementation with repeatable timing and competitive performance," *2012 IEEE 30th International Conference on Computer Design (ICCD)*, 2012, pp. 87–93, Montreal, QC, Canada, 2012.

[25] D. May, "The XMOS architecture and XS1 chips," *IEEE Micro*, vol. 32, no. 6, pp. 28–37, 2012.

[26] M. Sbeiti, J. Hinker, and C. Wietfeld, "VLX: a novel virtual localization extension for geographical leash-based secure routing in indoor wireless mesh scenarios," in *2012 IEEE 8th International Conference on Wireless and Mobile Computing, Networking and Communications*, pp. 292–299, Barcelona, Spain, 2012.

[27] G. Heiser and B. Leslie, "The OKL4 microvisor: convergence point of microkernels and hypervisors," in *Proceedings of the first ACM Asia-Pacific Workshop on Systems (APSys 2010)*, pp. 19–24, Delhi, India, 2010.

[28] A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi, "Embedded hypervisor xvisor: a comparative analysis," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 682–691, Turku, 2015.

[29] P. Mantegazza, E. Bianchi, L. Dozio, S. Papacharalambous, S. Hughes, and D. Beal, "RTAI: real-time application interface," 2000.

[30] V. Yodaiken, "The RTLinux manifesto," in *Proceedings of the 5th Linux Expo*, 1999.

[31] P. Gerum, *Xenomai-Implementing a RTOS Emulation Framework on GNU/Linux*, White Paper, Xenomai, 2004.

[32] V. David, A. Barbot, and D. Chabrol, "Dependable real-time system and mixed criticality: seeking safety, flexibility and efficiency with Kron-OS," *Ada User Journal*, vol. 35, no. 4, pp. 259–265, 2014.

[33] P. Paolillo, V. Rodriguez, O. Svoboda et al., "Porting a safety-critical industrial application on a mixed-criticality enabled real-time operating system," in *Proceedings 5th Workshop on Mixed Criticality Systems (WMC), RTSS*, pp. 1–6, Paris, France, 2017.

[34] T. Alves and D. Felton, "Trust zone: integrated hardware and software security," *ARM*, 2004.

[35] S.-C. Oh, K. W. Koh, C.-Y. Kim, K. H. Kim, and S. W. Kim, "Acceleration of dual OS virtualization in embedded systems," *2012 7th International Conference on Computing and Convergence Technology (ICCCT)*, 2012, Seoul, 2012.

[36] J. H. Shah, *ARMithril: A Secure OS Leveraging ARM's Trust Zone Technology*, North Carolina State University, 2012.

[37] J. Martins, J. Alves, J. Cabral, A. Tavares, and S. Pinto, "μRTZVisor: a secure and safe real-time hypervisor," *Electronics*, vol. 6, no. 4, p. 93, 2017.

[38] J. D. Ichbiah, B. Krieg-Brueckner, B. A. Wichmann, J. G. P. Barnes, O. Roubine, and J.-C. Heliard, "Rationale for the design of the Ada programming language," *ACM SIGPLAN Notices*, vol. 14, no. 6b, pp. 1–261, 1979.

[39] E. Kligerman and A. D. Stoyenko, "Real-time Euclid: a language for reliable real-time systems," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 9, pp. 941–949, 1986.

[40] N. J. H. Ip and S. A. Edwards, "A processor extension for cycle-accurate real-time software," *LNCS*, vol. 4096, pp. 449–458, 2006.

[41] D. Bui, E. A. Lee, I. Liu, H. D. Patel, and J. Reineke, "Temporal isolation on multi-processing architectures," in *Proceedings of the 48th Design Automation Conference (DAC)*, pp. 274–279, 2011.

[42] Liu, *Precision Timed Machines, [Ph.D. thesis]*, EECS Department, University of California, Berkeley, May, 2012.

[43] E. Antolak and A. Pułka, "Flexible hardware approach to multi-core time-predictable systems design based on the interleaved pipeline processing," *IET Circuits, Devices & Systems*, vol. 14, no. 5, pp. 648–659, 2020.

[44] D. Broman, M. Zimmer, Y. Kim et al., "Precision timed infrastructure: design challenges," in *Proceedings of the Electronic System Level Synthesis Conference (ESLsyn)*, pp. 1–6, Austin, TX, USA, 2013.

[45] "CMSIS-RTOS2 documentation," https://www.keil.com/pack/doc/cmsis/RTOS2/html/index.html.

[46] A. Waterman and K. Asanovi'c, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2, RISC-V Foundation*, 2017.

[47] S. W. Verilator, *Open simulation-growing up*, DVClub Bristol, 2013.

[48] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalens WCET benchmarks: Past, present and future," in *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET)*, vol. 15, pp. 136–146, Dagstuhl, Germany, 2010.

[49] X. Yang, L. Liao, Q. Yang, B. Sun, and J. Xi, "Limited-energy output formation for multiagent systems with intermittent interactions," *Journal of the Franklin Institute*, vol. 358, no. 13, pp. 6462–6489, 2021.