Dipl.-Ing. Fabian Mauroner, BSc

*mosart*MCU: An Operating System-Aware Microcontroller for Embedded Real-Time Multi-Core Systems

DOCTORAL THESIS

to achieve the university degree of
Doktor der technischen Wissenschaften (Dr. techn.)

submitted to

Graz University of Technology

Supervisor
Univ.-Prof. Dipl.-Inf. Univ. Dr.rer.nat.
Marcel Carsten Baunach

Institute of Technical Informatics

Graz, June 2019

Kurzfassung

Wir sind in unserem Alltag umgeben von eingebetteten Systemen, welche uns in verschiedensten Situationen unterstützen und uns unteranderem dabei helfen Unfälle oder gar Katastrophen zu vermeiden. Viele eingebettete Systeme sind derart in unseren Tagesablauf involviert, man denke hier beispielsweise an Smart Homes, Flugzeuge oder Fahrzeuge, dass sie unbemerkt Messungen vornehmen und auch ihre Umgebung beeinflussen können.

Ein eingebettetes System besteht aus verschiedenen, meist elektronischen Bauteilen, wobei der Mikrocontroller, als Rechen- und Datenverarbeitungseinheit, die zentrale Komponente darstellt. Ein Mikrocontroller erlaubt die flexible Ausführung von Software, welche es ermöglicht, das eingebettete System an neue Spezifikationen anzupassen, ohne dafür neue Hardware bauen zu müssen. Die auf dem Mikrocontroller laufende Anwendungssoftware baut oftmals auf ein Betriebssystem auf, welches die Ausführung von Applikationen unterstützt. In heutigen Betriebssystemen gibt es jedoch Probleme, wie beispielsweise das verschwenderische Verwenden von Stack Speicher, welche immer noch ungelöst sind. In den letzten Jahren wurden verschiedenste Lösungsansätze vorgestellt, um solchen Problemen entgegenzuwirken. Diese erhöhen jedoch den Platzbedarf im Chip enorm, schränken die Planbarkeit von Tasks in einem Multi-Tasking System ein oder erhöhen auch den Aufwand im Betriebssystem. Deshalb kann es vorkommen, dass Zeitschranken nicht eingehalten werden. Dies kann in weiterer Folge dazu führen, dass die Funktionsweise des eingebetteten Systems beeinträchtigt wird, was zu unangenehmen Störungen (Weiche-Echtzeit) oder sogar zu katastrophalen Folgen (Harte-Echtzeit) führen kann. Für harte Echtzeitsysteme wird daher oft ein statischer Systementwurf gewählt, damit sichergestellt wird, dass keine harte Echtzeitschranken überschritten werden. Mit nicht statischen realisierten eingebetteten Systemen kann man nämlich nicht zweifelsfrei vorhersehen, dass alle Echtzeitschranken eingehalten werden. So kann es sein, dass ein Interrupt Request (IRQ) einen Hard-Echtzeit Task unerwartet unterbricht was wiederum zu einem negativen Einfluss auf das Systemverhalten führen kann.

Des Weiteren, um dem ständig steigenden Rechenaufwand in zukünftigen Systemen gerecht zu werden, kommen immer mehr Mehrkernmikrocontroller zum Einsatz. Hierbei führt die echte Parallelität in der Softwareausführung zu neuen Problemen, wie etwa dem Zugriff auf gemeinsamen Speicher, welcher zu Race Conditions führen kann. Das Betriebssystem als auch die Prozessorarchitektur muss dafür Sorge tragen, dies zu vermeiden.

Das Betriebssystem und der Mikrocontroller werden meist unabhängig voneinander entwickelt. Bei der Betriebssystementwicklung wird der Mikrocontroller zumeist nur als eine Abstraktion implementiert, um somit die Portierbarkeit des Betriebssystems auf verschiedene Mikrocontroller zu ermöglichen. Durch diese Abstraktion, kann ein Betriebssystem folglich nicht oder nur mit hohem Implementierungsaufand die Besonderheiten jedes einzelnen Mikrocontrollers berücksichtigen und somit auch nur die Grundkonzepte der Mikrocontroller, welche teils aus den 70ern Jahren stammen, verwenden. Dadurch wird es auch erschwert, das Betriebssystem so zu entwickeln, dass alle Echtzeitschranken eingehalten werden und die oftmals limitierten Ressourcen des Microkontrollers effizient genutzt werden.

Diese Doktorarbeit zeigt einen innovativen Mikrocontroller mit Betriebssystem-Bewusstsein (OS awareness): Die *mosart*MCU besitzt Kenntnis über die Datenstrukturen des Betriebssystems und kann auf diese auch lesend und schreibend zugreifen. Das Betriebssystem-Bewusstsein ermöglicht eine Vermeidung bzw. eine zeitliche Begrenzung von Prioritätsinversionen sowie eine effiziente Stackspeicher-Nutzung. Letztere führt zu geringerem Speicherbedarf im Mikrocontroller, wodurch die Kosten des eingebetteten Systems reduziert werden können. Die effiziente Stackspeicher-Nutzung wird durch die vorgestellten Ansätze StackMMU und CoStack realisiert. Mit EventIRQ und EventQueue wird zudem eine Prioritätsinversion, ausgelöst durch das Betriebssystem, zeitlich beschränkt. Dadurch kann der aktuelle Task nicht durch einen niedrigeren priorisierten Task oder Interrupthandler unterbrochen werden. Außerdem wird es mit dem ebenfalls für die *mosart*MCU entworfenen Konzept des Remote Instruction Calls (RICs) ermöglicht, EventIRQ und EventQueue auf einem Mehrkernsystem zu realisieren. RIC basiert auf einem in dieser Arbeit vorgestellten System-on-a-Chip (SoC) Bus, der Prioritätsinversionen auf SoC Ebene vermeidet. Mit RIC werden globale Aufgaben in lokale Aufgaben überführt, um hiermit beispielsweise einen gemeinsamen Speicherzugriff mit potenziellen Race Conditions auf globalen Speicher zu vermeiden.

Das Betriebssystem-Bewusstsein im Mikrocontroller ermöglicht ein Systemverhalten, welches mit einer reinen Softwarelösung nicht möglich oder nur mit sehr hohem Rechen- und Verwaltungsaufwand erreichbar wäre. Deshalb eröffnen die in dieser Doktorarbeit vorgestellten Denkansätze, welche in den *mosart*MCU implementiert wurden, neue Möglichkeiten auf Basis derer ein Betriebssystem und ein Mikrocontroller nach einem *Mikrocontroller/Betriebssystem Codesign* entwickelt werden können.

Abstract

Today, we are surrounded by embedded systems that support us in safety-critical situations and help us to avoid accidents or even disasters. Many embedded systems are seamlessly implemented into our daily lives by silently making measurements and by influencing their environment, such as in smart homes, airplanes, or vehicles.

An embedded system consists of various components, mostly electronic, whereby the Microcontroller Unit (MCU), as a computing and data processing unit, very often represents the central component. An MCU allows the execution of flexible software, which makes it possible to adapt the embedded system's software to new specifications without having to build new hardware. The application software running on the MCU is often based on an Operating System (OS). In today's OSs, however, there are still unsolved problems, such as the overbooking use of stack memory. In recent years, various approaches were presented to counteract such problems. However, these approaches enormously increase the space requirement in the silicon, limit the schedulability of tasks in a multi-task system, or increase the overhead in the OS. Therefore, it can happen that time boundaries cannot be satisfied. This can furthermore result in the functionality of the embedded system being impaired (soft real-time) or lead even to a catastrophe (hard real-time). Therefore, for hard real-time system a static software architecture is mainly chosen, to ensure that no time boundaries are violated. Since the execution time of today's non-statically implemented embedded systems cannot be predicted precisely. The reason is, that these systems would have to adapt dynamically itself and so they cannot guarantee with certainty that all real-time boundaries will be met. For example, an Interrupt Request (IRQ) may unexpectedly interrupt a hard real-time task, which in turn may lead to a negative influence on the system behavior.

In order to accomplish the constantly increasing computing effort in future systems, more and more multi-core MCUs are being used. The real parallelism in software execution leads to new problems, such as the access to shared memory, which can lead to race conditions. The OS and the computer architecture must take care to avoid these new problems.

The OS and the MCU are usually developed independently of each other. During OS development, the MCU is usually implemented only as an abstraction (i.e., Hardware Abstraction Layer (HAL)), in order to enable portability of the OS to different MCUs. Due to this abstraction, an OS cannot take into account, or only with a high implementation effort, the special features of every individual MCU and can therefore only use the basic concepts of the MCU, some of which date back to the 70s. This also makes it more difficult to develop the OS in a way that all real-time constraints can be satisfied and that it can efficiently handle all the MCU resources.

This doctoral thesis shows an innovative MCU with OS awareness: The *mosart*MCU has knowledge of the OS's data structures and can access them by reading and writing. The OS awareness enables the avoidance or temporal bounding of priority inversions as well as an efficient stack memory usage. The latter leads to a lower memory requirement in the MCU, whereby the costs of the embedded system can be reduced. The StackMMU and CoStack approaches presented in this thesis enable the efficient use of stack memory. With EventIRQ and EventQueue a priority inversion, caused by the OS, is time bounded and no unpredictable interruption may occur by a lower prioritized task or interrupt handler. Furthermore, Remote Instruction Call (RIC), also developed for the *mosart*MCU, makes it possible to realize EventIRQ and EventQueue on a multi-core system. RIC is based on a System-on-a-Chip (SoC) bus presented in this thesis, which avoids priority inversions at SoC level. With RIC, global operations are transferred to local operations, for instance, in order to avoid potential race conditions on globally shared memory accesses.

The OS awareness in the MCU enables a system behavior that would be impossible with a pure software solution or would only be possible with a very high computing effort. Therefore, the approaches presented in this doctoral thesis, which were implemented in the *mosart*MCU, open a new way to develop OSs and MCUs based on an MCU/OS codesign approach.

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present doctoral thesis

....

date   signature

To my family

Acknowledgements

First, I want to thank my supervisor Prof. Marcel Baunach. You gave me the opportunity to do a PhD in your Embedded Automotive Systems (EAS) group. You gave me complete freedom in building up the *mosart*MCU including the *mosart*MCU-OS and in realizing all my research interests and ideas. Thank you for supporting me.

I want to thank my colleagues in the EAS group. In our discussions, you all helped me to find new ideas and solutions. Further, I want to thank my colleagues from the institute, since during my PhD you all directly and/or indirectly supported me.

Another thanks go to my parents. You fully supported me, when I had the idea to go study. You fully respected my ideas and dreams and supported me financially to achieve all of my dreams.

Thanks to my son Dominik, you gave me the energy and motivation for the final spurt of my PhD.

Last but not least, a special thanks goes to Sabrina. Sabrina you always were behind me, supported me, and pushed me ahead also in difficult times of my PhD.

Thank you all!

Contents

1. Introduction

This doctoral thesis introduces the concept of an OS-aware MCU for embedded multi-core real-time systems. The novel MCU design, which assists the OS in its execution, is called *mosart*MCU, the abbreviation for *Multi-Core Operating System-Aware Real-Time MCU*.

At the end of the last century, Marc Weiser [1] described his visions about computers in the 21st century and coined the term *ubiquitous computing*. He predicted that computers will be used as smart embedded systems that enhance our daily life with convenience and safety.

Applying novel concepts in the field was required to realize Weiser's visions. Intense research was conducted in the area of Wireless Sensor Networks (WSNs), resulting in the development of many different sensor nodes [2, 3, 4, 5, 6]. A sensor node is a (mostly) battery powered computer system with sensors and a radio unit for transferring measured sensor data to sinks. At some point, WSNs have been extended to Wireless Sensor/Actuator Networks (WSANs), which not only sense but also actuate and influence the field.

Nowadays, the trend goes from locally networked WSANs to globally distributed embedded computers, as in the Internet of Things (IoT). Thereby, the aim is to minimize computer systems and to integrate them into business, information, and social processes by connecting them to a global infrastructure (i.e., the Internet) for exchanging data [7]. These conceptions bring the idea of the IoT in line with Marc Weiser's prediction of ubiquitous computing.

Most of the embedded systems for the IoT are battery-powered. This implies that the operating time of the embedded system is limited, and a reduction and optimization of the power consumption of these systems is needed. To achieve this, many techniques can be applied: e.g., the operating frequency or voltage can be reduced and the software can put electrical components into sleep modes. In addition to that, developers seek to use low power electronic components whenever possible. In the case of a computational unit, low power MCUs suit well. An MCU contains one or more processor cores, non-volatile memory (e.g., flash), volatile memory (e.g., Random Access Memory (RAM)), peripherals (e.g., serial interfaces), and other computer system components in one single chip; wherefore, it is also called System-on-a-Chip (SoC).

Today's embedded computer systems (as MCUs) are restricted in terms of computational power and available memory compared to modern desktop or even server computer systems. This is primarily caused by the aspired low power consumption, available space on the die, and production costs. Furthermore, embedded systems are aiming for completely different purposes compared to desktop computer systems. This is the reason why many different OSs (e.g., [8, 9, 10, 11, 12]) were created to operate today's small and resource constrained embedded computer systems.

Due to the deep involvement of computer systems in our daily life, standards that ensure the functional safety of a developed product have been established. Functional safety describes how to achieve correct execution of a functionality regarding its inputs, failures in hardware and software, or environmental changes [13]. The general safety standard for *electrical/electronic/programmable electronic safety-related systems* IEC 61508 [14] describes approaches to avoid mistakes during the development process, to monitor the occurrence of failures, and to deterministically handle errors.

For a better alignment of the safety requirements to specific fields, special safety standards have emerged. For instance, the DO-178B *Software Considerations in Airborne Systems and Equipment Certification* [15] and the ISO 26262 *Road Vehicles – Functional Safety* [16] are safety standards for avionic and automotive systems. Well-known OS standards, such as the ARNIC-653 [17] and AUTomotive Open System ARchitecture (AUTOSAR) [18] implement the above mentioned safety standards for avionic and automotive systems, respectively. To be compliant with the AUTOSAR standard, all independent software components must be *free from interferences*. Thus, in the whole software, wrong processor allocations, incorrect memory synchronizations, and the occurrence of blocking due to deadlocks or priority inversions must be avoided or at least considered (Annex D in [16]), for instance by time bounding a priority inversion. Otherwise, the system cannot be certified by the respective authority and must not be deployed to series vehicles.

In the requirement specification, the customer specifies, among others things, the functionality and the behavior of the application software. In order to enable software engineers to develop jointly the same application, to reduce the development costs, and to shorten the development time, the applications are mostly developed on top of an OS. In the OS, among of multitude of other administrative work, the scheduler manages the allocation of a task (often independently developed) to the computational units (i.e., cores) according to the OS's scheduling policy. For desktop computers, the policy is to give the user a feeling of a highly reactive system. For server systems; however, a high data throughput is aimed [19]. For most embedded systems, the scheduler has to satisfy specified time boundaries. Hence, embedded OSs are often Real-Time Operating Systems (RTOSs). In an RTOS, a real-time process (e.g., measuring a physical process) is not allowed to start before a specified lower time bound and not allowed to end after a specified upper time bound. In a soft real-time system, the violation of time boundaries is accepted but could reduce the output performance (e.g., inaccurate measured physical process). In hard real-time systems, these violations must be completely avoided, otherwise they could result in a disaster. The probability of a disaster, caused by an embedded system, can be reduced by satisfying the safety standards.

An increasing number of complex algorithms will be implemented in low-power embedded real-time systems, which also increases the computational demands. The common solution so far to satisfy this computational demand was to increase the operation frequency of the computational unit. However, this further increases the power demand leading to critical thermal power densities [20]. Another approach to increase the computational power is to distribute the software among different computational units, namely to multiple cores. By simultaneously executing the software on different cores, new challenges arise for satisfying the *freedom from interferences*.

1.1. Problem Statement

In the context of real-time multi-core embedded systems, many issues are still not fully solved. The efficient stack memory usage and the time bounding of priority inversions are two of these issues, this thesis deals with.

1.1.1. Stack Memory Usage

An MCU, beside the computational unit and other peripherals, contains memory to store and read data. The memory size is constrained due to the limited space on a die and to reduce the costs. Thus, memory constrained embedded systems have to use the memory in an efficient way. The well-known Memory Management Unit (MMU) concept of modern desktop or server systems is only rarely found in embedded systems. The reason is the high power consumption, the required space on the die, as well as the non-deterministic execution times upon accesses [21, 19]. This also influences each task's execution time in a hardly predictable way and makes a classic MMU unsuitable for real-time systems in general.

After the initialization of a task in an MMU-less embedded system, the task's static variables are placed in the memory and will occupy the memory as long as the task is not removed by the OS. The same applies for a memory space to store temporary variables and function call related data, known as *stack*. The stack uses a Last-In First-Out (LIFO) data structure, and a stack pointer that points to an address in the stack to indicate dynamically the threshold of valid and invalid data in the stack. Furthermore, the stack pointer is used to access, with a relative offset to the stack pointer, the data in the stack. The Application Binary Interface (ABI) specifies how the stack is managed by the compiler, the OS, and which operations must be applied on it.

The stack contains dynamically stored variables and temporarily stored registers of the computational unit (e.g., the return address during a function call). If an MMU-less embedded system supports different logical execution flows, by using tasks scheduled by an OS, the

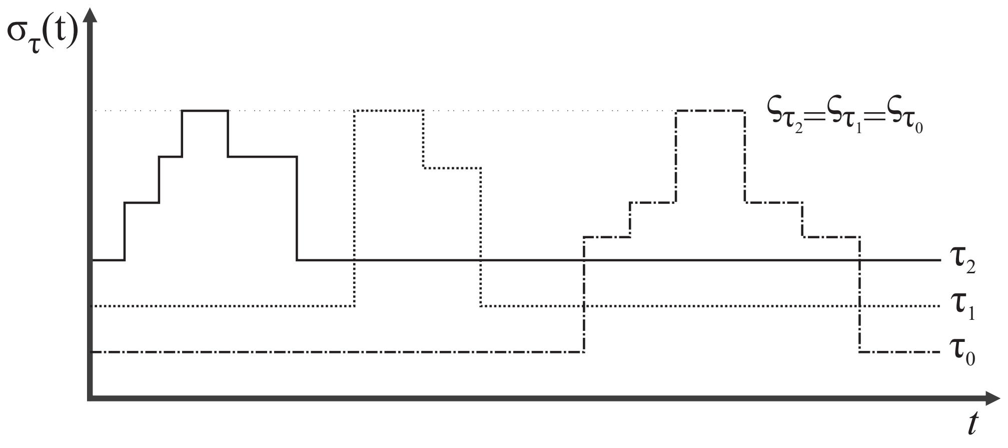


Figure 1.1.: Example stack consumption of three tasks with individual stacks.

straightforward approach would be to assign an individual stack to every task. This approach is frequently used in today's embedded system OSs (e.g., [22, 11]). However, it results in stacks that are not fully utilized simultaneously, as shown in the example in Figure 1.1. In this example, three different tasks (i.e., τ_2, τ_1, τ_0) are executing and every one reserves an individual stack with the size $\varsigma_{\tau_2} = \varsigma_{\tau_1} = \varsigma_{\tau_0}$. If we assume that all three tasks are always executing in the same sequence, the individual stack memory reserved to each task is not simultaneously fully used. To avoid this memory overbooking, there exist solutions in which a common stack is shared among all tasks (e.g., [9, 23, 24]), or a combination of an individual and a common stack is used (e.g., [25, 26]).

A common stack may reduce the overall required stack memory, but it may also restrict the schedulability of the tasks. This happens if the stack of the task that should be scheduled is currently not on top of the common stack. For instance, a high priority task is scheduled, builds up some stack data, and calls a sleep function, leaving the data on the stack. A low priority task is then scheduled by the OS and builds up stack data directly consecutive to the previous task. While executing, the previously sleeping high priority task shall be resumed. Now, the common stack contains data from the low priority task on the top. Hence, for the high priority task, all the relative addresses in the stack are invalid (which can be handled with an offset), and even worse, the situation would restrict the high priority task in growing its stack memory. This creates the need of using non-preemptive tasks with synchronization points [9], or of using *run to completion* [8] approaches, where tasks will never interleave.

Non-preemptive scheduling, where the stack is implicitly or explicitly cleared on each context switch, restricts the OS in scheduling tasks with arbitrary interleaving. The OS can only schedule a task on synchronization points or on the task completion. For real-time systems, this approach limits the schedulability and probably real-time constraints cannot be meet by the OS. This is the reason why most RTOSs reserve an individual stack to every task, with the drawback of overbooking memory.

1.1.2. Priority Inversion

In 1980, Lampson and Redell [27] were the first authors who mentioned the priority inversion problem, which was later coined by Rajkumar *et al.*, in [28].

The priority inversion problem arises, if a lower prioritized task prevents the execution of a higher prioritized task that would be otherwise executed on the computational unit, in a system with prioritized tasks where the priorities specify the importance of the task. The priority inversion must be avoided or be at least time bounded (to a specific time) to guarantee the schedulability of the tasks in a real-time system. Thus, systems that do not handle non-preemptive resources with an appropriate approach, may lead into timely unbounded priority inversions.

A famous example for a priority inversion occurred in the Mars Rover [29], where the watchdog realized that a high priority task did not signal its aliveness due to a priority inversion; and therefore, the watchdog triggered a system reset. After remote debugging, the engineers found out that a priority inversion occurred and fixed the problem by enabling the resource management protocol that time bounds the priority inversion, provided by the OS.

Resource management protocols are still an ongoing research field, with many different approaches for time bounding the priority inversions in single cores [30, 24, 31] and multi-cores [28, 32, 33, 34]. However, a priority inversion may not only occur on resources in an OS, but everywhere where prioritized instances (e.g., tasks, OS, cores) are using shared resources that might not immediately be accessible by the higher prioritized requesting instance.

This thesis addresses a special form of the priority inversion that emerges in the OS: While the OS performs code for a lower prioritized task, a higher prioritized task would be ready to be executed on the computational unit. We call this kind of priority inversion Operating System Priority Inversion (OS-PI). Figure 1.2 depicts an example with four tasks and two OS-PI occurrences. Let us assume that the tasks are prioritized according to their index, in which a higher number represents a higher priority and in which the OS is prioritized above all tasks. The tasks are using two OS events e_t and e_s, whereby an event e is a synchronization primitive for which a task may wait, and the event can be triggered by another task or by the OS.

- At time t_0, t_1, and t_2 the tasks τ_3, τ_2, and τ_1 start waiting for their events, respectively. Every waiting call is a jump into the OS, executed by a syscall. Then, the scheduler chooses the next task according to its scheduling policy.
- At time t_3, an interrupt is triggered and an Interrupt Service Routine (ISR) is immediately executed in the context of the OS. The ISR sets the event e_t, for which task τ_3 is waiting.

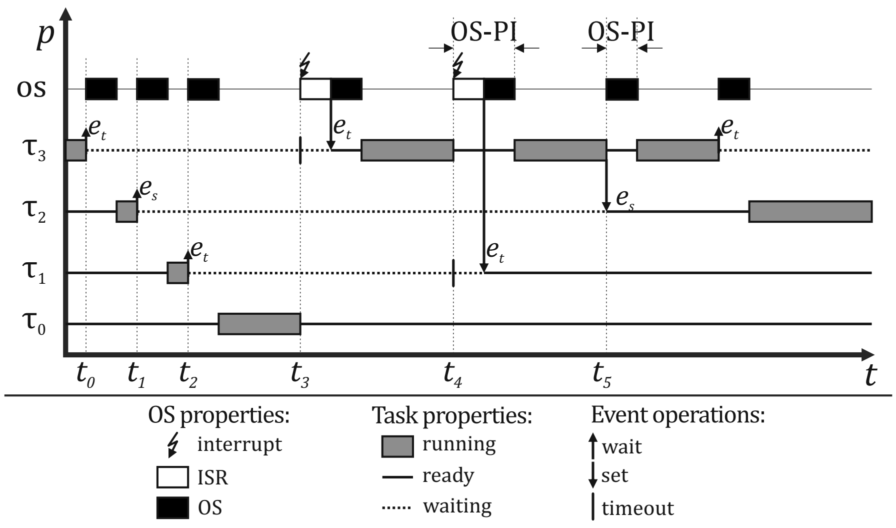


Figure 1.2.: Example of two different Operating System Priority Inversions (OS-PIs).

The lowest prioritized task τ_0 is preempted by the OS, and the OS schedules τ_3 as it is the highest priority ready task. Thus, no priority inversion occurs.

- At time t_4, once again, an interrupt is triggered and the ISR is executed. There, the event is directed to the task τ_1, but the higher priority task τ_3 is still interrupted by the ISR. Here, an OS-PI occurs, due to the jump into the OS context through an Interrupt Request (IRQ) relevant for a lower priority task.
- At time t_5, task τ_3 sets the event e_s, for which the lower priority task τ_2 is waiting. Here, a syscall is executed leading to a jump in the OS context. Once again, an OS-PI occurs, because task τ_3 is jumping into the OS for setting the event that is addressed to the lower prioritized task τ_2. This happens because task τ_3 does not know, to which task the event is addressed; therefore, which priority the receiver task has. To avoid this OS-PI, the triggering of the event could also be deferred (but only for lower prioritized tasks); i.e., until either τ_3 is preempted by a higher prioritized task or it preempts itself.

The example shows that the execution flow of a high priority task could be interrupted by a lower prioritized task through the OS. This leads to an OS-PI that delays the finishing of the required execution time for the task's work. An OS-PI could also lead to the violation of the task's time bounds. To prevent the violation of real-time constraints, the approaches of this thesis aim to avoid or at least to time bound OS-PIs. The approaches are not only limited to the real-time issue. We can also take into account security issues. For instance, a low prioritized task could deliberately produces Denial of Service (DoS) attacks by generating many OS-PIs (e.g., by triggering many software events e_s). However, if OS-PIs were avoided, DoS attacks would be also suppressed.

1.2. Contributions

Real-time embedded system issues are still an ongoing research topic. This doctoral thesis addresses them by providing a contribution in the following fields:

1. **OS awareness in the MCU:** OS (or part OS) integration into CPUs has been researched and proposed in the last years, but based on the assumption that the software is static and does not change over time. However, future embedded systems have to have support for dynamic (and part) updates of the application code at run-time (e.g., due to new regulations or application needs). The dynamic software adaptation is supported by the OS awareness concept, which has been implemented into the *mosart*MCU, proposed in this thesis. The OS-aware MCU integrates OS functionalities, which are traditionally implemented in software, into hardware. Thus, those hardware-implemented OS functionalities support the OS in its execution by performing some operations in hardware, concurrently to the application code that runs on the core. Furthermore, due to the hardware/software codesigned MCU approach, the OS has the knowledge of the hardware and in particular the hardware about the OS, whereby the hardware is tailored to the OS. In particular, the OS-aware extension is able to access and modify OS instances (e.g., tasks, events) and to assist the OS by performing OS operations in parallel to the running application. This novel concept opens new opportunities for future system designs and it is the base to implement the following presented new OS concepts for solving (or mitigating) still unresolved embedded systems issues.

2. **Efficient Stack Memory usage:** To overcome the overbooking of stack memory, this thesis proposes two different approaches. The first approach, *StackMMU*, divides a shared stack memory into pages, which are allocated on demand to the requesting task through the OS awareness of the *mosart*MCU. With the exception of an initialization phase, the execution of StackMMU is transparent to the executing software. All the StackMMU operations are executed in a constant time, which suits well for real-time systems. The second approach, *CoStack*, is based on StackMMU. There, a task collaboratively frees its stack memory for a higher prioritized task if no more memory is available on the shared stack. Both concepts, as well as use case evaluations, are shown in the following sections of this thesis.

3. **Priority Inversion time bound/avoidance:** In real-time systems, a priority inversion may occur on every shared resource used by prioritized instances in the system. Through the support of the *mosart*MCU, this thesis proposes concepts to time bound or even to avoid priority inversions in an embedded real-time system. The first concept, *EventIRQ*, is an approach, which maps all IRQs to OS events and all the ISRs are moved to regular tasks. Therefore, a priority unification of tasks and interrupts is achieved and the OS-PI

problem for interrupts and software events is time bounded. On top of EventIRQ, this thesis presents a second concept called *EventQueue*. EventQueue is an Inter-Process Communication (IPC) concept that allows a communication between tasks whereby here the OS-PI is time bounded, as for EventIRQ. Both extensions are supported in multi-core systems, too. This is achieved by the novel Remote Instruction Call (RIC) approach, based on the idea of a Remote Procedure Call (RPC). Here, instead of procedures, the instructions are performed on a remote core. Moreover, this thesis presents a concept to avoid priority inversions on the SoC bus, which connects the cores together. The following sections present these concepts along with use case evaluations.

1.3. Thesis Outline

This doctoral thesis is structured as follows: Chapter 2 presents related work in the context of OS assistance computational units, efficient stack memory usage approaches, and approaches to time bounding priority inversions for different situations in embedded systems. Chapter 3 introduces the underlying architecture for supporting OS awareness, the proposed concepts for efficient stack memory handling, and OS-PI time bounding approaches for single-core and multi-core embedded real-time systems. Chapter 4 shows evaluation results of the proposed approaches, while Chapter 5 summarizes this thesis with an outlook on future work. Chapter 6 collects all the publications related to this thesis.

2. Related Work

This chapter reviews related work in the scope of this thesis. Section 2.1 gives an introduction on the state of today's commonly used computer architectures in embedded systems provided by the industry. Section 2.2 investigates academic and industrial embedded systems projects that are (partly) integrating an OS into hardware. The last two sections summarize related work on efficient stack memory usage, priority inversions on IRQs, IPCs, as well as on SoC buses. The sections 2.2 to 2.4, in each case, conclude with a respective summary and differentiation of the mentioned related work with respect to the proposed solutions in this thesis.

2.1. State-of-the-art MCUs

Embedded computing systems are becoming more and more omnipresent in our daily life. In the last 50 years, different computer architectures for embedded systems have emerged that are targeting different kinds of applications. From consumer electronics, to healthcare systems, to automotive systems, each computer architecture supports its application domain with specific performance, behavior, features, as well as power characteristics.

Table 2.1 lists some MCUs, which are still relevant in the industry. An 8-Bit computational unit was the dominant computer architecture in the late 70ies. Twenty years later, the 8-Bit computer architecture was extended to support 16-Bit and 32-Bit operations as well as memory addresses. Independent of the bit width, the basic concepts of all these computer architectures exist for 15 to 40 years. OS awareness, which assists the OS to fulfill all the application's requirements, is either not existing or only rarely found in these architectures.

One reason for the missing OS awareness in these computer architectures is a strict separation of the OS and the computer architecture. This separation allows that an OS can support many computer architectures, and on a computer architecture many different OSs can be executed. This makes it possible to develop an application that works efficiently on several computer architectures with different Instruction Set Architectures (ISAs). Further, if a developer is already familiar with the OS, he or she can immediately concentrate on the application software, instead of learning the OS or computer architecture (thus, this helps to reduce the time to market). Therefore, the OS offers a hardware independent Application Programming Interface (API) to the developer for writing the application, and the OS uses the underlying computer architecture through the ISA. The ISA defines the interface to the computer architecture and allows the OS

Table 2.1.: List of frequently used computer architectures in today's embedded systems.

Architecture	Date	Bit width	Notes
MC68HCxx [35]	1974	8	CISC accumulator, based on Motorola 6800
PIC [36]	1976	8	RISC
8051 [37]	1980	8	CISC accumulator
MSP430 [38, 39]	1992	16	RISC-like, especially for low power
SuperH [40]	1992	32	RISC-like
Cortex-M [41]	1994	32	RISC, based on ARM7TDMI
megaAVR [42]	1996	8	RISC
TriCore [43]	1999	32	RISC-like with multi-core
MicroBlaze [44]	2002	32	RISC soft-core
NIOS [45]	2004	32	RISC soft-core
AVR32 [46]	2007	32	RISC, based on megaAVR

developers to access the hardware functions for realizing an efficient implementation of all the OS functionalities. However, Chris Schläger from AMD Inc. explained in his talk at the Conference on Architecture of Computer Systems (ARCS) in 2008, that this strict separation is not always applicable in the real world:

> "*The influence of the operating system interface of a CPU on its overall performance has grown tremendously. For AMD as a hardware vendor, this created a big challenge. The traditionally long feedback cycle between us and the OS vendors had to be shortened dramatically. Instead of relying on outside OS developers we had to bring OS development in-house. ...*" [47]

This means, to get out the full potential of those even more complex computer architectures, the OS developers have to have a closer interaction with the computer architecture manufactures. This trend can also be seen in the mobile phones field; Apple and Google are aligning their OSs to their own hardware, because they claim that the tight integration of hardware and software is obligatory [48]. The goal of a tight integration of hardware and software as well as the consideration of the OS in the hardware development, is to improve the whole system performance, to increase the dependability (i.e., availability, safety, security), to use features offered by the hardware in an efficient way, and to align the hardware to the software architecture. For instance in [49], the authors showed that if the OS has a deeper knowledge of the underling multithreading technology, the throughput of the computation could be improved up to 30%.

These facts lead to the conclusion that OS developers have to have a deep interaction with the computer architecture developers. Thus, why do not we extend the computer architectures with OS awareness? However, how can we develop the computer architecture and the OS by applying an appropriate MCU/OS codesign? The following related work tackles these questions, as well as this doctoral thesis is going to propose answers to them.

2.2. OS Implementation in Hardware

This section lists different OSs, either fully or partially implemented in hardware. The integration of OS functionalities in hardware helps the OS to provide some properties that are impossible with a pure software solution, as mentioned in the next academic and commercial projects.

2.2.1. FASTCHART and FASTHARD

The project FASTCHART [50] implements a complete RTOS in a user specific hardware. Its goal is to achieve a deterministic time behavior of all the tasks by dividing the hardware into two parts. One part is the main core for processing the task code and the second part (namely a coprocessor) implements the OS functionalities. The hardware OS offers three different OS functionalities provided as function calls: First, the activation of another task; second, the termination of a task itself; and last, a function to let the task sleep for a specified time.

The coprocessor contains the scheduler, the Task Control Blocks (TCBs), and the queues for handling the currently running task, the ready tasks, and the waiting tasks. When scheduling a new task, the coprocessor saves the context of the preempted task into the TCB and restores the context of the new scheduled task. Then, the new scheduled task executes on the main core. If one of the three OS function calls is triggered by the currently running task, the main core triggers the coprocessor to execute the requested functionality. This means, if no OS functionality has to be performed, the coprocessor is idle. For the whole process, a synchronization primitive between the main core and the coprocessor is used.

Due to the hardware implementation of the scheduler into the coprocessor, the approach has a limitation of 64 tasks and a maximum priority level of eight. For each priority level, the coprocessor has a separate queue, and a task in the highest prioritized nonempty queue will be scheduled by the coprocessor. To support waiting for a specific time, the coprocessor has one individual down counter for every task.

The project FASTHARD [51] implements the FASTCHART approach with some additional extensions, such as rendezvous synchronization, interrupts, or periodic tasks. Instead of the specialized coprocessor, FASTHARD uses a standard processor for the OS.

2.2.2. Silicon OS

The project Silicon OS [52] proposes an implementation of system calls and scheduling functionalities as a peripheral. To perform the system calls, the application has to write the arguments to the peripheral, by using Memory Mapped I/O (MMIO) registers. Afterwards, the application has to read out the return value from the peripheral's MMIO register. The detection if the system call or an external event lead to a rescheduling is processed in the peripheral simultaneously to the application execution.

In comparison to FASTCHART or FASTHARD, the CPU still executes a light OS kernel implemented in software besides the peripheral. The kernel is still responsible for performing the memory management and time management. For the task management, synchronization, and interrupt management; however, the peripheral is responsible. All the syscalls and external events are recognized by the peripheral that is able to signal, with an IRQ, the CPU for scheduling a new task.

In their work, the authors show the Field Programmable Gate Array (FPGA) resource utilization that increases constantly with an increasing number of tasks, events, or timers. After synthesizing the hardware, the number of supported tasks, events, or timers are unchangeable. Hence, it is impossible to add new tasks to the system without resynthesizing. Another limitation is that Silicon OS supports only eight different task priorities that may be insufficient for future embedded systems.

2.2.3. RTM

The Real-Time Task Manger (RTM) described in [53], implements only parts of an RTOS in hardware instead of a full RTOS. This leads to a partial OS on the software layer and a partial OS in the hardware layer. Over an MMIO interface, the hardware OS is accessible like a peripheral and communicates with the CPU that is executing the task's code.

The RTM does the scheduling of prioritized tasks, the time management, and the event management. Here, the hardware extension supports a predefined number of tasks, for instance 64 or 256. For every task, a record contains task information such as the priority and the running state i.e., running, ready, or waiting for a delay or an event. Thus, the hardware decrements the counter in the record on each task if the task is waiting for a delay. However, if the task is waiting for an event, the triggered event sets its trigger in the record, and the scheduler is responsible to resume the task according to the tasks' priorities. This approach has the same drawback as in the work before; the number of records is unchangeable at run-time. Thus, after the configuration of the system, the number of tasks stays constant.

2.2.4. SEOS

The project SEOS [54] aims to reduce the difficulty in adapting the system to new application requirements. Here, the OS is fully implemented in hardware and the hardware is accessible by MMIO address registers. The OS hardware is implemented as an Intellectual Property (IP) module, which can be easily integrated when developing a SoC. The authors provide an API in a source file, which makes it easy to use the OS hardware in software. They claim that with all their provided code, in-depth knowledge on the system design and the OS is not required anymore.

Their hardware module supports a Rate Monotonic (RM) scheduler, inter-task communication, synchronization, and time handling. The number of tasks, semaphores, and mailboxes can be configured at development time. Thus, there is no possibility to increase or decrease these numbers without resynthesizing the hardware.

2.2.5. Configurable Hardware Scheduler

Kuacharone *et al.* [55] proposed a configurable hardware scheduler. Their approach handles the scheduling, the time-tick processing, and interrupts in a separate hardware module, which relieves the CPU. Compared to the previous presented projects, here the hardware component is reconfigurable according to the application requirements. This means that the scheduler type (e.g., priority based, RM, or Earliest Deadline First (EDF)), the number of tasks, the number of external interrupts, and the timer resolution are reconfigurable with a tool. Then the tool synthesizes the hardware with the defined properties. The authors claim that in future systems, the configuration and synthesizing of the hardware scheduler could be performed directly on the SoC. In a SoC with a CPU and an FPGA (e.g., Xilinx Zynq-7000 [56]), the CPU may synthesize and reconfigure the FPGA in the SoC on run-time.

The communication between the CPU and the hardware scheduler is implemented by MMIO address registers. Furthermore, there exists an interrupt line from the hardware scheduler to the CPU, to notify the CPU about a required context switch. For the context switch, the software takes information from the hardware scheduler, accessible via the MMIO address registers.

The hardware scheduler supports up to eight external interrupts, which can be configured that the IRQ handler immediately interrupts the currently running task, or that the interrupt handler is inserted into the ready queue. Here, the ready queue is prioritized, according to the implemented scheduling policy. This leads to a defined postponing of the IRQ handling, and the currently running task can only be predictably interrupted, as aimed for real-time systems.

2.2.6. µC-OS-III HW-RTOS

Renesas sells their R-IN32M3 MCU family with an RTOS accelerator implemented in hardware [57, 58]. The implemented OS is named µC-OS-III HW-RTOS, and is based on Micrium's µC-OS-III [22]. Thus, almost all of the provided APIs from the software implementation are offered in hardware, too. This hardware implemented version improves the system performance compared to the pure software Micrium µC-OS-III implementation. The OS hardware implementation provides the task scheduling, OS resource management, timer processing, and syscalls that all are simultaneously executed in hardware besides to the code application code. In software, there is still the need of calling the syscalls and of performing the dispatching of the tasks on a context switch, which is initiated by an IRQ in the core.

Similar to the previous works, µC-OS-III HW-RTOS realizes the interface to the core with MMIO address registers. To perform a system call, the API sets the registers and a return value gives an error or the next task identifier. If a new task identifier is returned, the software will perform the context switch to schedule the new task.

The maximal number of tasks and OS resources is given by the MCU specification and cannot be increased or decreased. Although the accelerator supports resource management in hardware, it does not support priority inheritance [30] and deadlock protection mechanisms, as supported by the software OS.

2.2.7. xCore-200

The commercial xCore-200 [59] from XMOS offers a multi-core MCU for real-time and Digital Signal Processing (DSP) applications. The MCU contains at least two tiles; each one can implement eight cores. An interconnect enables the communication between the cores via different communication approaches.

XMOS implements many different RTOS features. Thus, the MCU allows scheduling real-time threads on different cores in a tile. Here, all the threads are event triggered, which means that the thread may wait for an event. If the event is triggered, the hardware automatically executes a context switch and resumes the triggered thread. The number of threads in a tile is limited to the number of cores. However, the compilation toolchain enables the usage of logical cores. Thus, the programmer could implement many different threads and the compilation toolchain conflates the threads to one thread per core.

XMOS's xCore-200 does not restrict the number of threads; however, if the compilation toolchain is unable to conflate the threads to fit into physical cores, the compilation will end up with an error. The number of resources, such as timers and locks, is limited. To exploit all the offered features of the MCU, the programmer has to use an extended `C` language.

2.2.8. OS Assistance in widely used MCUs

If we look into today's widely used Commercial Off-The-Shelf (COTS) MCUs, we can see that there exists almost no OS assistance.

The early and low-cost MCUs have no OS support at all. The newer MCUs have at least the support of a *user*-mode and *kernel*-mode environment. Here, the mode defines the privilege level, and may restrict the user-mode to execute some instructions or to access some registers. The user-mode is thought to execute code of the application, and the kernel-mode to execute code of the OS. This well-known and widely found existence of the two modes is used to prevent faults or to protect confidential data. Thus, the change from the user-mode to the kernel-mode should only be possible under defined conditions. One of them is initiated by a syscall instruction. The ISA implements an instruction to enter kernel-mode in an orderly way and then the computational unit jumps to a kernel address for executing kernel code. The purpose is to perform an OS API function, which must be handled by the OS (e.g., synchronization, task management). Another common way to switch from user-mode to kernel-mode is via an IRQ. The IRQ can asynchronously interrupt the currently running task and the OS performs the handling of the IRQ. The change back from the kernel-mode to the user-mode is usually done by a specific instruction that selects the mode, or the CPU keeps track of the last mode and with another specific instruction, the CPU changes to the tracked mode after executing the specific instruction.

Multi-threading technologies [60], which are found in almost every desktop computer today (e.g., Intel's Hyper-Threading [61]), are also increasingly found in today's embedded MCUs, such as in the Infineon's TriCore 2 [62]. The name multi-threading would suggest that the CPU is aware of thread handling (threads are OS constructs similar to tasks); thus, it seems to be an OS awareness feature. However, the multi-threading technology just pretends the availability of an additional logical core. This means that the OS sees the CPU as a multi-core architecture and the OS can assign the execution of code to the logical cores. Then, the CPU internally tries to use simultaneously all the available computational resources (e.g., ALU, FPU) to reduce the computation time. In the end, the hardware does not assist the OS in performing OS work, because it only provides a further logical core to which the OS can assign a task. Therefore, this technology is not an OS-aware feature in the sense of this thesis.

2.2.9. Summary and Difference to this Thesis

The mentioned projects partially or fully implement the OS in a hardware extension of the MCU. Through this hardware extension, the execution of OS functionalities is performed in a shorter and sometimes in deterministic time, which is preferred for real-time systems.

All of the mentioned works rely on the assumption that the number of tasks and other OS instances is fixed during system development. Some of the works synthesize the hardware exactly to the required number of OS instances (e.g., tasks), and others define a high number and leave some OS instances unused for later usage. However, the exact instantiation of OS instances restricts the adding of OS instances in the future, which is definitively required for future embedded computer systems. Otherwise, the arbitrary instantiation of OS instances negatively affects the resource utilization on an FPGA (or the size on a die for Application-Specific Integrated Circuits (ASICs)) as well as the power consumption. Moreover, the number of changeable OS instances in hardware would change the required area on a die with every configuration change. This complicates the mass production (i.e., ASICs), because the system optimization and validation process must be successfully performed for each individual configuration.

This doctoral thesis provides a solution to integrate OS awareness in an MCU, which avoids the restriction of fixing the number of OS instances at development time. Apart, the number of OS instances will not change the required area on a die and the *mosart*MCU must be optimized and validated only once, which makes it suitable to be implemented into an ASIC such as commercial MCUs.

2.3. Shared Stack Handling

As already introduced in Chapter 1, memory is a scare resource in most embedded systems. The stack memory is a memory that changes its size at run-time (see Section 1.1.1). To reduce the required overall embedded system's memory; and therefore the costs of the embedded system, the stack memory consumption is one part that must be efficiently managed.

In desktop and server systems, the allocation of stack memory to a task is enabled with the concept of address virtualization [63]. There, all the addresses in an application are virtual and a hardware component, namely the MMU, translates the Virtual Memory (VM) addresses to Physical Memory (PM) addresses. Thus, the stack does not have to be continues and the MMU can assign only that stack memory into the main memory (e.g., DRAM) that are required by the tasks at a moment (handled in fixed sized blocks). Unused blocks are released, or sparsely used blocks are swapped to another probably slower memory (e.g., flash). Thus, fast and costly main memory is not wasted. When a task requests new memory or the CPU accesses memory addresses which are not known by the MMU, the MMU interrupts the currently running task. Then, the OS handles this interrupt by internally allocating new memory to the task, and by configuring the MMU with the translation information (VM address to a PM address). The allocated new memory block has a constant size and is called *page*. If the page is not used anymore, the OS is able to remove the translation information and to free the page. Through the address virtualization and the controlled memory mapping by the OS, the MMU may allocate almost only that memory that is required by the task. For instance, if the stack memory grows, new pages are allocated; if the stack memory shrinks, the unused pages are deallocated. Furthermore, the approach can be used to isolate tasks from each other; thus, a task could not corrupt the system's data or other task's data [19].

If the application executes an instruction or accesses data from the VM space, the MMU looks up the VM address in the Translation Lookaside Buffer (TLB) and returns the PM (i.e., *TLB hit*). The TLB is a special cache memory for the MMU for storing the translation addresses. If the VM address is not located in the TLB (i.e., *TLB miss*), the MMU generates an interrupt. Then, the OS handles the TLB miss, by resolving the VM address to a PM address mapping in software (i.e., software memory manager), which will be than remembered by the TLB.

It is obvious that through an MMU, the CPU may be interrupted on every instruction fetch or data memory access, which enormously increases the Worst Case Execution Time (WCET) of a task. To avoid an interruption of the CPU if the VM address is not found in the TLB, the MMU in the MC68851 [64] or in the ARM's Cortex-A5 [65] uses a table-walker. Instead of resolving the VM to PM address in the software, the table-walker does this in hardware. It starts on a configured base address (stored in a register), searches in the data structure for the VM to PM translation, and updates the TLB with this translation. However, if the table-walker cannot find the correct VM address, still an interrupt is thrown to inform the OS to allocate a new page and to add the address mapping into the OS data structure. With the table-walker, the time for a TLB update on a TLB miss is reduced. Nevertheless, the time is non-deterministic because it depends on the length of the OS data structure.

To the best of our knowledge, no MMU implementation is able to achieve a predictable memory access time. For instance, in Ng *et al.*'s [66] MMU implementation, on a TLB hit the MMU takes 2 cycles and on a TLB miss 600 up to 227 000 cycles including the OS management. In Schamani *et al.*'s [21] configurable MMU implemented on a FPGA, a TLB hit takes 5 cycles and a TLB miss takes 11 cycles only to inform the OS. Furthermore, they show that the logic of the MMU consumes in average more than 50 % of the power consumption of the whole computational unit.

These properties (i.e., the non-deterministic behavior and the huge power consumption) lead to the fact that MMUs are avoided in embedded real-time systems. Apart, the MMU is not the only option to reduce the stack memory reservation, software approaches have been developed as well.

An efficient solution regarding minimum stack memory usage of concurrently running tasks is to use only one single stack for the whole application. The Stack Resource Protocol (SRP) [24] is often used to handle resource allocation. SRP sees the stack memory as a resource and allows using only one single stack, which can be shared by all tasks. Gai *et al.* [67] extended SRP to multi-core in which every core has only one common shared stack.

Dunkels and Schmidt [23] invented a multitasking approach based on stackless tasks, called protothreads, which are managed in the Contiki OS [9]. Contiki uses only a single stack for multiple tasks. If a task is scheduled, it may use the stack until it suspends itself. The programmer of the task must take care, that he or she backups *automatic variables* if they are still used after an API call that may suspend the task. Otherwise, if the task resumes the automatic variables, which were initial placed on the stack, may be corrupted by another scheduled task. Apart from self-suspension, tasks are scheduled in a non-preemptive way, which means that they run-to-completion. The reason is that if a task would preempt another task, the preempted task's stack pointer would not be on the top of the stack memory anymore once the task is resumed. Thus, the preempting task would overwrite other task's data.

A weaker approach of SRP is the preemptive threshold scheduling [68, 25] used in the ThreadX [69] OS. In this approach, the tasks, beside their nominal priority, have a threshold priority, which must be equal or beyond their nominal priority. Once a task is scheduled, its priority is risen to the threshold priority until it finishes its work. Thus, no task with priority lower than the threshold priority of the currently running task is scheduled by the OS, although a task's nominal priority is higher than the nominal priority of the currently running task. This approach creates non-preemptive task groups that are unable to preempt each other. Thus, the tasks in a task group are able to share a common stack.

Nevertheless, non-preemptive and partly non-preemptive scheduling would reduce the reactivity of the tasks or tasks in the task group, respectively. Whereas, for real-time systems the approaches require knowledge at the development time, which restricts dynamic application changes. Consequently, the time boundaries of real-time tasks are difficult to satisfy, making these approaches ineligible for dynamic real-time embedded systems.

Around 50 years ago, in [70], the authors proposed to allocate stack memory on the heap. This leads to a memory allocation and deallocation on the heap on every function prologue (entry) and epilogue (exit), respectively. Thus, the run-time overhead increased extremely. Consequently, different following works (e.g., [71, 72]) proposed the use of a code analyzer to reduce the number of memory allocations and deallocations. Nevertheless, run-time checks are required and an overhead is still present. Furthermore, if the heap memory management algorithm does not have a deterministic behavior, the approach would not suit well for real-time systems. In fact, predictable heap memory management approaches exist. For instance, [73] shows a quad-core CPU with an allocation and deallocation time of 28 cycles and 14 cycles, respectively. Still, this would notably increase the run-time overhead in every function prologue and epilogue.

In the project SenSmart [74], the aim is to have an adaptive stack management for MMU-less embedded systems. The idea is to use VM addresses for the stack memory that are translated by software. To support the software-based translation, a tool analyzes and modifies the compiled code between the compilation and linkage steps. After the modification, the code uses kernel functionalities for stack memory operations. Through the software stack handling the corresponding get, set, and reallocation operations for the stack pointer consume 45, 94, and 2326 cycles, respectively, on an ATmega128L.

Similar to the previously mentioned project, Yi *et al.* [75] proposed an approach to analyze the source code and to modify it at compile time. The modification uses an *on demand stack* library on every function prologue and epilogue. Here, the next free space on a global stack memory area is selected. The disadvantage of this approach is that the global stack memory area is fragmented with non-regular blocks and leads to an overhead, because of frequent calls to the library functions. Their WSN use case scenario shows a constants memory overhead and a non-constant run-time overhead of about 10 %.

Middha *et al.* [76] proposed a multitasking stack sharing approach without the use of VM addresses. Here, every task owns an individual stack. In every function prologue, a checker function checks for a potential stack overflow if the requested memory would overflow the individual stack. If it will not overflow, the requested stack memory will be allocated. Otherwise, if the stack memory would overflow, the task allocates a constant sized stack memory in another task's individual stack memory. Thus, unused stack memory of a task may be used by another task. Their evaluation show that without optimization, in every function call, the stack overflow check is executed. This results in an increase of the run-time and energy consumption by 23 % and 24 %, respectively. Thus, the authors proposed to use an optimization at compile time, which would improve the required stack overflow checks. Consequently, both overheads can be reduced to 3 %. However, the optimization does not allow all programming features such as recursive functions or function pointers. They compared their approach with an MMU, and the run-time overhead is much lower; however, their approach works with a non-deterministic time and therefore not suitable for hard-real time embedded systems.

In all mentioned works, a task that requires stack memory may block for a long time if stack memory is allocated to other tasks and no stack memory is available anymore (i.e., out of stack memory condition). None of the mentioned approaches collaboratively frees stack memory for that high priority task. In CoMem [77], the idea of collaboratively sharing memory is applied on the heap memory. The author proposed to divide the heap memory into blocks to which a resource is allocated. If a task requires heap memory, the task requests for the resource. If it is free, the memory block is allocated to the task. Otherwise, the task has to wait. Through the dynamic hinting approach [31], a lower priority task owning the resource is requested to release the memory. If the task follows the request, the memory will be allocated to the highest prioritized task that is waiting for the resource. However, the heap memory is organized and handled in a completely different way compared to the stack memory; therefore, CoMem cannot be applied to it.

2.3.1. Summary and Difference to this Thesis

The state of the art for efficient stack sharing in embedded real-time systems is improved by this thesis. The OS awareness in the MCU contributes to realizing the following two concepts:

Shared Real-Time Stack Management: The proposed solution for stack memory management supports, through the OS-aware support, to share the stack memory among *all* preemptive tasks with a predictable run-time behavior for stack allocation, deallocation, and access. Furthermore, the proposed solution does not restrict programming features as others do (i.e., recursive functions, function pointers).

Collaborative Stack: In the case, when the system runs out of stack memory, the traditional approach throws an overflow exception, and the OS has to solve this issue somehow. Normally, it resets the system or kills a task. The solution proposed in this thesis, demonstrates an approach to collaboratively share stack memory. This means that a lower prioritized task frees its collaborative stack memory if a higher prioritized task requires stack memory and no stack memory is available. Thus, the required stack memory can be reduced even further. Based on the predictable run-time behavior of the shared real-time stack management approach, the task's blocking time for stack memory is bounded; and the systems schedulability can be proven with a schedulability analysis.

2.4. Priority Inversion

This section investigates approaches to avoid or at least to time bound the priority inversion for IRQs, IPC communication, and in SoC buses.

2.4.1. IRQ Handling

To avoid the constant polling of an occurred event, such as for instance from a peripheral, processor cores offer a so called interrupt concept for the currently executing code. During the interruption, the event (i.e., IRQ) is handled in an ISR, which is injected into the execution flow. Such interruption can occur at an unpredictable time and this non-determinism affects the schedulability of real-time systems. The priorities of the IRQs are always higher than every task's priority. Often, this is desired; however, if the IRQ is meant to trigger a lower prioritized task compared to the currently running task, an OS-PI occurs.

Therefore, regarding interrupts in hard real-time systems, Stewart [78] proposes to disable all interrupts to avoid priority inversions. The same is proposed by Kopetz *et al.* [79] in the Mars approach, where all interrupts are disabled except the timer interrupt. The timer interrupt is used to schedule the task on specific times and to poll periodically the occurrence of an event (i.e., external event or peripheral). These solutions guarantee a deterministic execution of the code; however, the polling wastes CPU time for checking every external event or peripheral. Another disadvantage of static scheduling approaches are the difficulty in maintaining the software if new tasks would be added or the application requirements would change over time.

To better integrate IRQs into new software, Kleiman and Eykholt [80] propose to map all ISRs to tasks. In order to handle an IRQ in a task, the tasks are synchronized by synchronization primitives such as events or semaphores. In their publication, the goal was to improve the maintenance of server systems when integrating new software. The predictability was unimportant; and the approach is therefore unsuitable for real-time systems.

In [81], Leyva-del-Foyo *et al.* propose an IRQ handling approach for predictable interrupt handling on a conventional Personal Computer (PC) at software level. They introduce *interrupt service tasks* in which the interrupt handling is executed. Those interrupt service tasks are scheduled with the service tasks' priority such as regular tasks; thus, priorities of regular tasks and interrupt service tasks are unified. Through the mapping of an ISR to an interrupt service task, it is now possible to use all available OS synchronization mechanisms. Therefore, the interrupt service task may wait for a synchronization primitive, which is triggered by a universal ISR running at the lowest level in the OS. Their key idea is to introduce an *interrupt management component*, which is capable to enable and disable the IRQs of the hardware. Further, the component has an IRQ level, identifying the current priority and is able to set the priorities of all the interrupts. The component disables all the IRQs that have a lower priority compared to the IRQ level. The enabled IRQs are still handled in an ISR, actually in the universal ISR. The universal ISR notifies the OS of the occurrence of the event. The OS schedules the triggered interrupt service task if the scheduling policy allows it. However, the drawback of this approach is the high overhead of reconfiguring the interrupt controller on every context switch. To mitigate the overhead, the authors proposed, in [82, 83], the fusion of their approach with the *optimistic interrupt masking* approach [84]. Here, the idea is not to mask (i.e., disable) the interrupts on every context switch. If the priority of the IRQ level increases, no interrupt masking is performed. However, if an undesired interrupt occurs, the interrupt management component tracks the occurrence of the interrupt and masks all lower prioritized interrupts. Thus, a second undesired interrupt is prevented and the priority inversion is time bounded. The tracked interrupt will be caught up later on, when the priority allows it. Compared to their former implementation, the overhead is reduced but still there. Moreover, the timer IRQ must still be used in the traditional way with highest priority. The timer IRQ must trigger the OS either periodically or sporadically for setting the time of the next task in the timeout waiting queue.

In the project SLOTH [85], instead of moving the ISRs to tasks, all tasks are moved to ISRs. The base of SLOTH is the Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug (OSEK) [86] OS specification with the conformance class BCC1. The conformance class BBC1 specifies that the application can contain a maximum of eight tasks. All tasks have a unique priority, they cannot wait for an event, and they can only be activated by one source. With these restrictions, the interrupt system of a state-of-the-art MCU is suitable. The only requirement on the MCU is that IRQs can be prioritized and that the IRQs support at least eight priority levels. The ARMv8 [87] specification or the Infineon AURIX [88] (an automotive MCU) supports configurable priorities for IRQs; thus, their approach could be implemented there. Every IRQ is configured with a priority. To find out if it is allowed to interrupt the CPU, an IRQ arbitration unit checks the priority. If the priority allows an interruption, the corresponding handler (OSEK task or OSEK ISR) is executed. Thus, the scheduling is completely managed by the hardware, and implicit unification of the priorities is given. In this approach, the priority of a task could be set higher than the priority of the ISR. However, this is not done in SLOTH, because it would violate OSEK's BCC1 conformance class. The advantage of this approach is the priority unification of tasks, ISRs, and the small OS. In SLOTH, the OS consumes merely 200 lines. However, for future embedded computer systems, the BCC1 conformance class of OSEK restricts the system too much.

Gomes *et al.* [89] extended the interrupt controller of an ARM-compliant softcore with task priority awareness. Thus, the interrupt controller knows the priority of the currently running task and the priority of the IRQs. The 8-Bit IRQ priorities are configured by the task that waits for the IRQ. Therefore, the interrupt controller interrupts the currently running task only if the triggered task's priority exceeds the currently running task. The responsibility of the OS is to set the priority of the new scheduled task in the interrupt controller on every context switch. The authors have used an OS with a periodic interval timer, which is triggered periodically by a timer IRQ. The priority of this IRQ must be the highest, since otherwise the internal tick timer would omit some ticks. Further, their approach allows only one task to wait for a peripheral IRQ.

Another attempt to avoid or time bound an OS-PI caused by an IRQ is to use a second computational unit, which is responsible only for handling the IRQs. Thus, every core or processor would be able to support this approach in a multi-core system. However, most of the time, the computational unit for IRQ handling would be idle, and therefore, this approach would waste resources. To reduce the required space on the die for the IRQ handling computational unit, some MCU manufactures implement a coprocessor as the Freescale HCS12X [90]. These MCUs, beside their main core, contain a small RISC core, which preprocesses data for communication and handles the IRQs. Thus, an ISR could be concurrently processed on the coprocessor, which then notifies the main core about the handled ISR with a triggered semaphore. The main core or coprocessor do not take into consideration the priorities. Exactly this is considered in the work from Scheler *et al.* [91], based on the peripheral control processor of Infineon's TC1797 [92]. Here, the coprocessor handles the IRQ and sets a synchronization primitive for which a task is waiting on the main core. The main core is only notified by an interrupt, if the priority of the currently running task is lower compared to the priority of the triggered task. With this approach, there is no possibility for an interruption of the currently running task that would lead into a priority inversion. However, a hardware semaphore is required for synchronizing the main core and the coprocessor. This could also lead to priority inversions through a low priority task that allocates the hardware semaphore. Furthermore, on both computational units the code must be implemented which sometimes is based on different architectures. Thus, the developer must make himself/herself familiar with a new computer architecture, which may increase the development costs.

The mentioned IRQ handling approaches are based on MCUs without OS assistance in hardware. Partially or fully implemented OSs in hardware handle IRQs as follows: In the FASTHARD [51] project (see Section 2.2.1), the tasks may wait for an external IRQ by using an API call. Then, the IRQ priority is set to the priority for which the task is waiting. In case that the IRQ is triggered, the waiting task is inserted into the ready queue handled by the hardware, which schedules the task if necessary. Similar to FASTHARD, Silicon OS [52] (see Section 2.2.2) handles the IRQs simultaneously to the executing task. Therefore, the computational unit is notified to trigger a context switch if the scheduler selects a new task. In the SEOS project [54] (see Section 2.2.4), an interrupt posts a semaphore for which a task is waiting. The OS, partially implemented in software, is still responsible for the time management; thus, the timer IRQ must still interrupt the computational unit on a time IRQ. Renesas's µC-OS-III HW-RTOS [57, 58] (see Section 2.2.6) eliminates the interrupt handling in the same way as the mentioned hardware OS approaches. Here, the hardware triggers the semaphore, and then the hardware scheduler is responsible for scheduling the task if necessary.

In the commercial XMOS xCore-200 [59] MCU (see Section 2.2.7), the IRQ handling is performed with the idea of a logical core. Here, the tasks are waiting for an event. If the event, which could be either a software event or an IRQ, is triggered, the hardware scheduler may schedule the corresponding event handling code.

However, all of the mentioned OS hardware approaches limit the number of OS instances such as tasks, because of the hardware implementation of the task management.

2.4.2. IPC

IPCs are a helpful mechanism to transfer data between processes, or in our context, between tasks. The common way to realize an IPC is by software. For instance, in the automotive standard OSEK [86], the IPC is realized through events that signals a change in software. To transfer data, the developer has to implement a workaround, by using shared memory and the OSEK event mechanism. The avionic standard ARINC-653 [17] supports the event mechanism, too. In addition, this standard defines a queue approach, which makes it possible to transfer more than the change occurrence (i.e., event) to another task. Both software solutions relay on shared memory for the data transfer. Shared memory is on one hand problematic, because it must be synchronized. On the other hand, all the IPC involved tasks have to have unrestricted memory access, because they have to work on the OS's IPC instance. It is possible to prevent memory accesses with MMUs or Memory Protection Units (MPUs); however, those components do not allow arbitrary fine granular protected blocks, because this would explode the require space in the die. Therefore, there is the need of security mechanisms, which prevent the reading of potentially confidential information in the OS's IPC instance. Another drawback is that the traditional IPC calls are implemented as syscalls, which result in more context switches, which reduce the overall performance, and which even lead to possible OS-PIs. Especially for embedded RTOS systems, the IPC performance is a crucial property, because these RTOSs are often designed as microkernels with a heavy frequent IPC usage compared to monolithic kernels. In microkernels the tasks (and OS tasks) have to have to communicate between each other with a communication approach offered by the kernel (i.e., IPC), because shared memory may not be possible because the tasks' memories may be protected (e.g., with an MPU). In monolithic kernels however, shared memory is often used for IPCs, because all kernel functionalities are anyway in the same unprotected memory space.

More than 25 years ago, some research projects started to counteract the IPC performance problem. For instance, [93, 94] realized the IPC with a coprocessor. In their works, they show that the implementation of the IPC in hardware results in a huge performance increase. However, their target are server applications; thus, power consumption and space requirements in a die as well as a predictable execution time are of secondary importance. By contrast, these three properties are significant in embedded real-time systems.

Furunäs *et al.* suggested an IPC prototype [95] implemented in hardware for real-time applications. For supporting real-time applications, their implementation considers the priorities of the tasks. The hardware contains 32 slots in which the priority of the messages and the message references are stored. The message references are used to point to the content of the message, which is located in the RAM. The hardware extension is accessed with MMIO addresses, which are hidden by a provided API. However, the number of slots is limited to only 32 IPC channels.

A similar limitation can be found in IPC approaches for commercial computer architectures, such as ARM's IPC module [96] or NXP's Queue Manager [97]. The ARM IPC module offers a specific number of channels, called mailboxes. If a message is sent over a mailbox, the receiver core is signaled by an IRQ. However, the IRQ interrupts the current execution flow that could lead to an OS-PI. The NXP's Queue Manager is a powerful hardware extension that supports the transfer of data between all accelerators and cores in a SoC. The manager uses frames to store the content and descriptors to handle the buffers and frames. However, for low power embedded systems the power consumption and space requirement are too big.

To avoid the limitation of IPC channels, Srinivasan and Stewart proposed in [98] an IPC approach for real-time embedded systems. The structure of the approach is based on shared memory with two-levels. The first level is the local memory that every task owns. The second level is a global level, which is used to combine the data from the local levels. The key idea is to move the data between the two levels to exchange information between tasks. The local level is, in contrast to the global memory level, only used by the owner task. Thus, for the global memory level a synchronization primitive is required that increases the message passing overhead with the number of tasks. To reduce the overhead, the authors suggested an adapted Direct Memory Access (DMA) controller that transfers the data between the two layers. Therefore, the time to wait for the synchronization primitive is reduced. Their proposed approach enables a predictable execution time for the IPC; however, the double copying of the data between the two layers is still required. Here, the significant IPC transfer overhead affects the performance of the system. Another problem of this approach is the unprotected access to shared memory. Here, confidential information in the shared memory may be read by another untrusted task.

2.4.3. SoC Bus

The SoC bus within a processor is an interconnection between the cores, the peripherals, and the memories for exchanging data. The underlying technology significantly influences the behavior of the SoC. Thus, for real-time systems a deterministic bus is necessary to ensure an upper time bound for the data transfer. For a SoC with only one master, which is able to start a bus transfer, no concurrent access handling is required. The concurrent access handling is also called *arbitration*. Past embedded systems were still single-cores and this is the reason why the mostly used SoC buses for embedded real-time systems had not the need of an arbitration. However, in future, multi-core systems will become more important and used for real-time embedded systems; thus, arbitration will be required.

Today's commonly used SoC buses do not consider task priorities for the arbitration. For instance, the ARM AMBA AHB-Lite [99] or the Altera Avalon [100] specifications do not respect priorities when handling concurrent accesses. The royalty free Wishbone bus [101] specification does not even specify an arbitration in its specification. It only specifies that the arbitration technique must be defined by the end user.

The Processor Local Bus (PLB) bus [102], developed by IBM, specifies a SoC bus with priority awareness. The specification defines two additional lines, which allow four priority levels. The arbiter is responsible for granting access to the requester with the highest priority, and for letting the other concurrently operating units wait. The limitation of four priority levels restricts the mapping to task priorities, so it is mainly used to assign statically a priority to a core instead of reflecting task priorities. The ARM AMBA specification [103] uses four lines for the arbitration logic. Thus, it supports up to 16 different priorities, which is also insufficient in many cases to map all the task priorities.

The mapping of task priorities is impossible with the state-of-the-art SoC buses; thus, there exist different arbitration algorithms. In statically assigned priorities for a master [104], the highest prioritized master is able to starve other masters in the system. Apart, the static priority does not correspond to the priority of the task, which is accessing the slave. Thus, the mismatch may result in a priority inversion.

To avoid a statically assigned priority and to enable a more dynamic approach, a probabilistic approach could be used. The lottery [105] arbitration approach considers the last requesting master and a random number for the arbitration. This solution cannot guarantee an upper time bound. Therefore, it is not suitable for real-time systems, since the time predictability is not given.

To time bound the waiting time for an access, the Time Division Multiple Access (TDMA) [106] approach could be used. It assigns a time slot for each master, in which it can communicate with a slave. Because of its predictability, this approach is commonly used in different fields of real-time systems. However, the utilization of the SoC may be low, because not every slot is used. Thus, bandwidth is wasted and the WCET of the high-prioritized task is unnecessarily increased, because it must wait for its next assigned slot to transfer its data.

To increase bandwidth, the Round Robin [104] approach improves the TDMA by antedating transfers if a slot is free. Like in TDMA, the next allowed master is selected statically and no priorities are considered. The WCET of Round Robin is the same as for TDMA; however, the Average Case Execution Time (ACET) may be improved.

2.4.4. Summary and Difference to this Thesis

This thesis improves the state of the art by avoiding or at least time bounding the priority inversion problem for the following fields, with the support of OS awareness in hardware:

IRQ Handling: This thesis proposes a new IRQ handling approach based on the OS awareness of the *mosart*MCU. Thereby, it does not restrict the number of tasks compared to related works. Furthermore, this solution allows more than one task to wait for an IRQ, similar to the producer-consumer pattern, which is not supported by any of the mentioned related works. The most important improvement for real-time systems is the priority unification of IRQs and tasks for time bounding the OS-PI problem. This approach also eliminates the high priority timer IRQ, which is still required in all of the mentioned related works if the OS is only partially implemented into hardware. There, the software OS must still enable a high-prioritized timer IRQ to handle properly the scheduling and to update the internal timing, and these could lead into OS-PIs.

IPC: The solution for an IPC presented in this thesis is based on the IRQ handling approach. Thus, the IPC is implemented in hardware, which increases the performance and timely bounds the occurrence of an OS-PI. All mentioned related works only allow the usage of a limited number of IPC channels, which is not the case for the proposed IPC. Besides, the proposed IPC approach can be used to transfer confidential information, because the sender task does not need access to shared memory. Furthermore, it works not only in a single core, but it can also be used for communications between tasks on different cores.

SoC Bus: To support a real-time communication in multi-core systems, this thesis presents a deterministic SoC bus that allows to fully utilizing the bus bandwidth. In comparison to the SoC buses of the related works, the proposed solution avoids priority inversions on the SoC layer. This is achieved by making the SoC bus aware of the task priority without using additional wires for the priority.

3. *mosart*MCU: The Operating System-Aware Microcontroller

This chapter presents the *mosart*MCU, a multi-core real-time MCU with OS awareness. Section 3.1 introduces the terminology and the assumptions made for the MCU. Section 3.2 introduces the general idea of an OS-aware MCU. Section 3.3 presents strategies that use OS awareness in single-cores. One of the novel approaches in this work is an efficient stack memory usage. Other novel approaches are the time bounding of OS-PIs for IRQs and IPCs. Section 3.4 deals with multi-core systems. Further, this chapter presents techniques to avoid priority inversions on the SoC bus, and presents RIC that integrates some proposed single-core techniques in multi-core systems.

3.1. Terminology and Assumptions

In this doctoral thesis, we assume that a multi-core system contains several masters (in this thesis only cores), each one defined as master $m \in M$, which are able to communicate with a slave $s \in S$ at a time. The multi-core system follows the partial scheduling approach [107], which means that on every core a separate instance of an OS is running. The OS is a multi-tasking system, in which a *task* $\tau_{m,i} \in T_m \subset T$ describes a software unit executing on the core m. The OS is responsible for scheduling the tasks according to its scheduling policy, which in our case follows the RM [108] policy. Thus, we define for every task $\tau_{m,i}$ a priority $p_{\tau_{m,i}} \in \mathbb{P}$ with $\mathbb{P} := \{0, 1, \ldots, max\_priority\}$. For convenience, the priority of a task $\tau_{m,i} \in T_m$ is not assigned to another task in the taskset T_m (i.e., $\forall \tau_{m,x}, \tau_{m,y} \in T_m \wedge \tau_{m,x} \neq \tau_{m,y} : p_{\tau_{m,x}} \neq p_{\tau_{m,y}}$). Now, for convenience, a task that is allocated to master m, can be represented as $\tau_{m,p} \in T_m$ with the unique priority p in the taskset T_m. According to the partial scheduling strategy, a task executes only on the allocated core. The executing task is defined as the running task $\tau_{m,run} \in T_m \subset T$. In case of a single-core system, we simplify the symbol of a task to $\tau_{p_\tau} \in T$, and the running task to τ_{run}. Each task may be a part of a *ready* queue or of two different *waiting* queues (i.e., generic waiting queue or timeout waiting queue). To support a membership in two queues, every task has a previous pointer $prev_\tau$ and a next pointer $next_\tau$ for generic queue handling and a timeout previous pointer $tprev_\tau$ and timeout next pointer $tnext_\tau$ for timeout queue handling.

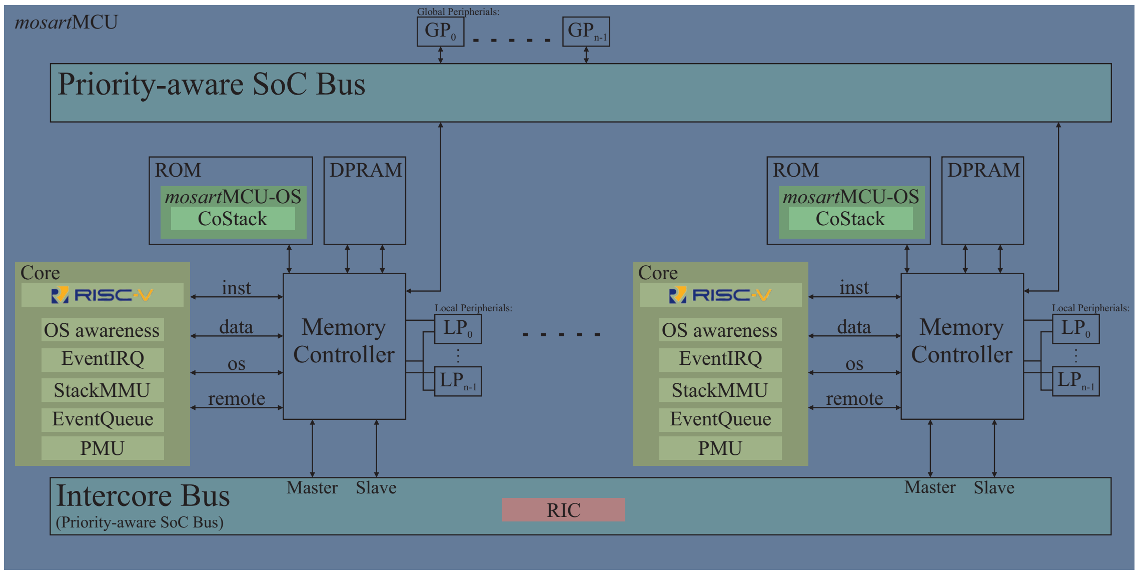


Figure 3.1.: Architecture of the *mosart*MCU.

A task $\tau \in T$ is allowed to be executed on the computational unit, either periodically or sporadically (with the constraints to be conform to a RM system). The period or the shortest time between release time and the following deadline (i.e., interarrival time) is defined as D_τ of the task τ. The longest computational time, the time that a task τ is executing on the computational unit between its release and end of the period, is called WCET and it is defined as C_τ. The WCET depends on the chosen compiler, core architecture or frequency, and many other system specifications. If a task τ is prevented to execute, because of a priority inversion, it is called *blocked*. The maximum time that a task τ is blocked within its interarrival time by a lower priority task, is defined as the blocking time B_τ.

In the next sections, more system specifications are introduced on demand. Before the OS-aware extensions are introduced, Figure 3.1 gives an overview of the system and shows the integration of the proposed OS-aware extensions of this thesis. The next two sections provide the basic understanding of the *mosart*MCU and *mosart*MCU-OS, in which all the proposed OS-aware functionalities are implemented.

3.1.1. *mosart*MCU

The idea behind the project *mosart*MCU is to introduce OS awareness into single and multi-core MCUs for real-time systems. Therefore, the full name of the *mosart*MCU is *Multi-core Operating System-Aware Real-Time MCU*. The aim of introducing OS awareness is to assist the OS in its execution. The OS awareness should help in reaching properties that are unfeasible with pure software solutions or only possible with a huge computational effort. The problems mentioned in Section 1.1 are only a subset of all unresolved problems in today's embedded systems. However, in this thesis those mentioned problems are mitigated and partly avoided by using the OS awareness in the *mosart*MCU.

The *mosart*MCU is based on vScale[1], which is a Verilog [109] implementation of the zScale, both offered by the University of Berkeley. The basis of vScale is the RISC-V [110] processor architecture. RISC-V is an open ISA specification that has being developed since 2010, with the aim of having an open ISA for education, research as well as for commercial projects. The architecture is inspired by the MIPS [111] architecture, which can be noticed in the RISC-V's ISA.

The RISC-V specification [112] specifies the ISA for 32, 64, and 128-Bit wide RISC architectures. The developer is free to choose one of the three different bit widths. RISC-V's target applications are embedded systems as well as desktop and server systems. Thus, RISC-V is not restricted to a specific domain, as it is usually the case with other computer architecture specifications. To support so many different fields, the specification defines what the instructions have to do, but not how they must be implemented. Thus, it is up to the developer to implement computer architecture's speedup techniques such as pipelining, out-of-order execution, etc. Furthermore, the specification defines instructions, which are optional to implement. Only the Integer Instruction Set, which contains the basic instructions for arithmetic, jumps, branches, loads, and stores are obligatory to implement. According to the RISC philosophy, only the load and store instructions are accessing the memory, while all other instructions are using the CPU registers and a possibly immediate (i.e., constant) value. The defined ISA extensions such as multiplication and division, atomic instructions, floating point arithmetic, bit manipulation, etc. are optional to implement. In the Integer Instruction Set some special instructions are specified to accesses the Control Status Registers (CSRs). The CSRs are registers to configure core functionalities and to get information out of the core. These registers are accessed by an atomic read-modify-write operation. For some CSRs, the access may be restricted, since access is only allowed for privilege modes. The RISC-V ISA specifies three different privilege modes: *user*-mode, *supervisor*-mode, and *kernel*-mode. The user-mode is the least privileged mode while the kernel-mode has full privileges. Some instructions are restricted to be executed only by higher privileged modes.

As mentioned, in the RISC-V ISA specification it is up to the developer how to implement the computational unit. For *mosart*MCU, the vScale is chosen due to the following reasons: First, the required logic resources are small by implementing only the 32-Bit specification, the multiplication and division extension, a pipeline of only three stages, and only the user-mode and kernel-mode. Second, for real-time it is necessary that the instructions are executed in a predictable way. For example, out-of-order execution would not be acceptable because of an

[1]`https://github.com/ucb-bar/vscale`

unpredictable behavior. Third, at the beginning of 2016 when the *mosart*MCU project started, available implementations primarily aimed on desktop, servers, or embedded systems that already have been extended with application specific extensions. Thus, at that time, vScale was the only suitable basis for implementing the OS awareness into an MCU.

3.1.2. *mosart*MCU-OS

mosart MCU -OS
Multi-core Operating System-Aware Real-Time Operating System

The *mosart*MCU-OS is an RTOS for the *mosart*MCU and is also compatible with the basic 32-Bit RISC-V ISA. It is a microkernel supporting events, IPC, resource management, and task management (see OS API in Appendix B.2). The tasks are scheduled by the RM scheduler. A static priority is assigned for every task and the scheduler selects a task according to its priority and state. The task state is either *running*, *ready*, or *waiting*. In the waiting state, the task waits for a synchronization primitive (e.g., event or resource) or/and a specific timeout. Otherwise, the task is ready. If it is the highest prioritized task among all ready tasks, its state is running and it is executing on the core. Beside the standard functionalities, *mosart*MCU-OS supports all the OS-aware extensions of the *mosart*MCU, which are discussed in the next sections.

3.2. General Idea of Operating System Awareness in MCUs

For most commercial systems, the MCU development and the OS development are conducted by different organizations. However, to support all the MCU features in the OS, the development of both areas must conflate. The *mosart*MCU project [113] follows exactly this idea by developing the OS together with the MCU, following an MCU/OS codesign approach. Figure 3.2 depicts one realization of this idea. The MCU offers beside its general ISA an OS-aware ISA that is triggering the OS-aware extensions. For supporting these extensions, the MCU and the OS must have a common knowledge about the OS data structures. This happens by developing and merging parts of the MCU and the OS together. This also leads that functionalities of the OS are moved into the MCU while retaining the same flexibility as in traditional OSs.

In this thesis, the *mosart*MCU introduces OS awareness that has the knowledge of OS internals. Here, the MCU is aware of the OS internal data structures and is able to access them by reading and writing. This leads to the fact that the MCU is able to have the same knowledge as the OS. To support this idea and to avoid performance restrictions caused by accessing the internal OS data structures using the same data bus, the *mosart*MCU introduces a new connection to the data memory by using a Dual-Port-RAM (DPRAM) as depicted in Figure 3.3. A DPRAM is a memory supporting up to two memory access ports that can be simultaneously used. This

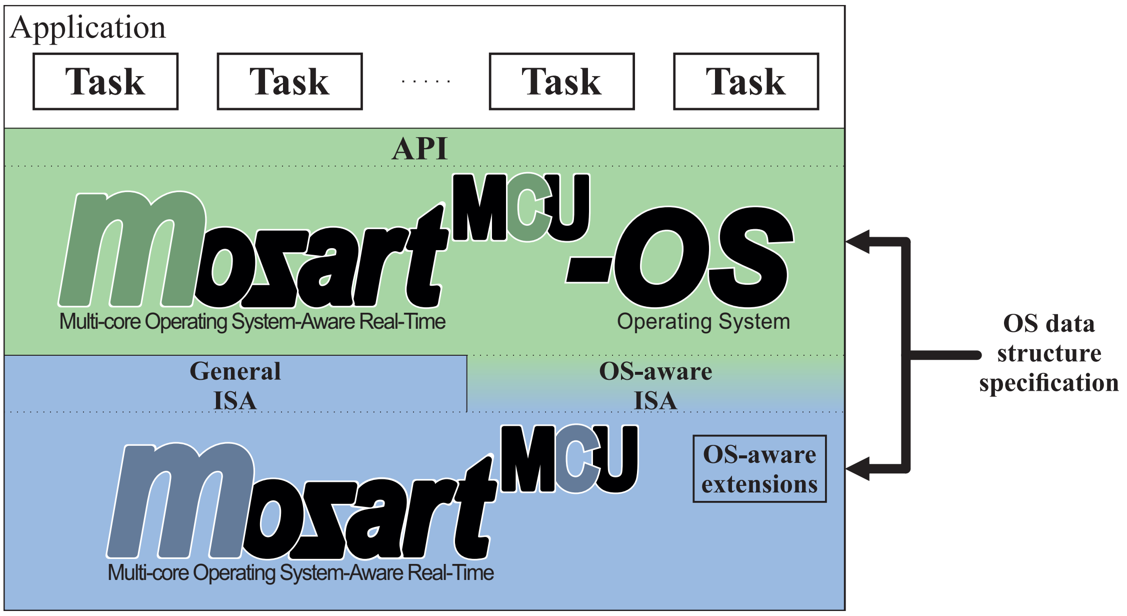

Application
Task
Task
Task
Task
API
mosartMCU-OS
Multi-core Operating System-Aware Real-Time
Operating System
General ISA
OS-aware ISA
mosartMCU
Multi-core Operating System-Aware Real-Time
OS-aware extensions
OS data structure specification

Figure 3.2.: The architecture of an MCU/OS codesigned embedded system, which breaks up the independently developed layers of an OS and an MCU.

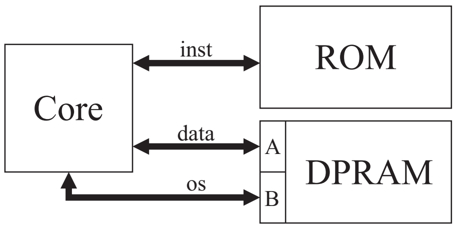


Figure 3.3.: The base architecture idea of supporting OS awareness in the MCU.

kind of memory is widely used in today's desktop computers, embedded systems, and it is an integrated part of most FPGAs. Thus, the currently running task and the OS awareness extensions are able to access simultaneously the data memory. There is no synchronization required, because the OS awareness part accesses only OS related data, and the currently running task only its own data. The memories are not using caches, because this would introduce unpredictable memory access times, which is undesired for real-time systems. This is also the reason why many real-time applications disable the offered cache memory of the MCU. Compared to a desktop or server computer system, an embedded system's memory is small and directly connected to the computational unit. This allows to have a data memory access in a few cycles (e.g., data memory read takes 2 cycles in the megaAVR [42] and 3-12 cycles in the Aurix [88]). For most of the applications, the internal memory in an embedded system is sufficient; therefore, usually no additional external memory is required. However, if the internal memory is insufficient, then the *mosart*MCU can be extended with an external

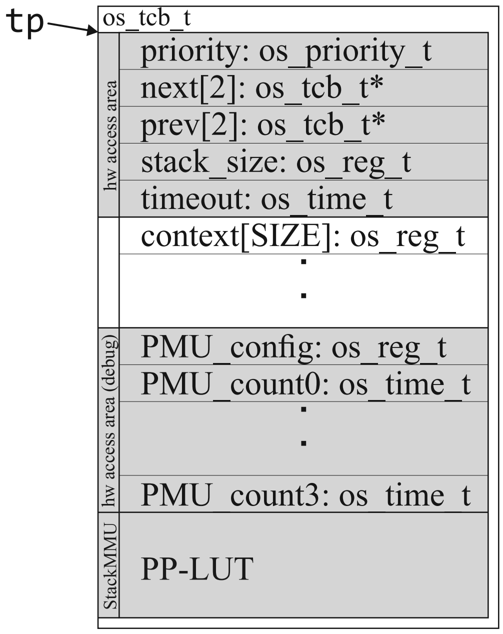


Figure 3.4.: TCB structure referenced by the CPU register `tp`.

memory, but the OS instances still have to be stored in the DPRAM. Consequently, the internal memory can be kept small, and the external memory can be large but slower. In any case, the external memory should be deterministic in order to obtain a predictable execution time of the software. The next section shows two examples on using the OS data connection to introduce OS awareness into the *mosart*MCU.

3.2.1. Task Priority Awareness in the MCU

The RM scheduling policy forces the developer to assign a static priority to every task according to the tasks' deadlines. The assigned priority of the task is stored in its TCB. The TCB is an OS data structure that stores all the required information that the OS needs to handle properly the task. In the RISC-V specification, the CPU register `tp` is named *thread pointer* and available individually for each core. This pointer is not managed by the compiler, but instead, the OS may use it to optimize some thread functionalities. In the context of this thesis, the thread pointer is renamed to *task pointer* and points to the TCB of the currently running task $\tau_{m,run}$; thus, the *mosart*MCU is already aware of the currently running task. Figure 3.4 shows the *mosart*MCU-OS's TCB of the currently running task referenced by the CPU register `tp`. With the awareness of the OS data structure and the usage of the `os` connection to the data memory, the *mosart*MCU is able to read and modify the data content of the currently running task. To let the OS be aware of the currently running task's priority $p_{\tau_{m,run}}$, the hardware is triggered to read the task's priority in the TCB at any `tp` change. The calculation of the TCB memory

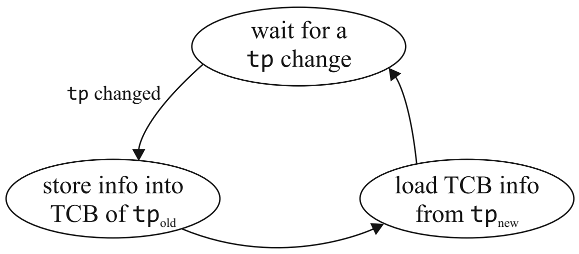


Figure 3.5.: *mosart*MCU state machine on a `tp` change.

addresses is straightforward, due to the known offset and the used base address from `tp`. The currently running task's priority $p_{\tau_{m,run}}$ is used for adding priority awareness in the multi-core *mosart*MCU, too. Thus, the master m's priority $\pi_m \in \mathbb{P}$ is defined as $\pi_m := p_{\tau_{m,run}}$. With this slight extension, every core is aware of the currently running task's priority, which can be used internally to perform operations concurrently to the running code. Therefore, the MCU acquires OS awareness through the knowledge of the TCB structure. Due to the MCU/OS codesign approach, the structure is known and can be adapted for the MCU and the OS in case of need. The OS awareness is not limited to the knowledge about the currently running task's priority. The next section shows how the *mosart*MCU's OS awareness can be used for an implementation of a performance monitoring unit.

3.2.2. Performance Monitoring Unit

In [114], we have implemented a Performance Monitoring Unit (PMU) that exploits the OS awareness in the *mosart*MCU. The PMU is a hardware module, which can be used by the OS to measure the computational time between two specific instruction addresses, or to count some internal events (e.g., IRQs while a task is running). The PMU divides the measurements into three groups. The first group, the *generic counter*, is used as a global counter, for times and events, independent of the current task; thus, for the whole system and is activated at startup. This kind of counter can also be found in other conventional computer architectures to measure the performance. The other groups implement the OS awareness of the *mosart*MCU. The second group, the *task-aware counter*, is used to measure and count a measurement for each running task, whereby it is for all the same measurement type. In the last group, the *counter assigned to a task*, on each task an individual measurement can be measured or counted. The task-aware counter and the counter assigned to a task reduce the required number of performance counter instances in hardware, by sharing them. In traditional PMU's, for each measurement or count a separate instance is needed, which increases the size of the required logic. In this PMU, the shared PMU instances have to be configured to measure the desired measurements (e.g., task's time running on the core) and the OS-aware MCU automatically reconfigures each PMU instance on a task change (recognized by a `tp` change), as shown in Figure 3.5. The *mosart*MCU does not just read the OS data structure, as for instance for the priority, it can also overwrite the OS data structure by storing data into the old TCB, referenced by the old `tp` value, on a `tp` change. In case of the PMU, the *mosart*MCU stores the configuration and the measured time or counter of the PMU instance. After finishing these operations, it loads the configuration of the new scheduled task. As shown in the previous section, the *mosart*MCU reads the currently running task's priority p_{run} and the PMU instance configurations of the new scheduled task's TCB. In the *mosart*MCU the writing and reading of those data consumes some constant cycles. Thus, the *mosart*MCU compensates the measured time or counters if while the write or read operation some time measurements or counts are missed, respectively.

3.3. Operating System Awareness in Single-Cores

This section introduces novel concepts that are handling known issues in single-core embedded systems by using the OS awareness of the *mosart*MCU. Section 3.3.1 deals with efficient stack memory reservation. Section 3.3.2 introduces a concept for handling IRQs to time bound the OS-PIs. Lastly, Section 3.3.3 presents an IPC approach which time bounds OS-PIs, too.

3.3.1. Stack Handling

As already discussed in the previous chapters, embedded real-time systems are commonly MMU-less and for every task $\tau \in T$ an individual stack is allocated. This decision leads to a inefficient stack memory reservation, due to temporarily unused stack memory in the individual stacks. The utilization of a task τ's stack is defined as the stack usage $\sigma_\tau(t) \in \mathbb{N}_0$ at time $t \in \mathbb{N}_0$, while the maximum stack memory required by a task τ is denoted by ς_τ. To avoid an out of stack memory condition, the individual stack size has to be at least the size of the maximum required stack memory ς_τ for a task τ. Hence, the totally allocated stack memory $\tilde{S}$ for all tasks is represented with the following equation:

$$\tilde{S} := \sum_{\tau \in T} \varsigma_\tau = \sum_{\tau \in T} \max\{\sigma_\tau(t) \mid t \in \mathbb{N}_0\} \tag{3.1}$$

StackMMU, presented in [115], counteracts this stack memory overbooking by introducing address virtualization of the stack memory addresses. This means that every stack memory access is performed through a VM address and StackMMU converts this address to a PM address. There is no need of a continuous physical memory space for the stack memory anymore per task; instead, StackMMU divides the whole stack memory into equal sized blocks, but provides a continuous virtual addressed stack space to the tasks. Thus, no compiler or stack usage modification are required and the totally allocated stack memory $\hat{S}$ of the StackMMU approach can be computed by

$$\hat{S} := \max\left\{\sum_{\tau\in T}\sigma_\tau(t) \mid t \in \mathbb{N}_0\right\} \leq \sum_{\tau\in T}\max\{\sigma_\tau(t) \mid t \in \mathbb{N}_0\} =: \tilde{S}, \tag{3.2}$$

without considering StackMMU administration overheads. With this triangle inequality, the required totally allocated stack memory of the StackMMU approach is lower or equal (in the worst case) compared to the traditional approach that instantiates an individually continuous physically addressed stack for every task.

To achieve the goal of reducing the overall stack memory reservation while keeping the stack memory access time deterministic, as required by the real-time systems, the OS awareness of the *mosart*MCU comes into account.

The idea is based on Infineon TriCore's [116] Context Save Area (CSA), which uses a constant sized block for storing registers of the task context on an interrupt or function call. However, for the stack the TriCore architecture has still stack pointer to allocate local variables on the stack. The StackMMU approach uses a stack area that is divided into same sized blocks, called *pages*. All the pages are allocated one after another, between the start and the end addresses of the stack area, as defined by the CSRs `start` and `end` on OS startup. The OS is responsible for initializing the reference pointers between the pages, leading to a linked list of initially unused pages. The CSR `free` pointer references the first free page, initialized by the OS. This CSR is then internally adapted by the *mosart*MCU at run-time. Another CSR, the task stack size register `tss` makes the *mosart*MCU aware of the maximal stack size that the currently running task τ_{run} can use. This CSR is filled on a `tp` change from the scheduled task's TCB similar to the task's priority. Thus, at the task initialization, the OS must configure for every task the maximal required stack memory, which can be analyzed with a static code analyzer. An example of the shared stack memory structure is depicted in Figure 3.6, where every task's TCB is extended by the Page Pointer Look Up Table (PP-LUT). The PP-LUT is used to store all the physically addressed references of the allocated pages to the corresponding task. Thus, the PP-LUT's size depends on the maximum supported stack memory size for the task. StackMMU accesses the PP-LUT to translate the VM address (growing from high addresses to low addresses) to a PM address (i.e., address in the stack area). With equation 3.3, the page base address of the corresponding VM address is read from the PP-LUT.

$$pm\_addr\_base = \text{PP-LUT}\left(\left\lfloor\frac{\texttt{tss} + vm\_addr - \texttt{end}}{\texttt{page\_size}}\right\rfloor\right) \tag{3.3}$$

Equation 3.4 calculates the offset inside the page with the VM address.

$$pm\_addr\_offset = vm\_addr \bmod \texttt{page\_size} \tag{3.4}$$

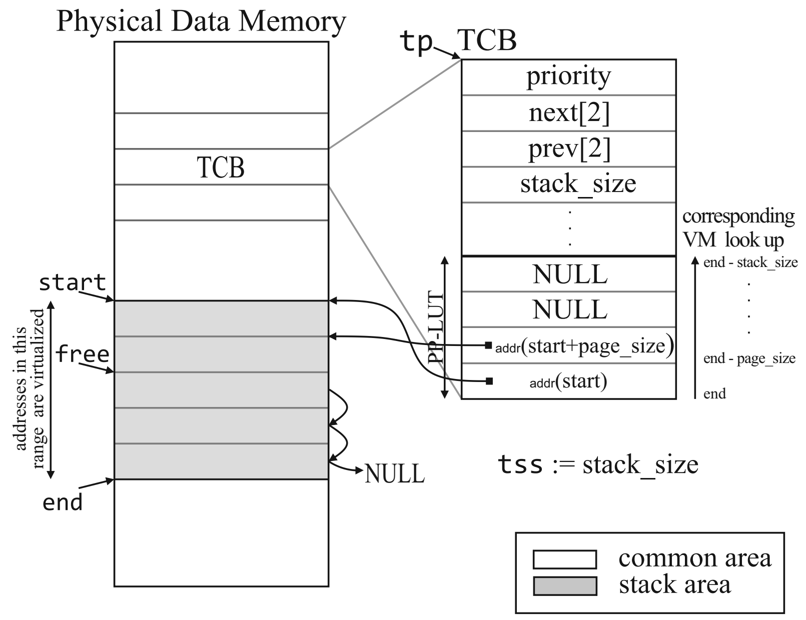


Figure 3.6.: Memory layout of the StackMMU approach including the structure of the TCB.

Finally, with the two equations 3.3 and 3.4 the PM address is calculated, as shown in equation 3.5.

$$pm\_addr = pm\_addr\_base + pm\_addr\_offset \tag{3.5}$$

The whole calculation is performed for a stack memory read or write. First, the base address of the corresponding page is read; second, the PM address is accessed in the stack area. StackMMU modifies the PP-LUT for allocating or deallocating a page to or from a task, respectively. Figure 3.7 shows the memory accesses for the stack growth, access, and shrinkage operations; whereby, the growth and shrinkage operations are only performed if the stack grows or shrinks across the page size alignment, respectively. In the first phase, all the StackMMU operations are reading the reference from the PP-LUT. In the second phase, the references are updated or the stack memory access is performed. If the stack growths, the old address in the `free` CSR is stored in the PP-LUT and the `free` CSR is updated with the next free page address. If a memory access is performed, the address *pm_addr* is read or written. In case of a stack shrinkage, the `free` CSR is updated with the freed page, which is stored in the PP-LUT, and the old address of the `free` CSR is stored in the new freed page. By contrast, the memory access outside the stack area is handled as a regular PM address.

In StackMMU, the stack growth and shrinkage size is limited to one page. Therefore, a modified compiler generates code that is limiting the size of every stack growth and shrinkage in one instruction to the page size. In MultiStackMMU [117], this limitation has been removed by automatically assigning and deassigning the required pages one after another. This leads to

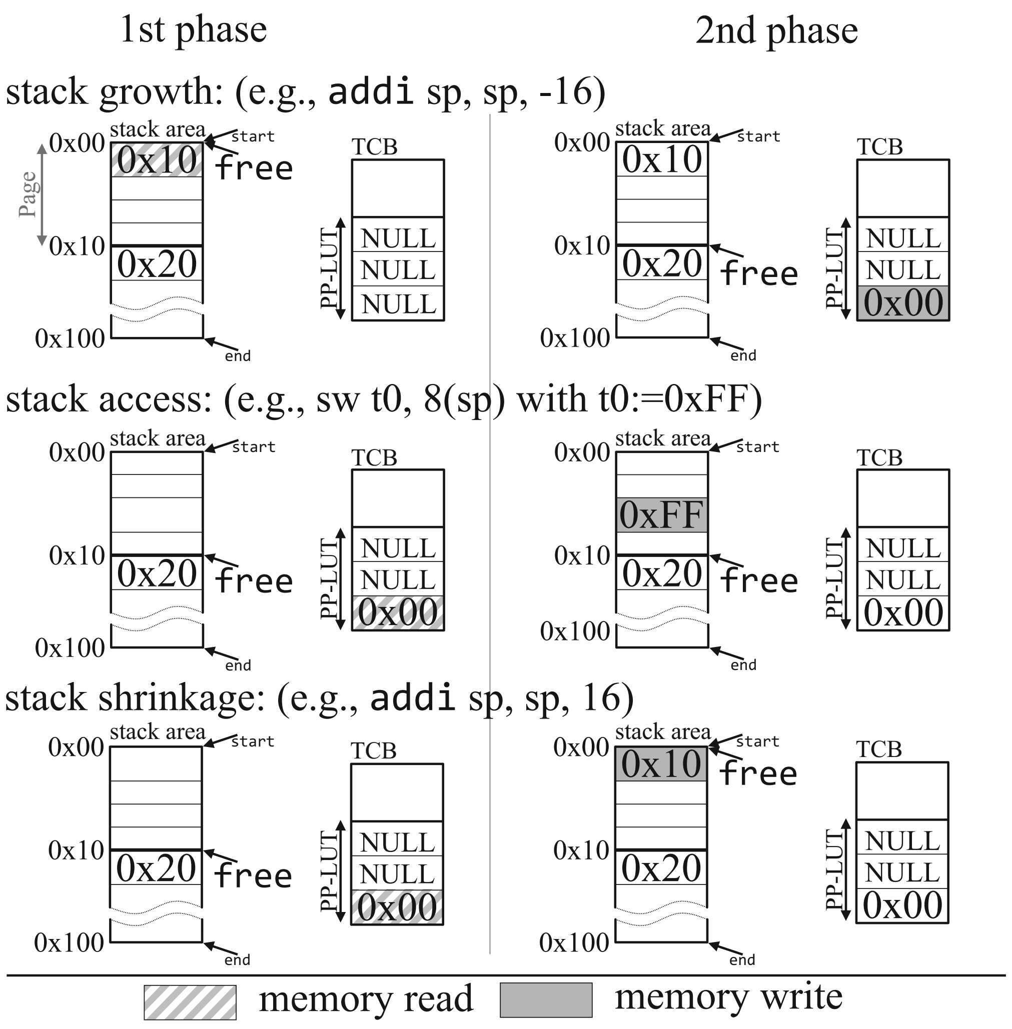
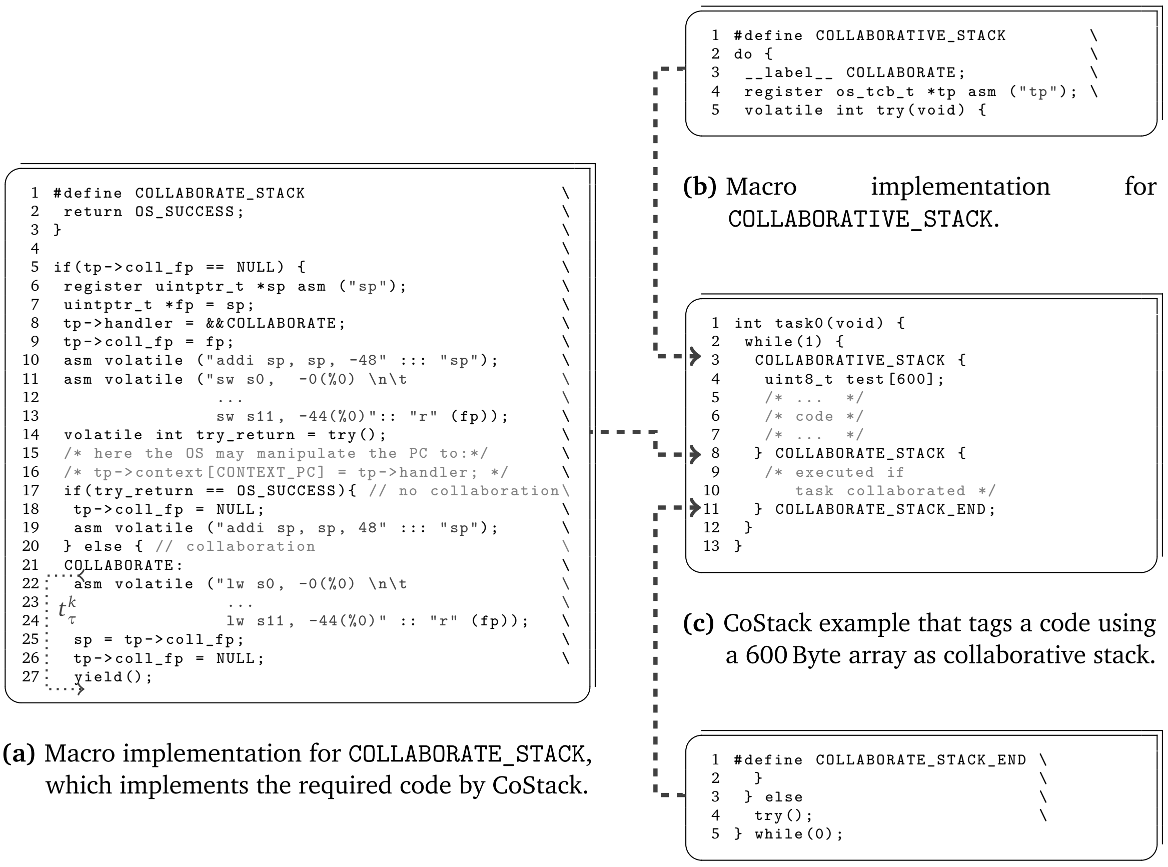

1st phase
2nd phase
stack growth: (e.g., addi sp, sp, -16)
stack access: (e.g., sw t0, 8(sp) with t0:=0xFF)
stack shrinkage: (e.g., addi sp, sp, 16)
stack area
start
end
free
TCB
PP-LUT
Page
0x00
0x10
0x20
0x100
0xFF
NULL
memory read
memory write

Figure 3.7.: StackMMU phases for stack growth, access, and shrinkage. The stack area is defined with the CSRs `start` and `end` and the next free pages' reference is stored on the first word in each page.

a longer execution time for growing and shrinking the stack memory; however, the execution time is still predictable if the underlying memory access executes in a predictable time, such as in the *mosart*MCU. The time predictability of MultiStackMMU is based on StackMMU that requires for a stack growth, shrinkage, or access only one further memory access.

StackMMU aims to reduce the totally allocated stack memory $\hat{S}$, but it cannot avoid an out of stack memory condition due to insufficiently allocated memory for the stack area. To avoid an out of stack memory condition, the currently used stack memory

$$U(t, \texttt{page\_size}) = \sum_{\tau \in T} \left\lceil \frac{\sigma_\tau(t)}{\texttt{page\_size}} \right\rceil \cdot \texttt{page\_size} \qquad (3.6)$$



```
#define COLLABORATE_STACK                              \
 return OS_SUCCESS;                                    \
}                                                      \
                                                       \
if(tp->coll_fp == NULL) {                              \
 register uintptr_t *sp asm ("sp");                    \
 uintptr_t *fp = sp;                                   \
 tp->handler = &&COLLABORATE;                          \
 tp->coll_fp = fp;                                     \
 asm volatile ("addi sp, sp, -48" ::: "sp");           \
 asm volatile ("sw s0,  -0(%0) \n\t                    \
                ...                                    \
                sw s11, -44(%0)":: "r" (fp));          \
 volatile int try_return = try();                      \
 /* here the OS may manipulate the PC to:*/            \
 /* tp->context[CONTEXT_PC] = tp->handler; */          \
 if(try_return == OS_SUCCESS){ // no collaboration\
  tp->coll_fp = NULL;                                  \
  asm volatile ("addi sp, sp, 48" ::: "sp");           \
 } else { // collaboration                             \
 COLLABORATE:                                          \
  asm volatile ("lw s0, -0(%0) \n\t                    \
                 ...                                   \
                 lw s11, -44(%0)" :: "r" (fp));        \
  sp = tp->coll_fp;                                    \
  tp->coll_fp = NULL;                                  \
  yield();
```

(a) Macro implementation for `COLLABORATE_STACK`, which implements the required code by CoStack.

```
#define COLLABORATIVE_STACK          \
do {                                 \
 __label__ COLLABORATE;              \
 register os_tcb_t *tp asm ("tp"); \
 volatile int try(void) {
```

(b) Macro implementation for `COLLABORATIVE_STACK`.

```
int task0(void) {
 while(1) {
  COLLABORATIVE_STACK {
   uint8_t test[600];
   /* ...  */
   /* code */
   /* ...  */
  } COLLABORATE_STACK {
   /* executed if
      task collaborated */
  } COLLABORATE_STACK_END;
 }
}
```

(c) CoStack example that tags a code using a 600 Byte array as collaborative stack.

```
#define COLLABORATE_STACK_END \
  }                           \
 } else                       \
  try();                      \
} while(0);
```

(d) Macro implementation for `COLLABORATE_STACK_END`.

Figure 3.8.: Macros for the collaborative stack sharing approach with a usage example.

is not allowed to exceed the totally allocated stack memory $\hat{S}$:

$$\forall t \in \mathbb{N}_0, \quad U(t, \texttt{page\_size}) \overset{!}{\leq} \hat{S} \tag{3.7}$$

However, what happens if an out of stack memory would occur? Then, MultiStackMMU would recognize this. Then the next question is how to handle this issue? In [117], we presented an extension of MultiStackMMU named CoStack that supports an ever bigger reduction of the totally allocated stack memory $\hat{S}$. Here, the idea is to define collaborative stack memory that will be handed over to a higher prioritized task if the higher prioritized task cannot continue because of an out of stack memory condition. Figure 3.8c shows, how a developer tags collaborative stack memory by using macros in the code. All the macro itemizations are demonstrated in Figure 3.8. The macro `COLLABORATIVE_STACK`, shown in 3.8b, is the starting point to tag a collaborative stack memory section. It opens an inner function in which the developer's code is inserted. The macro `COLLABORATE_STACK` (see Figure 3.8a) is the main part of the collaborative handling. Here, the inner function is closed and it will be checked if a collaborative frame pointer (i.e., `tp->coll_fp`) is already set. If so, an already upper function call tagged (i.e., nested collaborate call) a collaborative section and so the inner function is called. Otherwise, the executing task stores the label address (i.e., `COLLABORATE`) as well as the collaborative frame pointer in the TCB for collaborating. Furthermore, the callee saved registers are saved (lines 11-13 in Figure 3.8a) to be possibly restored in case of collaborating. Then, the inner function is called. Unless a higher prioritized task runs in an out of stack memory condition and the OS schedules a collaborative task, the collaborative task's saved registers are thrown away. Otherwise, the callee saved registers are restored and all the collaborative stack memory is released (line 25 in Figure 3.8a). Then, the task yields for returning to the OS, where the OS schedules the highest prioritized ready task. The computational time for releasing the collaborative stack memory by a task τ is the collaborate time t^k_τ. The macro in Figure 3.8d concludes the collaborative stack handling.

With hardware assistance, the CoStack approach is performed as follows: If a task requests stack memory, the hardware checks if enough stack pages are available. In the *mosart*MCU, the hardware is aware of the number of free pages and knows the requested pages for the requested stack memory size. If enough pages are available, they are allocated to the task according to the MultiStackMMU approach. Otherwise, no pages are allocated to the task and a *collaboration exception* is thrown. The exception is handled in the OS, which stores the number of requested pages into the currently running task's TCB. Then, the general OS operation is performed. Before the OS dispatches the (probably new) scheduled task, it checks if the (probably new) scheduled task requests more stack pages than available. If it does not request more stack pages than available, the scheduled task is executed. Otherwise, the OS searches for a lower prioritized task in the ready queue that provides collaborative stack memory, which is visible from the task's TCB content. If a collaborative task is found, the OS schedules it and executes the mentioned CoStack release functionality (lines 21-27 in Figure 3.8a).

To calculate the savable stack memory for each task τ, the collaborative stack memory $\kappa_\tau(t)$ at time $t \in \mathbb{N}_0$ is defined. With this information, the available collaborative stack size K_τ at time t for task τ can be calculated as:

$$K_\tau(t) = \sum_{\forall i: p_{\tau_i} < p_\tau} \left\lceil \frac{\kappa_{\tau_i}(t)}{\texttt{page\_size}} \right\rceil \cdot \texttt{page\_size} \tag{3.8}$$

In CoStack, lower prioritized tasks will voluntarily release stack memory for a higher prioritized task if the higher prioritized task gets into an out of stack memory condition. The available collaborative stack size K_τ helps to reduce the CoStack's totally allocated stack memory $\hat{S}\prime$

(compared to equation 3.7) by offering collaborative memory which decreasing the actually currently used stack memory $U(t, \texttt{page\_size})$:

$$\forall t \in \mathbb{N}_0 : \quad \max_{\forall \tau \in T} \{U(t, \texttt{page\_size}) - K_\tau(t)\} \overset{!}{\leq} \hat{S}\prime \leq \hat{S} \tag{3.9}$$

MultiStackMMU performs time predictable due to its underlying memory access and due to the approach itself. For real-time systems, this behavior is desired to get non-pessimistic schedulability analyses and no priority inversion can occur if enough stack pages are available. In CoStack however a high priority task could be prevented from executing due to a low priority task i.e., a priority inversion occurs through the collaborative release of the collaborative stack memory. Thus, a higher prioritized task may be blocked. The time that a higher prioritized task τ gets blocked is bounded by the blocking time

$$B_\tau := \sum_{\forall i : p_{\tau_i} < p_\tau} \max_{\forall t \in \mathbb{N}} \{t^k_{\tau_i}(t)\} \tag{3.10}$$

The blocking time B_τ is the sum of the longest collaborate time $t^k_{\tau_i}(t)$ of all lower prioritized tasks $\tau_i \in \{\tau_x \in T \mid p_{\tau_i} < p_\tau\}$. With the blocking time B_τ, the task τ's WCET C_τ, and the period or interarrival time D_τ, a schedulability test can be performed for an RM scheduling [30] with $n = \#T$:

$$\frac{C_{\tau_0}}{D_{\tau_0}} + \ldots + \frac{C_{\tau_{n-1}}}{D_{\tau_{n-1}}} + \max\left(\frac{B_{\tau_0}}{D_{\tau_0}}, \ldots, \frac{B_{\tau_{n-1}}}{D_{\tau_{n-1}}}\right) \leq n(2^{\frac{1}{n}} - 1) \tag{3.11}$$

The blocking time increases the system load, and that may exceed the threshold of a feasible scheduling. Here, the assumption is made, that the OS is not interfering the tasks. However, this assumption is not feasible, because the OS has an impact on the system load. In the next section, a new approach is introduced that considers OS's priority inversions by time bounding all OS-PIs; and therefore, enables the possibility to consider the OS also in a schedulability analysis.

3.3.2. IRQ Handling

Priority inversion is a big issue in embedded real-time systems, because a lower prioritized task could block a higher prioritized task. Not only could a task directly block the execution of a higher prioritized task, but also it could indirectly block a higher prioritized task through the OS, such as shown with OS-PIs introduced in Section 1.1.2. There, the OS performs some work for a low priority task and blocks the execution of a high priority task. IRQs are also executed in the OS context. Therefore, if the IRQ is addressed for a low priority task, a high priority task is blocked because the OS is executing work for a low priority task. Thus, an OS-PI is the result.

With EventIRQ [118], the OS-PI issues can sometimes be avoided but can always at least be bounded to an upper time. For that, all the ISRs, which are handlers for the IRQs, are mapped to respective tasks and all the IRQs are mapped to OS events. An event $e_m \in E_m \subset E$ is a synchronization primitive, assigned to the master m that signals a task $\tau \in T_m$ of a condition change. For single core systems, and following, the event is simplified to the event $e \in E$. On a triggered event e, only the highest prioritized task from the event queue q_e is notified. The event queue q_e is defined as follows:

$$\forall \tau_k, \tau_l \in q_e \land \tau_k \neq \tau_l \land p_{\tau_k} > p_{\tau_l}: \quad q_e := (\ldots, \tau_k, \ldots, \tau_l, \ldots). \tag{3.12}$$

Thus, all the tasks waiting for the event e are sorted by descending priority and the head task $h_e := first(q_e)$ is the first task in the event queue q_e for the event e.

Now, let us define for every IRQ $i \in I$, with $I := \{i_0, ..., i_{|I|-1}\}$, an event $e_i \in E$. For each IRQ an own event is defined and a task processes this event for handling the IRQ. The interrupt vector table is transferred to an *event vector table* that consists of all the events e_i for handling the IRQs i. For modeling the IRQ handling approach in a traditional way, for each IRQ a task with priority beyond all non-IRQ handling tasks must be defined. This task waits for the event e_i (triggered by the hardware on IRQ i) and implements the ISR code. If that code is executed, the task will again wait for the event e_i.

To avoid an OS-PI, or at least to time bound it, EventIRQ uses the OS awareness of the *mosart*MCU. With the additional connection to the data memory, the *mosart*MCU is able to perform some data memory accesses in parallel to the software execution, without interfering the currently running task. Through the priority awareness of the *mosart*MCU, EventIRQ is able to decide, depending on the priorities, if the event must be handled immediately or if it can be deferred. For managing this, the task $\tau\prime$ is appended to the pending task list ϕ if an event e_i triggers the task $\tau\prime$ (i.e., event e_i queue's head task h_{e_i}) waiting for the event e_i at time $\tilde{t}_{\tau\prime, e_i}$:

$$\begin{aligned} &\forall i, j \in I, \exists e_i, e_j \in E \land \forall \tau_k \in q_{e_i} \land \forall \tau_l \in q_{e_j} \land \tau_k \neq \tau_l \\ &\land \left(\left(\tilde{t}_{\tau_k, e_i} < \tilde{t}_{\tau_l, e_j} \right) \lor \left((\tilde{t}_{\tau_k, e_i} = \tilde{t}_{\tau_l, e_j}) \land (i < j) \right) \right): \quad \phi := (\ldots, \tau_k, \ldots, \tau_l, \ldots). \end{aligned} \tag{3.13}$$

To handle EventIRQ properly, *mosart*MCU internally uses the pending tasklist tail pointer $ptt := last(\phi)$, which allows to append triggered tasks to the pending task list ϕ. Due to this appending, EventIRQ is aware of the chronological order of tasks triggered by an event. After appending the triggered task $\tau\prime$ to the pending task list ϕ, EventIRQ checks if the currently running task τ_{run}'s priority p_{run} has lower priority compared to the triggered task $\tau\prime$'s priority $p_{\tau\prime}$ to avoid unpredictable interruptions by a lower prioritized task. The following and Figure 3.9 demonstrate the steps performed in hardware by EventIRQ to handle an IRQ (whereby $\emptyset$ demonstrates the $NULL$ pointer):

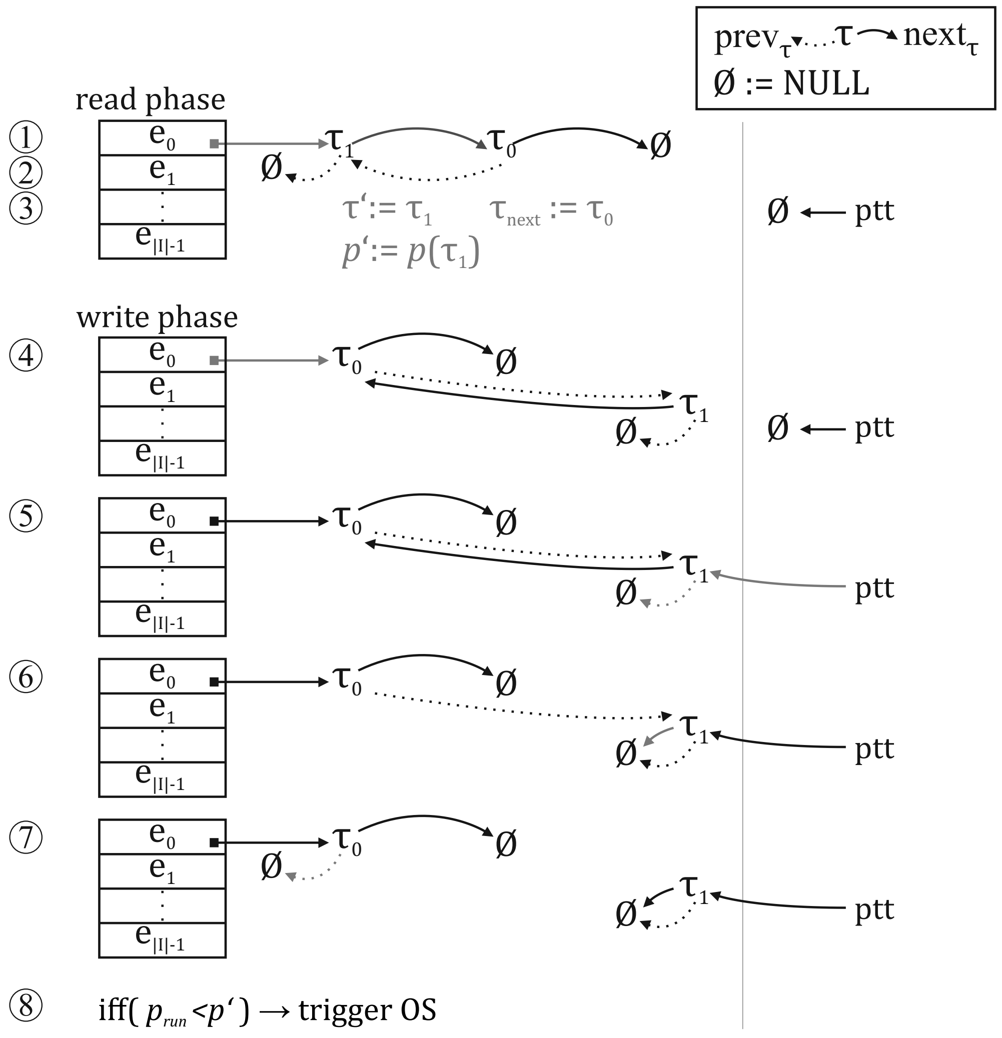


Figure 3.9.: EventIRQ example for the triggered IRQ $i_0 \in I$ (gray colored indicates a read or write access to the os port, i.e., channel B of the DPRAM).

① If an IRQ $i \in I$ is triggered, the hardware extension internally saves the triggered task $\tau\prime$, which is the head task h_{e_i} of the event queue q_{e_i}. EventIRQ accesses the event queue q_{e_i} by using the event vector table.

$$\tau\prime := h_{e_i}$$

② The next waiting task for the event e_i (i.e., $second(q_{e_i})$) is read if the event queue q_{e_i} is not empty. We define the next waiting task $\tau_{next} \in T$ as follows:

$$\tau_{next} := \begin{cases} next_{\tau\prime} & \text{if } \tau\prime \neq \emptyset \\ \emptyset & \text{if } \tau\prime = \emptyset \end{cases}$$

③ The triggered task's priority $p\prime \in \mathbb{P}$ is read from the TCB. If no task is triggered (i.e., $\tau\prime = \emptyset$), the priority will be set to the lowest priority.

$$p\prime := \begin{cases} p(\tau\prime) & \text{if } \tau\prime \neq \emptyset \\ 0 & \text{if } \tau\prime = \emptyset \end{cases}$$

④ The event vector table of event e_i is updated with the next waiting task τ_{next}, that becomes the new head of the event e_i.

⑤ The triggered task $\tau\prime$ is appended to the pending task list ϕ if task $\tau\prime$ is not $\emptyset$: First, the previous pointer of the triggered task is updated with the old pending task list tail pointer:

$$prev_{\tau\prime} = ptt \text{ if } \tau\prime \neq \emptyset$$

Second, the pending task list tail pointer ptt is updated with the triggered task $\tau\prime$:

$$ptt = \tau\prime \text{ if } \tau\prime \neq \emptyset$$

⑥ The triggered task τ' is now the tail of the pending task list ϕ and the next pointer must be adapted:

$$next_{\tau\prime} = \emptyset \text{ if } \tau\prime \neq \emptyset$$

⑦ The event queue q_{e_i} now points to the next task τ_{next}, which becomes the new head of the event queue. Thus, the previous pointer of the next task τ_{next} must also be adapted:

$$prev_{\tau_{next}} = \emptyset \text{ if } \tau\prime \neq \emptyset$$

⑧ Finally, the running task τ_{run} will be preempted, if and only if (iff) the triggered task's priority $p\prime$ exceeds the currently running task's priority p_{run}:

$$\text{iff}(p_{run} < p\prime) \rightarrow \text{trigger OS}$$

The OS is also involved in the EventIRQ approach by performing some work in software:

- If the OS is called by an interrupt, the triggered task $\tau\prime$ has a higher priority compared to the currently running task τ_{run}. The OS removes the last appended task from the pending task list ϕ (i.e., triggered task $\tau\prime$), inserts it to the ready queue, and schedules it, because it has now the highest priority in the ready queue.
- If the OS is called by a syscall, the syscall may change the currently running task τ_{run}, which may lead to a change of the currently running task's priority p_{run}. It could happen

now that the new scheduled task has a lower priority than the previous task (e.g., task calls a sleep syscall). In this case, some tasks in the pending task list ϕ might have a higher priority compared to the new scheduled task in the ready queue. Therefore, the OS catches up all tasks in the pending task list ϕ and inserts them into the ready queue (sorted by priority). Now, the OS considers all deferred tasks and the OS will schedule the next task according to the OS scheduling policy and priority.

A new IRQ cannot be handled while the OS is changing the internal OS structure, because this may lead to a race condition. To counteract that, either a synchronization primitive for the OS data structure must be used or EventIRQ has to be restricted to work only if the *mosart*MCU is in the user-mode.

EventIRQ does not only support the handling of regular IRQs, as in previous related works. It also supports the timeout event $e_t \in E$ (i.e., timer peripheral used for the system timer) and software events $e_s \in E$.

Due to the OS support for waiting for a specific time by a task, the timeout must be handled properly: The timeout queue q_{e_t} is not sorted by the task's priorities, but according to the task's timeout value. There, a trigger on the timeout IRQ does not select the highest prioritized task, but the task with the closest timeout. EventIRQ defines the timeout event $e_t \in E$ with the corresponding timeout queue q_{e_t}:

$$\begin{gathered}\forall \tau_k, \tau_l \in q_{e_t} \wedge \tau_k \neq \tau_l \wedge \left(\left(\hat{t}_{\tau_k} < \hat{t}_{\tau_l}\right) \vee \left(\left(\hat{t}_{\tau_k} = \hat{t}_{\tau_l}\right) \wedge \left(p_{\tau_k} > p_{\tau_l}\right)\right)\right) : \\ q_{e_t} := (\ldots, \tau_k, \ldots, \tau_l, \ldots)\end{gathered} \tag{3.14}$$

The timeout queue q_{e_t} sorts all the tasks according to their timeout $\hat{t}$. The timeout defines the time as long as a task sleeps or waits for an event. If the task is ready or running, it is not part of the timeout queue q_{e_t}. EventIRQ performs the same steps such as the previous mentioned ones for a regular IRQ, if the timeout event e_t is triggered. Instead of the pointers $prev_\tau$ and $next_\tau$ the timeout pointers $tprev_\tau$ and $tnext_\tau$ are updated, respectively, in hardware. Furthermore, for the timeout event e_t, the hardware extension sets in the hardware timer peripheral the new head task h_{e_t}'s timeout value. Thus, there is no need of setting the timeout value in software and this avoids unnecessary context switches into the OS, as required in previous related works.

EventIRQ allows using the event functionality not only for IRQs; but also for software events e_s. For making that possible, the *mosart*MCU extends the ISA with the instruction `sev src1`, whereby the source register `src1` points to the event e_s's Event Control Block (ECB) address. The instruction triggers the same functionality as in the regular EventIRQ approach; but instead of operating on the event vector table it operates on the event's OS instance.

EventIRQ avoids, in many cases, an OS-PI by processing internal and external events concurrently to the normal execution flow. EventIRQ theoretically completely avoids OS-PIs by design due to the avoidance of unnecessary context switches, the priority unification (i.e., task, IRQ, and OS), and because of the procedure itself. In practice however, EventIRQ cannot completely avoid OS-PIs: If EventIRQ starts to handle a triggered IRQ, it is not possible to handle a second IRQ before the previous one is completely processed. Thus, a possible IRQ addressed to a higher prioritized task must wait for finishing the previous IRQ handling in hardware. The reason is, that the functionality in the EventIRQ hardware extension requires some cycles and can only be computed sequentially. However, the time is time bounded: EventIRQ has a deterministic execution in hardware, where the upper time for handling an IRQ is known. Another upper time bound is caused by the OS accessing the internal OS data structures. To avoid race conditions, it is not possible to let the OS and EventIRQ access the data memory simultaneously. Thus, a synchronization leads to mutual exclusion when accessing OS data structures. However, this time can be measured (better computed) and considered in the schedulability analysis.

Idea for an OS-aware schedulability analysis

In past schedulability analyses approaches, the OS and interrupts were not considered. However, with the elimination of unpredictable interruptions by a lower prioritized task, the IRQs as well as the OS can now be considered for a non-pessimistic schedulability analysis. In EventIRQ the idea is, that all ISRs are now regular tasks with a regular priority according to the RM's priority selection approach (i.e., shorter deadlines have a higher priority). The currently running task will only be interrupted if the IRQ handling task's priority have a higher priority as the currently running task. Therefore, the interrupt handling task is considered as a normal task in the schedulability analysis, with an interarrival time D_τ that is depending on the shortest time of two consecutive IRQs of which the task τ may wait. The avoidance of unpredictable interruptions by lower prioritized tasks allows to consider also the OS in the schedulability analysis. Thus, if the OS performs operation for a task in the privileged mode, as well as the time that the OS needs to schedule the tasks, can now be considered as additional computation time for the task (e.g., $C\prime_\tau := C_\tau + t^{OS}_{schedule} + t^{OS}_{task\_work}$). Whereas, with the traditional IRQ approach each interrupt request must be considered, to which often follows the OS and may results in many context switches, pessimistic schedulability analysis, and consequently in an overdesigned embedded system. This OS-aware schedulability analysis is not limited to EventIRQ; the same idea can also be applied for the IPC approach presented in the next section.

3.3.3. Inter-Process-Communication

An event is a synchronization primitive, which notifies the highest prioritized task waiting for an internal or external condition change. There is no possibility to transfer an additional information, such as a pointer or integer, without a workaround (e.g., shared memory synchronized by events). With EventQueue [119], this drawback is eliminated. EventQueue is an

IPC approach that aims to avoid or time bound an OS-PI for transmitting data by a hardware extension based on the previously proposed EventIRQ.

EventQueue enables the communication from many tasks to a single task, by a unidirectional buffer. For transmitting data, a queue $\rho \in Q$ of the queueset Q, is required. The queue ρ is defined as:

$$\rho := (e_{\rho_s}, e_{\rho_r}, s_\rho, read_\rho, write_\rho, r_\rho, b_\rho) \tag{3.15}$$

It is a tuple of the events e_{ρ_s} and e_{ρ_r}. Both are used to notify the (possible many) senders and the receiver about the queue's state changes; i.e., the senders or the receiver are suspended as long as the queue is full (i.e., $full_\rho = true$) or the number of requested data r_ρ is not reached, respectively. The size s_ρ represents the number of items in the buffer b_ρ, where the data is stored. The indices $read_\rho$ and $write_\rho$ are used as indexes for reading and writing the content in the buffer b_ρ, respectively. For notifying the receiver task about received data over the queue ρ, the requested size r_ρ (with $1 \leq r_\rho \leq s_\rho - 1$) is considered; the receiver is only notified (by the event e_{ρ_r}) if the buffer length l_ρ is the same as the requested size r_ρ. The buffer length l_ρ represents the number of items stored in the buffer b_ρ. The buffer length l_ρ is calculated with the indices $write_\rho$ and $read_\rho$ as follows:

$$l_\rho := (write_\rho - read_\rho) \bmod s_\rho \tag{3.16}$$

EventQueue performs the transmission of the data in hardware. To support a hardware transmission, the *mosart*MCU ISA is extended with the instruction `qwr dst, src1, src2`. The first source register `src1` defines the Queue Control Block (QCB) address (i.e., a queue OS instance address), and the second source register `src2` the data content to transmit. The destination register `dst` is used to return the success of the instruction; successful if the data is stored in the buffer, or failure if the buffer is full. Figure 3.10 shows the single steps performed by the instruction in hardware:

① The size s_ρ of the queue ρ is read from the addressed QCB.

② Reading of the index write $write_\rho$ of queue ρ from the addressed QCB.

③ Reading of the index read $read_\rho$ of queue ρ from the addressed QCB.

④ Reading of the requested size r_ρ of queue ρ from the addressed QCB.

⑤ If the buffer condition $full_\rho$ is false, the index $write_\rho$ is incremented and stored in the QCB. The destination register is filled with a value indicating success and the EventQueue continues with the next step. Otherwise, the index write $write_\rho$ is not updated, the destination register is filled with an error code (i.e., MCU/OS codesigned *buffer full* error code), and the instruction is finished. To avoid race conditions, the computational unit is not allowed to continue within the first five memory operations. Thus, the pipeline is stalled for 4 cycles.

```
① read $s_\rho$
② read $write_\rho$
③ read $read_\rho$
④ read $r_\rho$
⑤ if $full_\rho$ then
      dst := ERROR
      abort
  else
      dst := SUCCESS
      $write_\rho := (write_\rho + 1) \bmod s_\rho$
  end if
⑥ $b_\rho[write_\rho] :=$ src2
⑦ if $r_\rho = l_\rho$ then
      set event $e_{\rho_r}$ {following handled by EventIRQ }
  end if
```

pipeline stalled

Figure 3.10.: Pseudocode performed by the instruction `qwr dst, src1, src2` in hardware.

⑥ If the queue ρ is not full, the content in the second source register will be stored to the buffer b_ρ at the index $write_\rho$.

⑦ If the buffer length l_ρ reaches the requested size r_ρ, EventQueue triggers the receiver event e_{ρ_r} that is located in the QCB of the queue ρ. After that, the *set event* approach of EventIRQ is executed.

With a pure software solution, either the OS handles the appending of data in the queue, which would lead to possible OS-PIs, or the sender task must have full access to QCB's data structure, which would require access to possible confidential data in the buffer. However, EventQueue transfers the data to the destination buffer in hardware. Through the hardware execution, the sender does not have to enter in the kernel mode that could produce an OS-PI, e.g., a high prioritized task send data to a low prioritized task. Furthermore, the sender task does not require access to the QCB's data structure; therefore, the receiver could put the QCB in a memory protected region, and only the hardware and the receiver task access the QCB's data structure memory. If the buffer length l_ρ reaches the requested size r_ρ, EventQueue, based on the EventIRQ approach, signals the destination task of this occurrence. Then, EventIRQ is responsible for properly handling the event and for avoiding or at least for time bounding the OS-PI. Figure 3.11 shows the code (queue creation, send, and receiver implementation) in the *mosart*MCU-OS for the mentioned EventQueue approach. With this approach, EventQueue is an IPC that enables secure IPC and moreover avoids OS-PIs by design required by real-time systems.

The proposed EventIRQ and EventQueue concepts work on a single-core *mosart*MCU. However, the computational power demand will increase even more for future embedded systems; therefore, the next section shows how to extend EventIRQ and EventQueue to multi-core systems by using the OS awareness of the *mosart*MCU.

```c
// wrapper for the ASM instruction
int send_queue(os_queue_t *q, os_reg_t d) {
  os_reg_t ret;

  asm volatile ("qwr %0,%1,%2 \n\t"
                : "=r" (ret)
                : "r" (q), "r" (d) );
  // q->e_r is triggered if required by EventQueue
  return ret;
}

/*
  q in protected memory
  q##_event in non protected memory
*/
#define send_queue_until(q, data, size, t) \
  intern_send_queue_until(q, &(q##_event), data, size, t)

// sycall - executed in user-mode!!!
int intern_send_queue_until(os_queue_t *q, os_event_t *e_s, os_reg_t *data, os_reg_t size, os_time_t t) {

  //non-kernel user-mode function
  clear_event(e_s);

  while(size) {
    if(send_queue(q, *data) == OS_SUCCESS) {
      data++;
      size--;
    } else { /* buffer is full */
      /* if event is set while no task waits
         -> immediately returns OS_SUCCESS */
      if (wait_event_until(e_s, t) == OS_TIMEOUT)
        return OS_TIMEOUT;
    }
  }
  return OS_SUCCESS;
}
```

```c
#define CREATE_QUEUE(name, size)                          \
  os_event_t NON_PROTECTED_MEM name##_event;              \
     /* (e_s, e_r , s , read , write , r , b ) */         \
  os_queue_t PROTECTED_MEM name = {name##_event,          \
      NONSET_EVENT, size + 1, name.buf, name.buf,         \
      0, {0, REPEAT(CONSTANT,size,0) } }

int syscall_wait_queue_until(os_queue_t *q, os_reg_t *data, os_reg_t size, os_time_t t) {
  signed long diff;

  // calculates number of items
  diff = (signed long)q->write
         - (signed long)q->read;
  diff >>= 2;   // bufferitem is 32 bit wide
  if(diff < 0)
    diff += q->size;

  if((os_reg_t)diff < size) {
    //There are not enought data in the buffer yet
    q->requested_data = size;

    if (syscall_wait_event_until_new(q->e_r, t) == OS_TIMEOUT)
      return OS_TIMEOUT;
  }

  while(size--){  // read out the buffer
    *data = *(q->read);
    q->read++;

    if(q->read == (os_reg_t*)&q->buf[q->size])
      q->read = (os_reg_t*)&q->buf[0];

    /* notify a sender task or set the event */
    set_event(&(q->e_s));
   }

  return OS_SUCCESS;
}
```

Figure 3.11.: *mosart*MCU-OS code for supporting the creation of a queue and the sending and receiving of data over the created queue.

3.4. Operating System Awareness in Multi-Cores

Commercial MCUs are increasingly realized as multi-core systems. The true parallelism in these systems opens a completely new field of possibilities. However, programming guides suggest to develop applications as independent as possible between the cores to reduce the possibility of interferences. The reason for this suggestion is the lack of approaches for embedded multi-core real-time systems, which this thesis is going to counteract. In this section, first a SoC bus with priority awareness for real-time systems is shown. Then, based on a priority-aware SoC bus the Remote Instruction Call (RIC) approach is presented, which aims to move globally handled functionalities to local ones.

3.4.1. Task Priority-Aware SoC Bus

Priority inversion is not limited to the software layer, it can also happen at the hardware layer such as in the SoC bus. Traditionally, SoC buses do not use priorities to arbitrate concurrently accessing components. Some SoC buses, as shown in Section 2.4.3, do support priorities but these are not in the same priority space as the software tasks. In [120] we presented a new

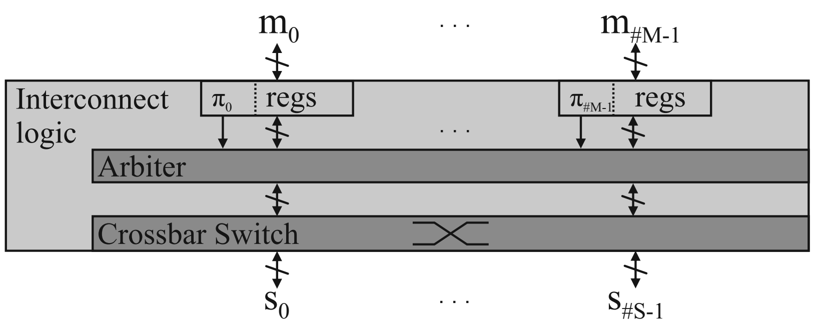


Figure 3.12.: Masters connected to the interconnect to electrically isolate them from each others.

priority-aware SoC bus, which uses the task priority to perform the arbitration without using additional wires for the priority.

To support multiple masters that are accessing a slave, on each slave access an arbitration must be performed and the SoC bus must support a multi-master protocol or all the masters must be electrically isolated from each other. The latter approach is used for the task priority-aware SoC bus, and keeps moreover the communication protocol simple. Figure 3.12 depicts the structure of the SoC bus. Every master is connected to the interconnect logic. There, the master's priority π_m (i.e., $\pi_m := p_{\tau_{m,run}}$) is used for the arbitration and the crossbar switch forwards the SoC bus signals to the addressed slave $s \in S$. The addressed slave s is selected with an $\#S$-Bit enable signal $\nu_m := s$. Thus, for a slave access, the address to which slave the access is directed is resolved in the masters and not in the interconnect logic. If the master's bus signals are not forwarded to the slave s due to another master accessing the same slave s with higher priority, the interconnect logic uses registers (i.e., Flip Flops (FFs)) for buffering the SoC bus signals. Then, the lower prioritized master's slave access will be delayed and the master will be stalled until access can be granted. To handle a delayed slave access, the interconnect logic manages for every master m a state

$$\gamma_m \in \Gamma \text{ with } \Gamma := \{\text{IDLE}, \text{DELAY}, \text{ADDRESS}, \text{BUSY}, \text{DATA}\}. \tag{3.17}$$

Figure 3.13 shows the transitions between the states. If a master's priority is lower than another master's priority accessing the same slave, the lower prioritized master moves to the state DELAY. There, the master waits (recognized by the `WAIT` signal) with the slave access and all the SoC's address phase signals are stored in registers by the interconnect logic. If no higher prioritized master any longer accesses the same slave, all the SoC's address phase signals are caught up, and then the slave access continues.

Since the connection to a slave s cannot be shared by multiple masters at the same time, the interconnect logic admits access to only one of the requesting masters A_s:

$$A_s := \{m \in M \mid ((\gamma_m = \text{IDLE}) \lor (\gamma_m = \text{DELAY})) \land \nu_m = s\} \tag{3.18}$$

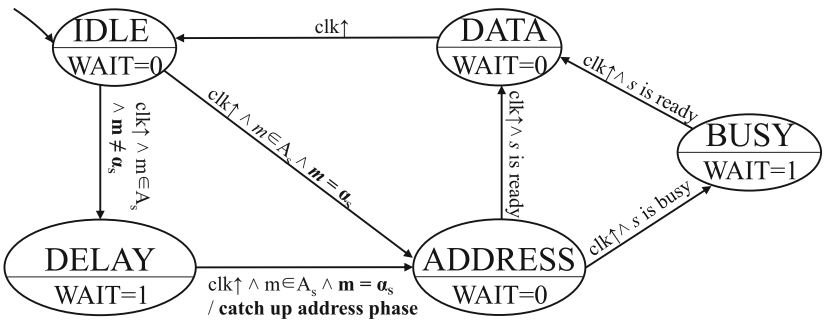
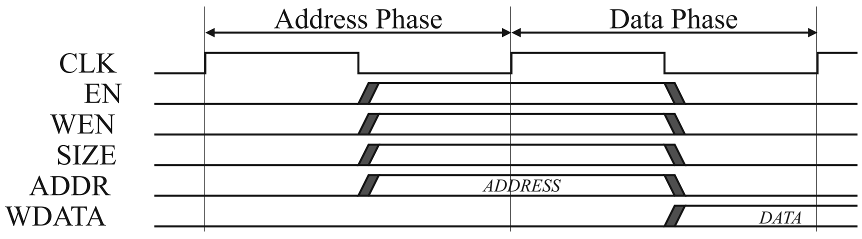
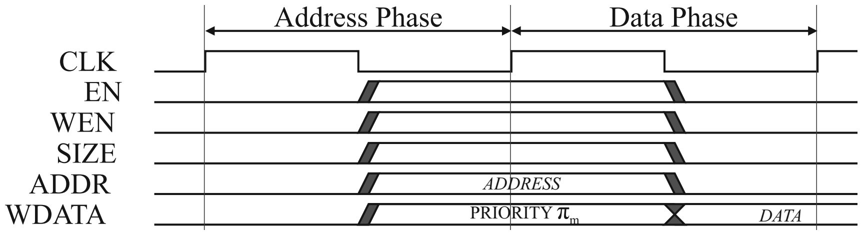


Figure 3.13.: Interconnect finite state machine.



(a) The SoC bus Write data transfer without priority injection.



(b) The SoC bus Write data transfer with priority injection.

Figure 3.14.: SoC bus write transfer demonstrating the address phase and data phase.

The set A_s contains all masters that want to access the slave s. Based on the master m's priority π_m the slave s accessing master α_s that is allowed to access the slave s is defined as:

$$\alpha_s := \exists a \in A_s \text{ with } \pi_a := \max_{\forall x \in A_s} \{\pi_x\} \wedge \min_{\{m \in A_s | \pi_m = \pi_a\}} \{master\_id(m)\} \tag{3.19}$$

To integrate the task's priority into the SoC bus protocol, the priority-aware SoC bus requires the separation of an *address phase* and a *data phase*, such as shown in Figure 3.14a. In the address phase, the control signals and the address are transmitted, and in the data phase, the data is transmitted. This division allows pipelining the slave accesses. If no pipelining of the slave access is done, the bus for the data transfer is unused in the address phase. In this case, the master's priority (equal currently running's task priority, see Section 3.2.1) is injected, as depicted in Figure 3.14b. Through the priority awareness of the *mosart*MCU and the used architecture, it can be proved (see [120]) that the master's priority π_m is always the same as the currently running task's priority $p_{\tau_{m,run}}$ of master m. Therefore, the master's priority π_m is used to unify the task priorities and the core priorities.

With the proposed task priority-aware SoC bus, priority inversions are avoided at the interconnect layer, because the tasks' priority is considered on every slave access. This allows the SoC bus to become predictable, in the sense that it is known that the highest prioritized task gets preferred and the access is immediately granted, as desired by a real-time system.

3.4.2. Remote Instruction Call

Based on the predictable interconnection bus described above, RIC [121] has been developed. The idea behind RIC is the well-known RPC concept. In RPCs, a system hands over the procedure as well as the workload to another system. The procedure is remotely executed and the system that transmitted the workload waits for the remote answer. This is useful if the remote system has more computational power to compute higher workloads than an embedded system. Moreover, an RPC executes a functionality in a completely different system with probably a different data knowledge. Thus, the workload might be executed on a remote system where all the required information are located in its local memory, and this avoids the necessary to synchronization memory as a global memory that would introduce a huge performance and management overhead. RIC's idea is based on the property that the workload should be executed where the information is located, but instead of performing a procedure in another system (often in a network), it performs an instruction in another core of a multi-core system that will not interrupt the remote currently running task (except if it is the instruction's intention, e.g., EventIRQ).

Today's mostly used RISC architectures (which are mostly load-store architectures) suit well for transmitting the arguments of an instruction over a SoC bus, because of the 3-address code format (i.e., `instruction dst, src1, src2`). The address bus is used for transmitting the first source register and the data bus for the second source register. The result of the instruction can again be transmitted over the address bus as another RIC. The first source register must be an address, which defines the OS instance (e.g., event, queue) on which the operations has to be performed. Thus, it is also not necessary to distinguish between RIC instructions or regular instructions, because the hardware recognizes, based on the first source register, if the instruction must be performed locally or remotely. Currently, the `sev` (i.e., EventIRQ) and `qwr` (i.e., EventQueue) instructions are implemented in the *mosart*MCU to support them as RICs. To support fully RIC in the *mosart*MCU, some additional extensions were made:

Figure 3.15 gives an overview of the components that are involved in supporting RICs, and next it is discussed how the RIC approach is implemented into the *mosart*MCU:

For transmitting the RIC instruction to another core, the mentioned task priority-aware SoC bus is used. There, the address bus and the data bus are used for transmitting the first source register and the second source register, respectively. To identify the remote instruction,

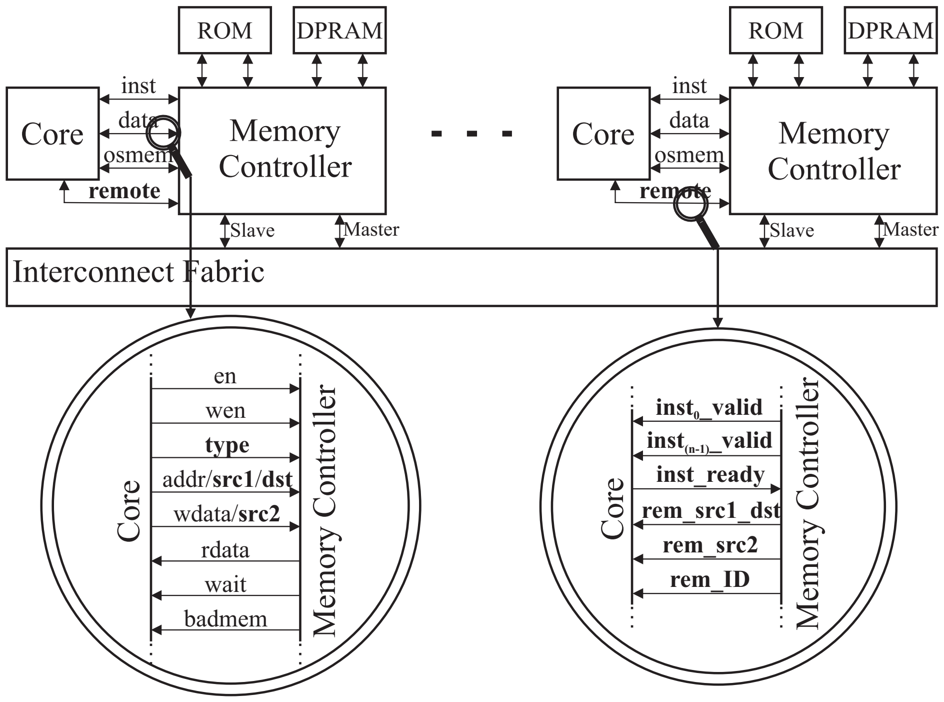


Figure 3.15.: Architecture structure for supporting the RIC approach (new connections are emphasized in bold).

the SoC bus must be adapted. The standard SoC bus has a control signal that defines if the memory access is performed on a byte, half-word, or word. This control signal is now extended to transmit the instruction type. The memory controller forwards all the SoC bus signals to the interconnect fabric to which the master is connected. According to the first source register, which represents the OS instance address, the memory controller detects to which core the RIC is addressed. Then, all the bus signals are forwarded to the destination core by the interconnect fabric.

The destination core receives the bus information at the slave interface. Here, the priority awareness of the interconnect fabric arbitrates the slave access if several cores want to perform a RIC on the same slave (i.e., a core on the slave interface). The memory controller is able to distinguish between a remote memory access and a RIC. For a remote memory access, the memory controller forwards the bus signals to the requested memory. For a RIC, the memory controller forwards the information of the RIC to the core by an additional connection, namely the connection *remote*. Then, the receiver core executes the RIC locally in hardware, concurrently to the currently running task, then the core generates the return value where required. The return value is sent back to the requester core (known by rem_ID), with another RIC. The sender core will wait until the destination core replies if the RIC expects a return value. Otherwise, the core does not wait and immediately continues with the next instruction.

Through the instruction extensions `sev` and `qwr` in the *mosart*MCU (see Section 3.3.2 and Section 3.3.3), EventIRQ and EventQueue are supported for multi-core systems, respectively. In systems without RIC support, a triggered software event or an IPC to another core must be synchronized with a multi-core synchronization primitive (e.g., test-and-set, compare-and-swap), for avoiding race conditions. These multi-core synchronization primitives result in busy-waiting or in global management that must be also synchronized somehow. With RIC the global synchronization is not required anymore: RIC passes the arguments for the instruction, which must be performed remotely, with the task priority-aware SoC that already arbitrates the remote core (i.e., slave) access. Then, RIC's workload (i.e., the instruction) is performed on the remote core only with local resources (e.g., data). Thus, RIC shifts an operation usually handled with global information to a local operation; consequently, global synchronization is not necessary anymore. Another approach without RIC would be to use IRQs to communicate between the cores; however, this may result in OS-PIs. RIC time bounds this by using the OS awareness of the *mosart*MCU. For the EventIRQ approach, the `sev` instruction is sent to the destination core where the addressed event is located. Thus, the whole EventIRQ work is performed on the remote core where the addressed event is allocated. Remark, an event is assigned to a specific core, to which only tasks on these cores can wait for it. The same applies for IPCs by using EventQueue. In EventQueue, the data that must be transmitted to the addressed queue is transferred via a RIC to the core on which the addressed queue is instantiated. Then, the EventQueue approach is executed locally, generates the return value of the instruction, and sends the return value back to the requester core as a RIC and this all simultaneously to the currently running task. Hence, with RIC, the global synchronization is not required anymore, because the instruction has to be performed only locally (including possible local synchronizations) on the core to which the RIC is addressed. Furthermore, unpredictable interruption by a lower prioritized task cannot occur because the RIC is executed concurrently in hardware to the currently running task. OS-PIs are eliminated by design; however, due to the time required to perform the remote instruction an OS-PI may occur; however, time bounded. It is the same reason as for EventIRQ and EventQueue; a high prioritized task requests to perform a RIC while a RIC is performing for a low prioritized task. However, locally the instructions are performed in deterministic time, and so the possible OS-PI is time bounded. Furthermore, through the proposed SoC bus, two RICs sent to the same core are synchronized and the RIC sent by the higher prioritized task is preferred and executed firstly. Once the first RIC finishes, the SoC bus caught up the RIC from the lower prioritized task. The RIC approach is still working if two cores are sending a RIC to the other core if both are waiting for a return value: Both core's pipelines are stalled (because each currently running task waits for the RIC's return value), while remotely the return value is computed and returned with another RIC. This works, because the passing of RIC's arguments and its return value are two individual RICs and no locks are necessary. Thus, also in this case RIC still avoids unpredictable interruptions by lower prioritized tasks and does not lead to deadlocks.

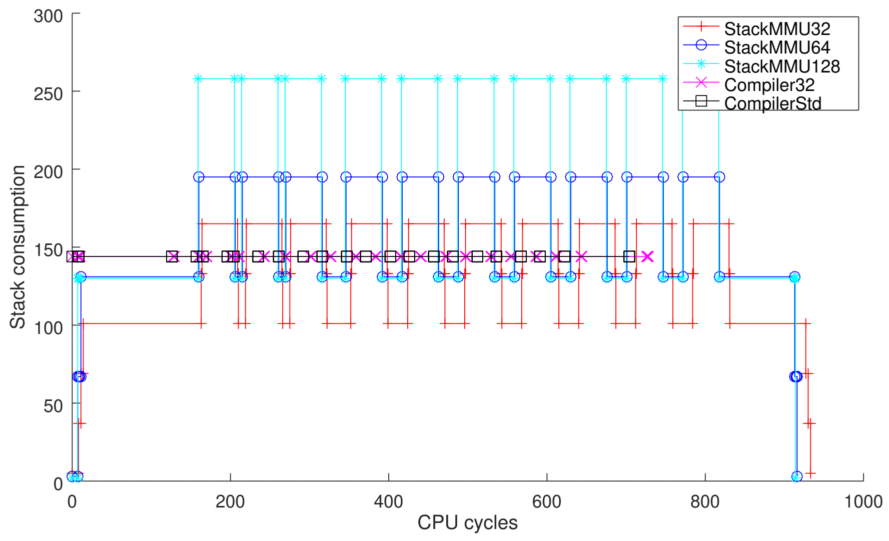
4. Evaluation Results

This chapter demonstrates the evaluation results of the proposed OS-aware extensions in the *mosart*MCU project presented in Chapter 3. Section 4.1 shows StackMMU in two examples. Section 4.2 investigates the performance of CoStack, which collaboratively releases stack memory on demand. Section 4.3 shows how the EventIRQ approach avoids the OS-PI issue in an example. The impact of the priority-aware SoC bus is evaluated in Section 4.4. Section 4.5 demonstrates the EventQueue approach, which is used in combination with RIC in a multi-core environment. Section 4.6 demonstrates the proposed OS awareness extensions in a single-core mixed critical system. Finally, Section 4.7 evaluates the resource consumption of the proposed approaches in the FPGA with different configurations. All evaluations are performed in the *mosart*MCU with our own developed *mosart*MCU-OS. The *mosart*MCU runs for all evaluations with a frequency of 50 MHz and is synthesized for the Xilinx Artix 7 FPGA[122]. The measurements were taken with an oscilloscope or were evaluated with the Xilinx Vivado simulator's output.

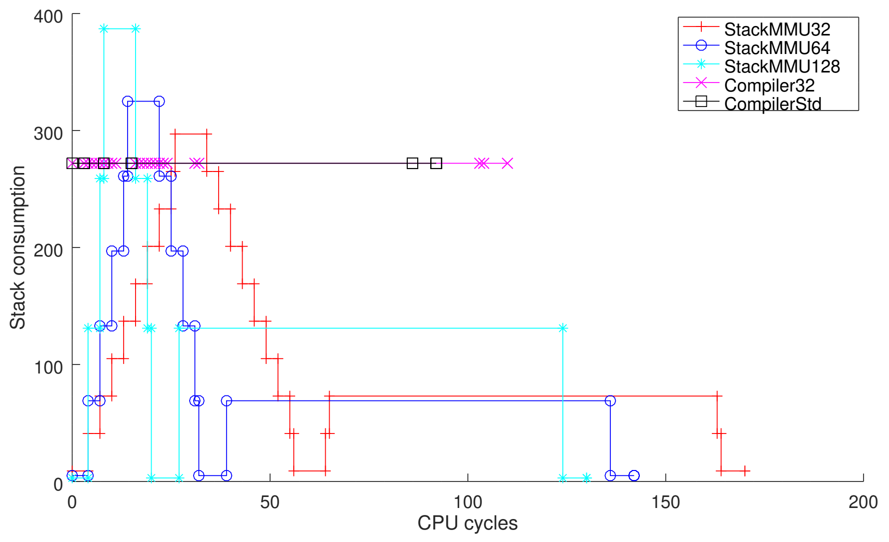
4.1. StackMMU

This section investigates the stack memory consumption of the StackMMU approach, including the execution time impact. Figure 4.1 shows two function measurements: one of a quicksort algorithm and the other with an allocation of a temporary buffer followed by a bubble sort algorithm. We used StackMMU page sizes of 32 Byte, 64 Byte, and 128 Byte for both measurements. In addition, we compared the different StackMMU page sizes with the standard individual stack memory allocation approach by using the standard compiler and the adapted compiler, which growths and shrinks the stack for maximal 32 Bytes. Remember, StackMMU limits the stack growth and shrink size in one instruction to the size of the page (i.e., `page_size`). This restriction has been eliminated with the MultiStackMMU. Furthermore, both figures contain in the stack memory consumption the additional memory required for the PP-LUT; therefore, starts and ends each StackMMU configuration with a different stack memory consumption.

Figure 4.1a shows the stack memory usage and execution time of a quicksort algorithm. The quicksort algorithm uses the *divide and conquer* technique and is recursively implemented, leading to many nested function calls. A function call affects the execution of the function prologue and epilogue, which are performing register saves and restores on the stack. Thus,



(a) Run-time stack consumption of a quicksort algorithm.



(b) Allocation of a temporary buffer with 256 Bytes followed by a bubblesort algorithm.

Figure 4.1.: Stack consumption and required execution times for the StackMMU approach with different page sizes (32 Bytes, 64 Bytes, 128 Bytes), a static individual stack allocation with the adapted compiler (i.e., compiler32) for page sizes of 32 Bytes, and the standard compiler (i.e., compilerStd).

the stack memory grows and shrinks on every function call and exit, respectively. The example shows that the selection of the stack page size influences not only the whole stack memory consumption but also the execution time, because each stack growth and shrink requires two cycles, if the changed size exceeds the page size `page_size`, otherwise only one. Here, for a stack page size of 32 Bytes (i.e., StackMMU32), on every function prologue and epilogue two stack growths and shrinkages are required. This leads to the fact that more instructions have to be performed, leading to a longer time for growing or shrinking the stack. The same behavior can be seen with the restricted compiler (Compiler32) that shows a longer execution time compared to the standard compiler (CompilerStd) due to the additional required stack growth and shrink instructions. If the stack pages are chosen to be larger, the problem of many growths or shrinkages is reduced, but the pages could have more unused memory, leading to more unused stack memory.

In the second measurement, shown in Figure 4.1b, at the beginning a buffer of 256 Bytes is temporarily used. After releasing the buffer (e.g., for StackMMU32 on cycle 60), a bubblesort algorithm is executed. Here, the example shows that with a larger stack page size, the temporary buffer is allocated and deallocated faster; however, the stack memory consumption may heavily exceed the required stack memory (compare Compiler128 and CompilerStd). After the temporary buffer deallocation, the bubblesort algorithm executes, and the stack memory consumption highly benefits from StackMMU. Comparing the StackMMU approach with the individual allocated stack memory approach, StackMMU's memory consumption is reduced by 55 % to 75 % (after cycle 100) depending on the used page size. StackMMU increases the execution time for the function by 30 % to 85 % compared to the traditional approach; however, it still behaves deterministically if enough stack memory is available and the underlying memory access is deterministic, which is the case for the *mosart*MCU.

Both measurements show that the stack page size must be chosen depending on the application and its requirements. A smaller page size would contribute to minimize unused stack memory, but it will deterministically increase the execution time for growing and shrinking for the same stack memory size. With a bigger page size, the unused stack memory may increase, but less stack growth and shrinkage operations are required and StackMMU achieves faster the new stack memory size. Furthermore, a bigger page size reduces also the size of the PP-LUT in the TCB. Therefore, a compromise must be found: For the *mosart*MCU, I would recommend to use a page size of 32 Byte or 64 Byte to optimize StackMMU for memory consumption or for speed, respectively. The *mosart*MCU is based on a RISC-V and it specifies that the stack must growth and shrink for at least 16 Bytes. Thus, a page size of 16 Bytes leads to a page growth or shrink on every function prologue. Whereas, a page size of 128 Bytes may lead to more unused memory in a page, which is rare in a memory constrained embedded system. The same tradeoff issue is for the page size in MMU environments. For instance, in general purpose Linux and Windows the default MMU page size is configured to 4096 Byte. However,

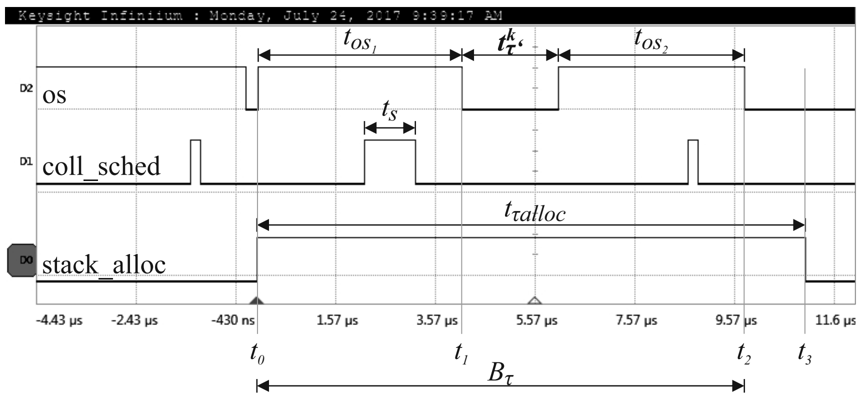


Figure 4.2.: Execution flow example of CoStack.

Table 4.1.: Measured times for the CoStack example in Figure 4.2.

t_{os_1}	t_s	t_{os_2}	$t^k_{\tau\prime}$	B_τ	$t_{\tau alloc}$
3.74 µs	1.02 µs	4.08 µs	1.94 µs	9.76 µs	11.00 µs

comparing general purpose systems to embedded systems, they usually have times bigger memory sizes; e.g., an MCU has 1 MB and a PC 8 GB of RAM. Both measurements show the potential of StackMMU to reduce the stack memory consumption. If multiple tasks are not fully utilizing their stack memory concurrently, StackMMU helps to reduce the totally allocated stack memory $\hat{S}$. However, if an out of stack memory condition would happen in the StackMMU approach, CoStack will support the OS to handle this issue.

4.2. CoStack

In case of an out of stack memory condition, CoStack searches for a lower prioritized task that voluntarily releases its collaborative stack memory. Here, we investigate the impact of CoStack in a real system. Figure 4.2 depicts time measurements of a task that offers 600 Byte collaborative stack memory and Table 4.1 lists the time measurements. The measurement *os* shows the execution time of the OS, *coll_sched* shows the time t_S required to find a collaborative task $\tau\prime$, and *stack_alloc* represents the time $t_{\tau alloc}$ for allocating the requested 600 Bytes stack memory by task τ:

- At time t_0, task τ requests 600 Byte stack memory that is currently unavailable on the system. An out of stack memory condition occurs and a collaborate exception is thrown, leading to the execution of the OS. There, the context of task τ is stored. Then the OS does management work, the scheduling of the collaborative task $\tau\prime$ (i.e., t_s), and the context restore for $\tau\prime$, which all takes in total time t_{os_1}.

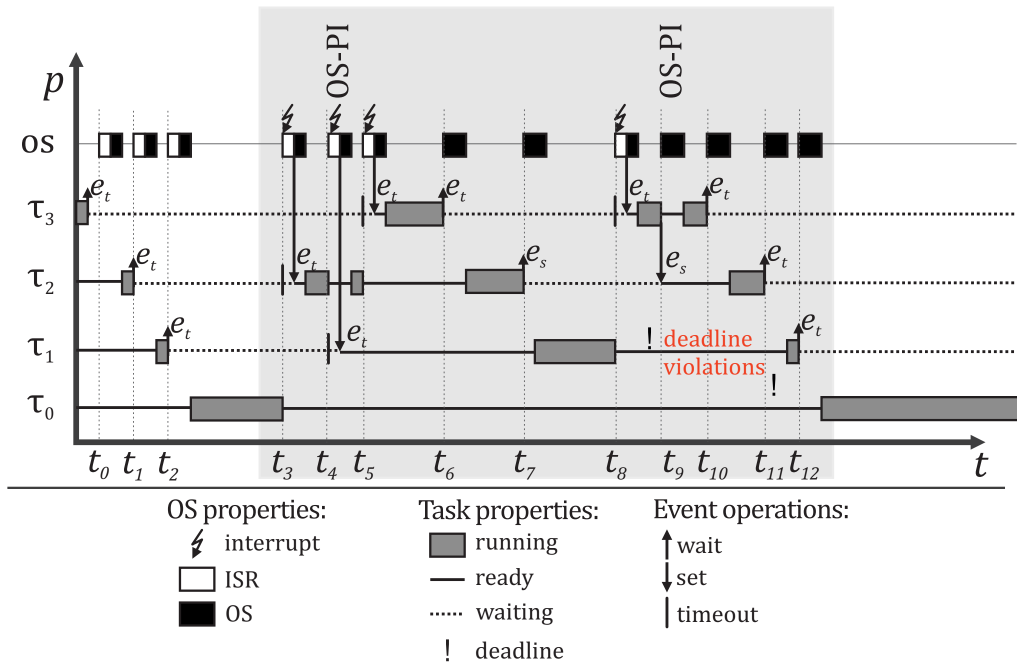
- At time t_1, the collaborative task $\tau\prime$ starts to run. However, now instead of continuing on the preempted program counter, task $\tau\prime$ continues at the collaborate label (i.e., `COLLABORATE`, see Figure 3.8a). The task restores all its callee saved registers, releases its collaborative stack memory $\kappa_{\tau\prime}$, and yields to return to the OS. Now, the OS schedules the task τ again at time t_2. The releasing of the collaborative stack takes in total $t^k_{\tau\prime}$.
- At time t_2, the OS leaves, and task τ runs. Now, enough stack memory is available and the requested stack memory is available to be finally allocated at time t_3. The time between t_2 and t_3 is required by MultiStackMMU for growing the stack memory.

CoStack does not take a constant time, because the size of the collaborative stack, the number of required collaborative tasks, and the requested stack memory influence the execution time. Nevertheless, in CoStack, all the instructions for collaborating are executed in a predictable time; therefore, they can be considered in the schedulability analysis as the blocking time B_τ (see equation 3.10 and equation 3.11 on page 40).

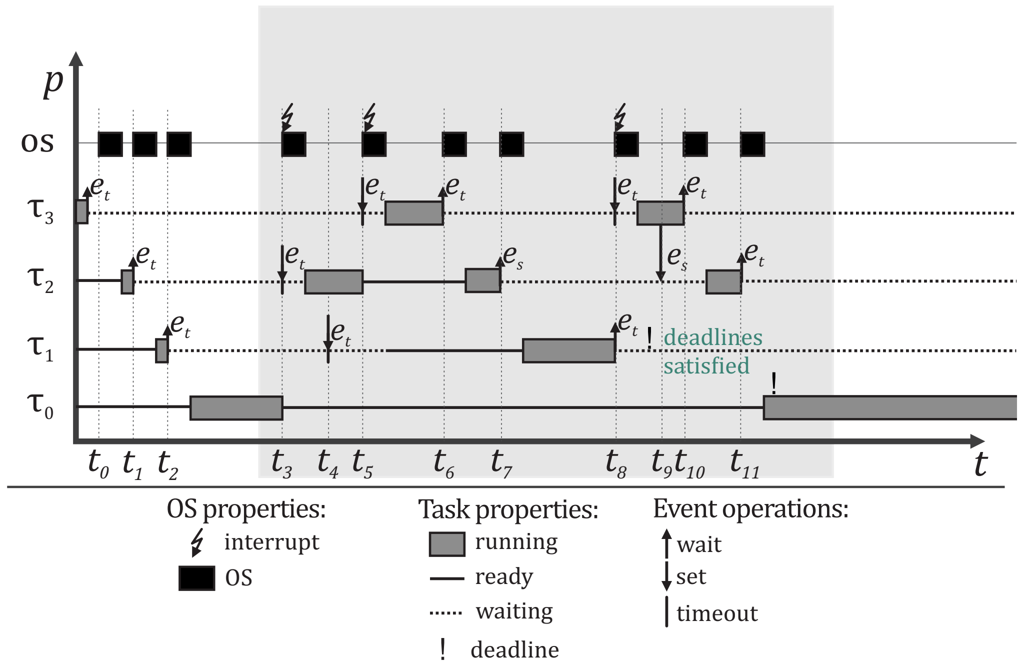
4.3. EventIRQ

This section evaluates EventIRQ with four tasks $T := \{\tau_0, \tau_1, \tau_2, \tau_3\}$, where the index defines the priority and a higher index defines a higher priority. Further, two events $E := \{e_t, e_s\}$ are used to notify the tasks of the timer event e_t and the software event e_s. This evaluation shows the task execution and event notification of the traditional interrupt handling approach (Figure 4.3a) and of the EventIRQ approach (Figure 4.3b), that avoids the OS-PI issue completely in that example (in practice, EventIRQ cannot completely avoid OS-PIs but it can at least time bound them). Both measurements were performed on a real implementation with a core frequency of 50 MHz. The following list explains Figure 4.3 in more detail:

- The tasks τ_3, τ_2, and τ_1 start to wait for the timer event e_t at the times t_0, t_1, and t_2, respectively. This causes that the OS schedules the lowest priority task τ_0.
- At time t_3, the timer event e_t is triggered:
  Traditional: The ISR is executed in the OS context. Then the OS schedules task τ_2.
  EventIRQ: The OS is triggered and schedules task τ_2.
- At time t_4, the timer event e_t is triggered. The timer event e_t notifies task τ_1, because it is the task waiting for event e_t with the closest timeout. However, it has a lower priority than the currently running task $\tau_{run} = \tau_2$:
  Traditional: Task τ_2 is preempted by the ISR leading to an OS-PI. Then, the OS returns and still schedules the last scheduled task τ_2.
  EventIRQ: No unpredictable interruption occurs, and EventIRQ adds the triggered task $\tau\prime = \tau_1$ to the pending task list ϕ. Thus, EventIRQ avoids an OS-PI at time t_4.



(a) Trace of the traditional IRQ handling.



(b) Trace of the EventIRQ approach.

Figure 4.3.: Example with four tasks and two events for the traditional IRQ handling and the EventIRQ approach.

- At time t_5, the timer event e_t triggers. Now, the timer event e_t notifies task τ_3, because it is the task waiting for event e_t with the closest timeout. This time it has a higher priority compared to the currently running task $\tau_{run} = \tau_2$. Thus, the OS schedules task τ_3.
- At time t_6, task τ_3 starts to wait for the timer event e_t. The OS schedules task τ_2, which is the highest prioritized task of all ready tasks.
- At time t_7, task τ_2 starts to wait for the software event e_s. Thus, the OS schedules task τ_1 that sets a deadline, within it must be finished with its computational work.
- At time t_8, the timer event e_t, for which task τ_3 waits, is triggered. Therefore, the OS schedules this task, due to the higher priority compared to the currently running task $\tau_{run} = \tau_1$.
  Traditional: Task τ_1 is preempted even if it has not finished its computational work yet.
  EventIRQ: Before task τ_1 is preempted by task τ_3, it has already finished with its computational work before its deadline and started to wait for a new timer event.
- At time t_9, the software event e_s is set by task τ_3 for which the lower prioritized task τ_2 is waiting.
  Traditional: The syscall `set_event` is called by task τ_3. This results into a jump to the OS that does its management work. Due to the fact, that task τ_3 executes a syscall notifying task τ_2 with a lower priority, an OS-PI is the consequence. Furthermore, task τ_1's deadline is already violated.
  EventIRQ: EventIRQ adds the triggered task $\tau\prime = \tau_2$ to the pending task list ϕ. EventIRQ checks the priorities and detects that the currently running task's priority p_{run} is higher than the triggered tasks priority $p\prime$. Thus, no jump into the OS is performed and an OS-PI is avoided.
- At time t_{10}, task τ_3 starts waiting for the timer event e_t and the OS schedules task τ_2.
- At time t_{11}, task τ_2 starts waiting for the timer event e_t. The OS selects the next task in the ready queue for processing the task on the core.
  Traditional: The OS schedules task τ_1, which deadline is already violated.
  EventIRQ: The OS schedules task τ_0. It satisfies its deadline and continues with its work.
- At time t_{12} in the traditional interrupt handling approach, task τ_0 is scheduled by the OS and it already missed its deadline because of the previous OS-PI occurrences.

EventIRQ avoids the occurrence of OS-PIs. This not only leads to avoided unpredictable interruptions of the high prioritized currently running task by lower prioritized tasks, it also supports the OS to avoid deadline violations that are unacceptable for hard real-time systems. EventIRQ, which is based on the OS awareness of the *mosart*MCU, eliminates or at least time bounds OS-PIs, leading to a more realistic schedulability analysis that now can also consider

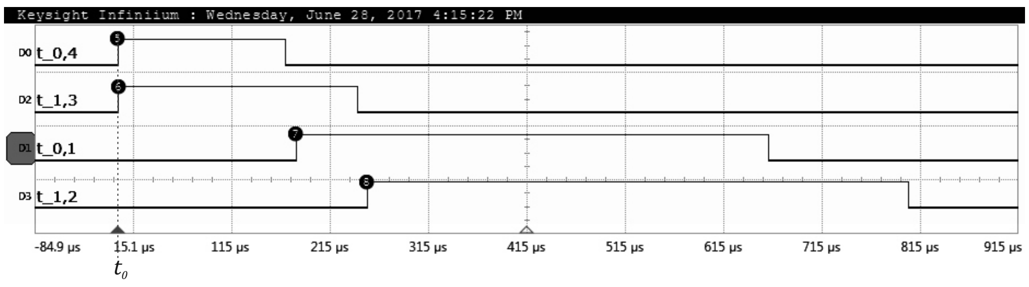


Figure 4.4.: Task executions of four tasks assigned core m_0 and m_1.

Table 4.2.: Tasks' WCETs and WCRTs comparison for static and task assigned priority to the master.

core priority	static priority ($\tilde{\pi}_{m_0} < \tilde{\pi}_{m_1}$)		task priority ($\pi_m = p_{m,run}$)	
task	WCET	WCRT	WCET	WCRT
$\tau_{m_0,4}$	155.20 µs	155.20 µs	145.30 µs	145.30 µs
$\tau_{m_1,3}$	240.32 µs	240.48 µs	242.34 µs	242.50 µs
$\tau_{m_0,1}$	440.34 µs	630.96 µs	439.68 µs	620.40 µs
$\tau_{m_1,2}$	560.32 µs	811.22 µs	560.32 µs	813.24 µs

the OS (see Section 3.3.2). However, not only the OS may produce a priority inversion but also other layers in the embedded system, such as the SoC bus.

4.4. Priority-Aware SoC Bus

With the priority-aware SoC bus, the priority inversion at the SoC bus layer is eliminated. This example shows the impact of priority inversion on the SoC bus, through static assigned priorities to the masters compared to the task priority-aware SoC bus. Figure 4.4 shows for selected tasks in a multi-core embedded system, the times where the tasks are executing on the computational units. All four tasks are accessing, while executing on the computational unit, heavily non-local memory. Thus, the memory access will be routed by the SoC-bus to the destination memory. Thereby, two of the four tasks are each assigned to different master: On master $m_0 \in M$ runs the tasks $T_{m_0} := \{\tau_{m_0,4}, \tau_{m_0,1}\}$ and on master $m_1 \in M$ the tasks $T_{m_1} := \{\tau_{m_1,3}, \tau_{m_1,2}\}$. The task identifier gives the priority, higher numbers for higher priorities; and all tasks are released at time t_0. Table 4.2 lists the WCET (i.e., time on which the task executes on the computational unit between its release and its end) and the WCRT (i.e., time between task's release and its end) of each task for statically assigned core priorities to the masters (with $\pi_{m_0} < \pi_{m_1}$) and for the priority-aware SoC bus (with $\pi_m = p_{m,run}$). For the highest prioritized task $\tau_{m_0,4}$ the WCET and WCRT increase around 6 % due to the priority inversion in the static priority approach: Task $\tau_{m_0,4}$ is higher prioritized than task $\tau_{m_1,3}$; however, the static priority prefer master m_1 and leads to a preference of task $\tau_{m_1,3}$. Thus, the execution time of task $\tau_{m_0,4}$ increases, due to the priority inversions; although, according to the priority it must be preferred.

In the task priority-aware SoC bus, the WCET and WCRT of task $\tau_{m_0,4}$ are reduced, because task $\tau_{m_0,4}$ is not stalled by the lower prioritized task $\tau_{m_1,3}$. On the other two tasks $\tau_{m_0,1}$ and $\tau_{m_1,2}$ no priority inversion occurs, since the order of the static priority is the same as for the task priorities. However, in the static priority approach, due to the previous priority inversions, the lower prioritized task $\tau_{m_0,1}$'s WCRT is increased because the higher prioritized task $\tau_{m_0,4}$ was stalled and the whole core m_0 was not able to proceed with its work.

If we recall the RM schedulability analysis, the blocking time must be considered because it reduces the schedulability of the real time system. Therefore, for the priority inversion caused by the static assigned priorities to the masters, must be considered as blocking time and therefore reduces the schedulability of the real time system. However, the proposed priority-aware SoC bus avoid priority inversion on the SoC bus by design; and therefore, it will not negatively influence the schedulability of the real time system.

Furthermore, a priority inversion would not only prolong the response time of the task of which a priority inversion occurs, but it would also prolong the response times of the following tasks. Thus, not only the task on which a priority inversion occurs may violate timing constraints, the priority inversion may affect also following scheduled task and their time constraints.

4.5. EventQueue with RIC

The RIC concept has been developed based on the proposed priority-aware SoC bus. The priority-aware SoC bus allows predicting the memory access and the execution time of a RIC performed on another core. In this section, we evaluate the performance of the EventQueue instruction by using EventQueue as a RIC.

The EventQueue implements the instruction `qwr dst, src1, src2` and stalls the pipeline as long as the return value is not received, which returns the state of the buffer (i.e., if the buffer is full or not). Figure 4.5 depicts the execution flow of the `qwr` instruction initiated by core1 and executed on core0.

The following list explains the marked times in more detail:

- At time t_0, core1 starts a queue write instruction `qwr`, addressed to the remote core0.
- At time t_1, through the memory address, the memory controller recognizes that the RIC must be performed on the core0. Thus, the memory controller forwards the memory access to the interconnect. Now, core1 is stalled until the return value is received (indicated by the signal `osaware_WB_stall` in core1).
- At time t_2, core1's memory access signals are forwarded to the core0 by the interconnect logic. The reason for this additional cycle is that the interconnect contains a register to reduce the longest path.

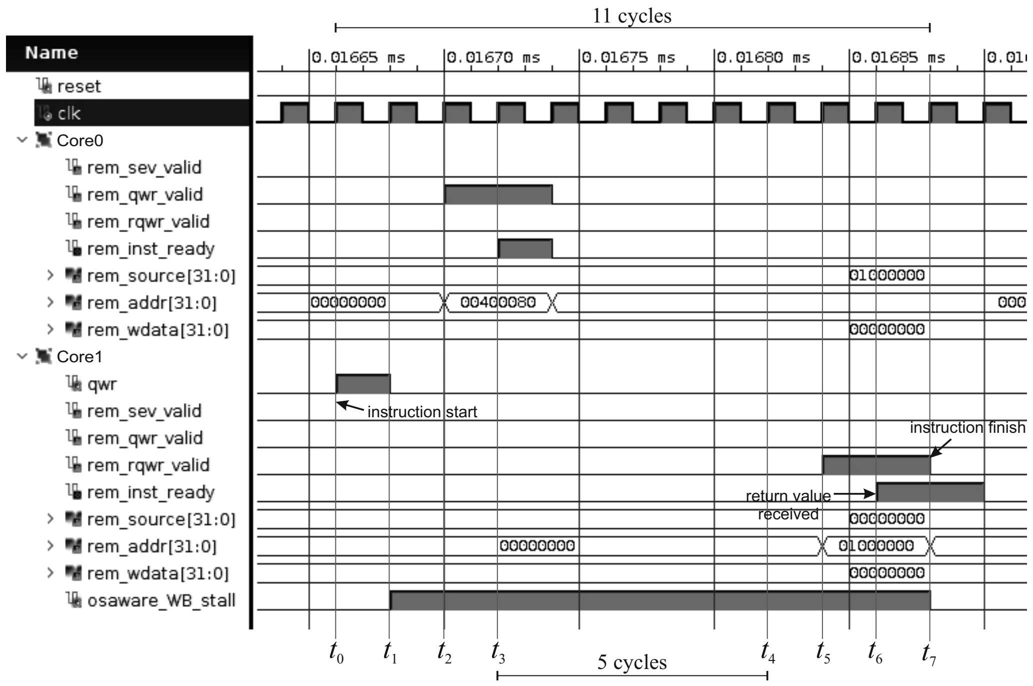


Figure 4.5.: Example of a remotely executing queue write instruction.

- At time t_3, core0 takes over the two arguments and processes the EventQueue approach (indicated by the acknowledged instruction `rem_qwr_valid` by `rem_inst_ready` in core0).
- At time t_4, the EventQueue instruction is ready to continue (after executing 5 cycles). Then, the buffer status is known by EventQueue that returns the status to the initiator core core1. Once again, due to the registers, the interconnect requires an additional cycle to forward the memory access to core1.
- At time t_5, the bus control signals are asserted on core1 and the memory controller detects a RIC, which is the return value of the `qwr` instruction.
- At time t_6, core1 receives the remaining return value (i.e., `rem_inst_ready` in core1 is set). Then, an additional pipeline stage is required to store the received value into the destination register, and at time t_7 RIC is finished and core1 is able to resume with the next instruction.

The execution of a RIC `qwr` instruction needs 11 cycles in the *mosart*MCU, where the core is stalled for 10 cycles. From these 11 cycles, 5 cycles are needed by the `qwr` instruction itself. Thus, for all RICs, a RIC adds 6 cycles to the remotely executing instruction if a return value is defined, otherwise only 3 cycles are added. Through RICs, there is no need for a global synchronization anymore, because the RIC approach, in combination with the proposed SoC, performs the RIC operation locally on the core where the data (e.g., event or queue OS instance) is resided. Further, EventIRQ and EventQueue RICs are avoiding an unpredictable interruption of the currently running task by a lower prioritized task. Thus, RIC introduces for multi-cores an event and queue mechanism that is free of OS-PIs by design, which is an intended aim for a real-time system. This is achieved by performing the RIC operations simultaneous to the currently running task, which does not interfere the currently running task on this core.

4.6. Mixed Critical System Use Case

This use case evaluates a mixed critical system (i.e., real-time and non-real-time tasks) in the *mosart*MCU with different enabled OS awareness extensions in a single core. The mixed critical system consists of the taskset $T := \{\tau_0, \tau_1, \tau_2, \tau_3, \tau_4\}$. The task priorities are ordered according to their index, where higher index means higher priority. The control system part of the mixed critical system is a safety-critical part that must keep all time constrains for an accurate control; the visualization part is the non safety-critical part and visuals user information about the controller. Figure 4.6 describes the structure of the whole system. Task τ_3 is the sensing task, which periodically senses ($D_{\tau_3} = 500\,\mu s$) the environment and transmits the processed sensor value to the controller task τ_2. The controller task τ_2 is an event triggered task, and waits until input data is transmitted over the IPC channel. Thus, if new data is sensed, the controller task starts the calculation of the actuating signal, which is sent via an IPC channel to the output task τ_4. Task τ_4 is the highest prioritized task in the system and is released each $D_{\tau_4} = 133\,\mu s$. If no new actuating signals are provided by the controller task τ_2, the output task τ_4 will output an estimated output value, which is calculated by the help of the output history. The output task τ_4 also transmits the output value to the controller task τ_2, which considers also the estimated output value on demand. The transmission of the output value is done over the same IPC channel that the sensing task τ_3 uses. Here, the control task τ_2 recognizes the different sources by a flag in the transmitted value. The user task τ_1 is not a real-time task and waits for an event that is triggered by the output task τ_4 on each performed output. The purpose of the user task τ_1 is to visualize the output trigger on a user interface (e.g., LCD or led). Finally, the system instantiates the idle task τ_0 with priority $p_{\tau_0} := 0$. This task is

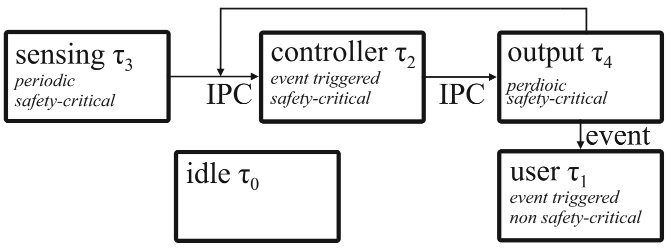


Figure 4.6.: Software architecture of the mixed critical system use case.

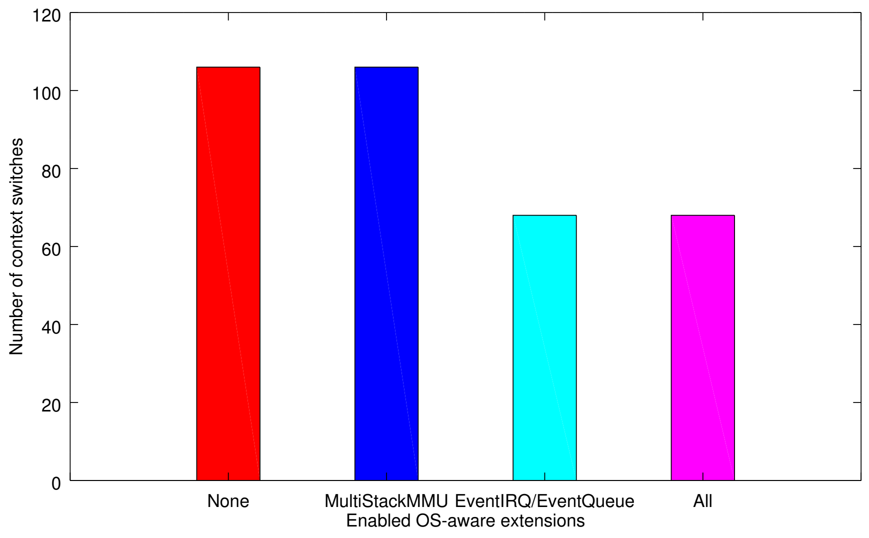


Figure 4.7.: Number of context switches for the first 10 ms.

instantiated by the OS at startup, and is scheduled by the OS if no other task has to perform operations on the computational unit.

In the *mosart*MCU, we measured the number of context switches, the stack consumption, the computational load of every task, and the computational load of the OS. All of the measurements have been executed once with no OS awareness extensions, with enabled MultiStackMMU only, with enabled EventIRQ and EventQueue only, and finally with all enabled OS awareness extensions. Remark, MultiStackMMU extends StackMMU with an automatic page allocation and deallocation one after another; thus, no specific compiler is required anymore.

Figure 4.7 depicts the number of context switches (i.e., meaning a jump from a task to the OS and back) of the mixed criticality system for the first 10 ms after system start. With EventIRQ and EventQueue enabled, the number of context switches reduces from 106 to 68, which are 36 % less. As already demonstrated in the EventIRQ evaluation (see Section 4.3), EventIRQ prevents context switches caused by IRQs or software events that would lead into an OS-PI. EventQueue is using the EventIRQ for reaching the same property; thus, it will also avoid context switches that would result in an OS-PI.

Without MultiStackMMU (i.e., None and EventIRQ/EventQueue), the OS initializes an individual stack for every task at the task initialization. Thus, the stack memory allocation is constant for these approaches. MultiStackMMU uses pages and allocates them only on demand. However, it also requires some additional memory for handling the PP-LUT in the TCBs. Nevertheless, in this use case as depicted in Figure 4.8, the currently used stack memory $U(t, page\_size)$ plus the memory needed for the PP-LUTs (i.e., enabled MultiStackMMU) is lower than the totally allocated stack memory $\tilde{S}$ of the individual stack memory approach

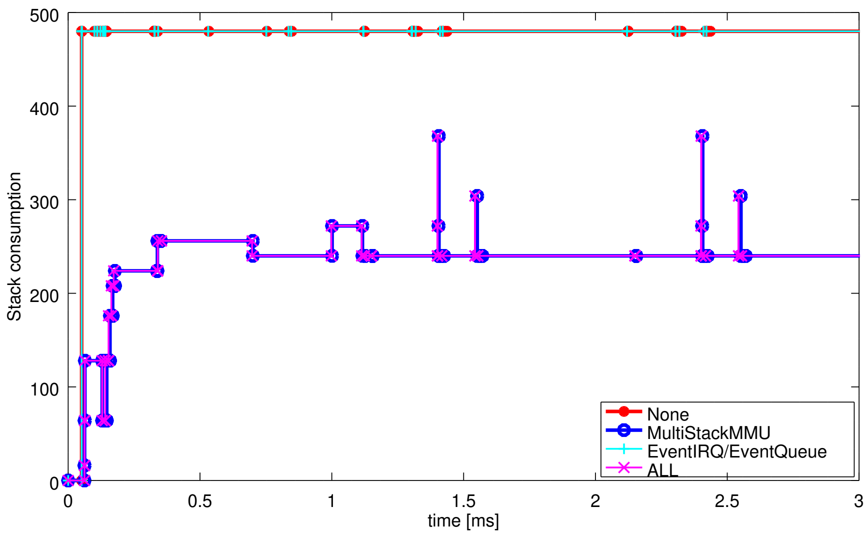


Figure 4.8.: Sum of all task's stack allocations as excerpts for the first 3 ms.

for every task. Thus, MultiStackMMU reduces for this system the required stack memory by around 22 % for the specific measured time.

MultiStackMMU has an impact on the performance, because the stack growth and shrinkage operations allocate and deallocate the pages one after another. Thus, MultiStackMMU needs more cycles to increase or decrease the stack memory compared to the traditional approach. This behavior can be seen in the load on the computational unit, depicted in Figure 4.9. In cases where the MultiStackMMU is enabled, the computational unit load of the tasks τ_1, τ_2, τ_3, and τ_4 increase. Thus, the computation unit load of the idle task τ_0 reduces. MultiStackMMU increases the computational demand, but also reduces the stack memory as shown in Figure 4.8. Therefore, this trade off must be contemplated for every system. MultiStackMMU performs predictably by design if the underlying memory access architecture executes in a predictable time, as in the *mosart*MCU. EventIRQ and EventQueue have the goal to avoid unpredictable interruptions of high prioritized task by lower prioritized tasks, i.e., to avoid or time bound OS-PIs and to avoid unnecessary context switches. The avoided context switches can be seen in the load evaluation, because with enabled EventIRQ and EventQueue, the computation unit load of the OS is reduced.

We investigated this use case scenario also with longer execution times, and all measurements showed the same behavior.

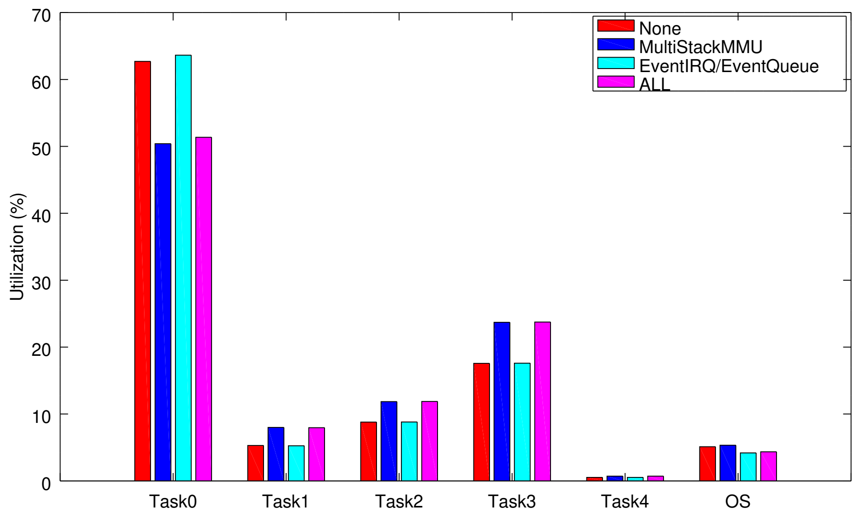


Figure 4.9.: Average load for the tasks and the OS.

4.7. Resource Consumption

The last part of the evaluation investigates the resource consumption of the *mosart*MCU in a Xilinx Artix 7A100T [122] FPGA by using the Xilinx Vivado 2017.4. For the *mosart*MCU-OS memory consumption evaluation, the RISC-V gcc compiler (build on 24th January 2018) is used for different enabled OS awareness extensions. Table 4.3 lists the resource utilization of a single core *mosart*MCU without OS awareness support, with EventIRQ and EventQueue, with StackMMU and CoStack, and with all OS awareness extensions enabled. The Lookup Tables (LUTs) and Flip Flops (FFs) utilization increase with more enabled extensions. The reason for this is that internal states must be saved and logic is required for handling all those extensions. This also affects the maximal frequency that decreases with more enabled resources. On one hand, the increasing logic results in longer paths in the FPGA. On the other hand, StackMMU is implemented in the already longest path of the computational unit. An additional pipeline stage would help to reduce the longest path and an additional register for the internal memory would improve all the maximal achievable frequencies. Through the additional LUTs and

Table 4.3.: Synthesis results of the *mosart*MCU without and with OS-aware extensions.

Extensions	None	EventIRQ & EventQueue	StackMMU & CoStack	All
LUT slices	5214	6529 (+25 %)	6749 (+29 %)	7649 (+47 %)
FF slices	3802	4433 (+17 %)	4153 (+9 %)	4717 (+24 %)
max. frequency	66.32 MHz	58.12 MHz (88 %)	41.49 MHz (63 %)	39.31 MHz (59 %)
Dynamic power	29 mW	30 mW (+3 %)	32 mW (+10 %)	33 mW (+14 %)
.text	11 416 Byte	11 712 Byte (+3 %)	11 892 Byte (+4 %)	12 152 Byte (+6 %)
.data	2816 Byte	2784 Byte (-1 %)	2816 Byte (+0 %)	2784 Byte (-1 %)

Table 4.4.: Synthesis results of the *mosart*MCU with different number of cores.

Number cores	2	4	8	16
LUT slices	17 078	34 059 (×1.99)	74 168 (×4.34)	162 664 (×9.52)
FF slices	9182	18 307 (×1.99)	36 870 (×4.02)	74 318 (×8.09)
max. frequency	29.55 MHz	27.97 MHz (95 %)	27.33 MHz (92 %)	25.11 MHz (85 %)
Dynamic power	112 mW	230 mW (×2.05)	458 mW (×4.09)	897 mW (×8.01)

FFs the dynamic power consumption is slightly increased. Nevertheless, StackMMU's power consumption only slightly increases compared to other address virtualization approaches (e.g., MMU [66], with a power consumption of around 50 % of the whole unit). Comparing the *mosart*MCU extensions with other solutions that implement the OS fully or partly in hardware, the *mosart*MCU resource utilization does not depend on the number of OS instances (e.g., tasks, events, etc.). This is achieved with the knowledge of the internal OS data structures by the *mosart*MCU and this makes the *mosart*MCU ready for future dynamic embedded computer systems.

Table 4.4 lists the resource utilization and maximal reachable frequency for a multi-core system with different number of cores and all OS-aware extensions enabled. The LUT and FF utilization increases almost linearly with the number of cores. It is not exactly linear because the interconnect's complexity is $\mathcal{O}(\#M \cdot \#S)$, which influences the whole complexity nonlinearly. The interconnect is also the main reason why the maximal reachable frequency is reduced with increasing number of cores, since it has to check more concurrently accessing masters for more slaves. The power utilization of the multi-core however increases linearly depending on the number of cores. With the usage of another interconnect approach, which longest path complexity growths linear (e.g., Network on Chip (NoC)), the *mosart*MCU may have a linear utilization (depending on the number of cores), and may lead to a higher maximal reachable system frequency.

5. Conclusion and Future Work

This chapter concludes this doctoral thesis: it gives a summary of the contributions, discusses remaining issues, and shows potential future work.

5.1. Conclusion

This doctoral thesis presented the *mosart*MCU, an OS-aware MCU for real-time embedded multi-core systems. The OS awareness is reached by giving the MCU the awareness of the OS data structures and by enabling the MCU to access (read and write) the OS data structure. With an additional connection to the data memory, the OS awareness works independently from the currently executing task, leading to a real parallel execution of the OS-aware functionalities.

This thesis started with motivating the need for new MCU concepts, then related works were presented that show fully or partially hardware implemented OSs. All these approaches restrict the number of OS instances due to the static hardware. This restriction is avoided by the *mosart*MCU presented in Chapter 3, which additionally covered the basic idea of OS awareness in hardware and showed novel approaches that use the OS awareness in the *mosart*MCU.

StackMMU is the first presented approach that realizes a shared stack memory based on pages, which are allocated to the task on demand. The page base addresses are stored on the task's TCB, which is deterministically accessed by the OS awareness in the *mosart*MCU. StackMMU is able to detect an out of stack memory condition, which the presented handling strategy CoStack is able to handle. In CoStack, a task voluntarily frees its collaborative stack memory if a higher prioritized task is blocked due to an out of stack memory condition.

The OS awareness in the *mosart*MCU is not only limited to efficient stack memory handling, it also allows avoiding or at least time bounding priority inversions caused by IRQs. With EventIRQ, all the IRQs are mapped to OS events. This hardware extension is responsible for handling a triggered IRQ for which a task is waiting. Through the priority awareness of the triggered task and the currently running task, EventIRQ interrupts the currently running task only if its priority is lower than the priority of the triggered task. Otherwise, the IRQ will be handled later by deferring it. EventIRQ works completely simultaneously to the currently running task. Based on EventIRQ, EventQueue has been developed. EventQueue transmits data to a destination task in hardware. Based on the EventIRQ approach, the currently running task is only interrupted if the triggered task (triggered by EventQueue through an event) allows it by considering the priorities. Otherwise, the handling is deferred and an unnecessary jump into the OS, which would result in an OS-PI, is prevented.

With a task priority-aware SoC bus, the *mosart*MCU is made ready for multi-cores. The underlying protocol uses the task priority for arbitrating a simultaneous access to the same slave. No additional wires are required for this purpose. Upon the task-aware SoC bus, RIC is presented. Based on the idea of an RPC, RIC executes an instruction on another core. This leads to a shifting of a usually globally handled work to a locally one; thus, RIC reduces the complexity for handling remote work.

The evaluation showed results of all the mentioned OS-aware extensions with use cases. The evaluation shows, for StackMMU, that depending on the goal the page size must be defined (i.e., speed vs more economical memory usage). The other extensions show that the performance can be increased and the extensions can avoid the occurrence of OS-PIs. All these extensions aim to make the schedulability analyses realistic, and not as pessimistic as in the past, by additionally considering the OS and IRQs. The last part of the evaluation shows how the extensions affect the resource requirements in an FPGA and how they influence the maximal reachable frequency and power consumption. A further investigation shows how the *mosart*MCU scales with all enabled extensions in a multi-core.

5.2. Remaining Issues and Future Work

The *mosart*MCU is not perfect yet. Some remaining issues are discussed here including future works on how to solve these and which additional extensions could be implemented to the *mosart*MCU.

One remaining issue is the sequential handling of OS-aware functionality in the *mosart*MCU. It could happen that an IRQ is not handled immediately due to the handling of a previous triggered IRQ, although the priority of the new IRQ is higher. Therefore, EventIRQ does not eliminate the OS-PI problem, but limits it to an upper time bound. To avoid completely OS-PIs, a future work could investigate the usage of out-of-order concepts for all the OS-aware functions in the *mosart*MCU. This may lead to a more complex control unit that may lead to a higher resource utilization. However, the out-of-order concept has the possibility to avoid completely an OS-PI caused by an IRQ. A weaker solution would be to use the idea of Intel's Hyper-Threading. There unused internal resources can be used by the next executing OS-aware functionality leading to a better resource utilization. However, also here the resource utilization increases.

The scalability of the *mosart*MCU is primarily restricted by the interconnect, that is based on a crossbar switch with non-linear complexity. NoCs are constantly researched and are evermore used in today's embedded systems. Further, the real-time behavior of NoCs that possibly may avoid priority inversions on the interconnection level is researched. Therefore, a future work could adapt the *mosart*MCU to a task priority-aware NoC.

The OS awareness does not limit the OS-aware functionalities to the proposed extensions in this thesis. According to that, another work could be to implement resource management in hardware by using the OS awareness idea of the *mosart*MCU. Therefore, a resource management protocol could be processed in hardware, simultaneously to the current execution flow. Another work could be to develop new lock-free algorithms with the idea of OS awareness in mind, for further reducing possible priority inversions for synchronization primitives with locks. Furthermore, the OS awareness could open new possibilities to implement security features into an embedded real-time system.

Another work could be, to develop and to perform a schedulability analysis on a computational unit with OS awareness, such as the *mosart*MCU. Thus, the new schedulability analysis may consider the whole OS and all IRQs; whereby, this thesis gives already the foundation for it.

Finally, a new development process can be investigated, where developing an OS and an MCU in one company, with developers coming from different fields. This starts by defining new supported extensions and goes up to an automatic adaptation of the hardware through an OS change (by using reconfigurable logic) and/or vice versa.

6. Publications

Thesis publications (ordered by publication date), and the correlation to each other is shown in Figure 6.1:

A F. Mauroner and M. Baunach. EventIRQ: An Event based and Priority aware IRQ handling for Multi-Tasking Environments. *In Proceedings of the 20th Euromicro Conference on Digital System Design (DSD)*. Vienna, Austria. August 2017.

B T. Scheipel, F. Mauroner, and M. Baunach. System-Aware Performance Monitoring Unit for RISC-V Architectures. *In Proceedings of the 20th Euromicro Conference on Digital System Design (DSD)*. Vienna, Austria. August 2017.

C F. Mauroner and M. Baunach. StackMMU: Dynamic Stack Sharing for Embedded Systems. *In Proceedings of the 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Limassol, Cyprus. September 2017.

D F. Mauroner and M. Baunach. Task Priority aware SoC-Bus for Embedded Systems. *In Proceedings of the 19th International Conference on Industrial Technology (ICIT)*. Lyon, France. February 2018.

E F. Mauroner and M. Baunach. Remote Instruction Call: An RPC approach on Instructions for Embedded Multi-Core Systems. *In Proceedings of the 19th International Conference on Industrial Technology (ICIT)*. Lyon, France. February 2018

F F. Mauroner and M. Baunach. CoStack: Collaborative Stack Sharing for Embedded Real-Time Systems. *In Proceedings of the 13th International Conference on Systems (ICONS)*. Athens, Greece. April 2018.

G F. Mauroner and M. Baunach. EventQueue: An Event based and Priority aware Interprocess Communication for Embedded Systems. *In Proceedings of the 13th International Symposium on Industrial Embedded Systems (SIES)*. Graz, Austria. June 2018.

H F. Mauroner and M. Baunach. *mosart*MCU: Multi-Core Operating-System-Aware Real-Time Microcontroller. *In Proceedings of the 7th Mediterranean Conference on Embedded Computing (MECO)*. Budva, Montenegro. June 2018.

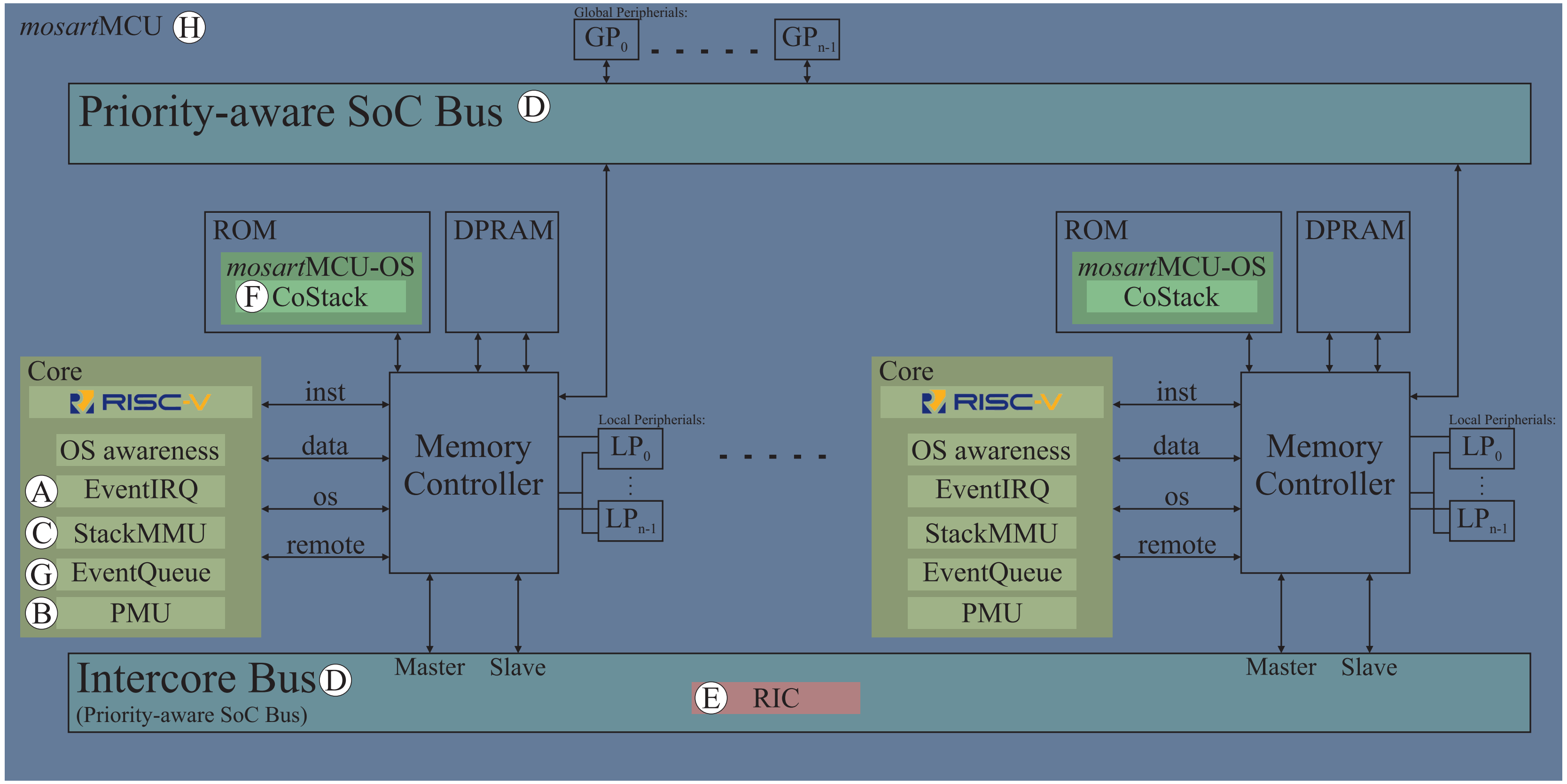


Figure 6.1.: Overview of the publications related to this thesis.

Publications not included in my thesis:

1. R. Martins Gomes, F. Mauroner, and M. Baunach. Collaborative Resource Management for Multi-Core AUTOSAR OS. *In Betriebssysteme und Echtzeit: Echtzeit 2015*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.
2. N. Sailer, F. Mauroner, and M. Baunach. meto^1 - A Versatile and Modular 32 bit low power Sensor Node Protoyping Platform for the IoT. *In Proceedings of the International Conference on Embedded Wireless Systems and Networks (EWSN)*. Uppsala, Sweden. February 2017.
3. R. Martins Gomes, M. Baunach, M. Malenko, L. Batista Ribeiro, and F. Mauroner. A Co-Designed RTOS and MCU Concept for Dynamically Composed Embedded Systems. *In Proceedings of the 29th Euromicro Conference on Real-Time Systems (ECRTS)*. Dubrovnik, Croatia. June 2017.
4. T. Scheipel, F. Mauroner, and M. Baunach. Einheit zur anwendungsbezogenen Leistungsmessung für die RISC-V-Architektur. *In Halang W., Unger H. (eds) Logistik und Echtzeit, Informatik aktuell*, Springer Berlin Heidelberg, 2017.
5. M. Baunach, R. Gomes, M. Malenko, F. Mauroner, L. Ribeiro, T. Scheipel. Smart mobility of the future – a challenge for embedded automotive systems. *In e&i - Elektrotechnik und Informationstechnik*, Springer Vienna, 2018.

A. EventIRQ: An Event based and Priority aware IRQ handling for Multi-Tasking Environments

Publication Information

F. Mauroner and M. Baunach. EventIRQ: An Event based and Priority aware IRQ handling for Multi-Tasking Environments. *In Proceedings of the 20th Euromicro Conference on Digital System Design (DSD)*. Vienna, Austria. August 2017.

Contribution

Main author

2017 Euromicro Conference on Digital System Design

EventIRQ: An Event based and Priority aware IRQ handling for Multi-Tasking Environments

Fabian Mauroner and Marcel Baunach
Institute of Technical Informatics
Graz University of Technology, Graz, Austria
Email: {mauroner, baunach}@tugraz.at

***Abstract*—Temporal predictability is a crucial requirement for hard real-time applications. Thus, deterministic software execution flows are commonly aspired to achieve that requirement. However, as an apparently unavoidable contradiction to this approach in today's embedded systems, both Interrupt Requests (IRQs) and concurrently running tasks are also required to react to dynamic environments and to allow the modular composition of complex software. These concepts operate non-deterministic and thus interleave unpredictably the program flow leading to timing violations. Even worse, determinism is initially introduced at application level, but affected by task scheduling at Operating System (OS) level, and violated by IRQs at hardware level introducing the so-called Operating System Priority Inversion (os-pi) problem: A high priority control task can easily be preempted by an IRQ that is eventually relevant for just a lower prioritized task. Thus, we propose a new hardware extension to avoid os-pi by unifying the concepts and mapping all IRQs to regular OS events. Since the extension keeps track of task priorities and event dependencies, an interrupt will only be executed if the priority of the task waiting for the triggered event is higher than the priority of the currently running task.**

***Keywords*-real-time and embedded systems; operating system awareness; interrupt handling; FPGA implementation**

I. Introduction

Since many years, there has been ongoing research in the field of real-time systems. New techniques and approaches have been invented to increase real-time capability. Most of these inventions affect the Operating System (OS), which is responsible for scheduling tasks adequately to meet their specific timing requirements. To simplify the design and implementation of real-time applications, modern OSs support a multitude of features, resulting in significant administration overhead. As a particular challenge for meeting real-time constraints in modern embedded systems, software became much more complex in general (e.g., internet of things or automotive domain). There, the OS has to support high modularity at run-time, has to react dynamically to the environment, and still has to satisfy all the real-time requirements.

The event-driven programming paradigm is a well-known approach and is supported by many embedded OSs (e.g., [1]–[3]) to interact with the environment, which is accessible through so-called Input/Output (I/O) ports. To be responsive to external stimuli, input ports commonly trigger an Interrupt Request (IRQ). If the IRQ handling is enabled, the Central Processing Unit (CPU) will immediately interrupt the current program flow and will execute the corresponding Interrupt Service Routine (ISR), which is implemented in software. Finally, the OS is responsible for scheduling a task waiting for its corresponding event, and eventually to react within a specific upper time bound, called deadline.

However, the unspecific interruption of the current program flow interleaves any other parts of the software (i.e., any task or the OS). This behavior is inherently caused by the non-overlapping execution priorities of tasks (lowest), the OS (medium), and the IRQs (highest) [4]. Therefore, programming manuals recommend to keep the ISRs as short as possible, and to forward the corresponding work to a task [5]. Nevertheless, for each IRQ and independent from the currently running task's priority, the ISR will cause two context switches: First, storing run-time information of the running task before executing the ISR; second, restoring the context of another or the same task after executing the ISR, otherwise the ISR would destroy the context of the currently running task.

To avoid such non-deterministic task switches, developers tend to use time-driven OS concepts (e.g., [6], [7]). Here, the input ports are explicitly and regularly checked ("polled") in software to detect and react on environmental changes. The responsiveness of this approach and its support for modularity is not as good as for event-driven OSs, but the software execution remains predictable; therefore, it is easier to prove it for real-time demands with a schedulability analysis [8].

To provide the convenience of event-driven programming and task modularity while improving the problem of non-determinism at the same time, we suggest moving some parts of a traditional OS into hardware. The contributions of this paper are the following:

- Priority unification of tasks, IRQs, and the OS.
- Mapping of all IRQs to OS events trough a hardware extension with an internally used linked list.
- Inclusion of timeout handling and event triggered by tasks.

The rest of the paper is organized as follows. Section II

978-1-5386-2146-2/17 $31.00 © 2017 IEEE
DOI 10.1109/DSD.2017.16

illustrates in detail the Operating System Priority Inversion (os-pi) problem, and Section III shows our approach to solve that problem. Implementation details are addressed in Section IV, and Section V shows the evaluation using our research project *mosart*MCU in combination with our *mosart*MCU-OS. Section VI compares similar approaches with our proposed solution. Lastly, Section VII concludes this work.

II. Operating System Priority Inversion

Today's state-of-the-art CPUs feature an interrupt controller that could interrupt the current program flow in case of a triggered IRQ. Due to independent priority levels, IRQs' priorities exceed the tasks' priorities. Thus, it can easily happen that a higher prioritized task is preempted because of an IRQ that is meant to trigger a lower prioritized task. This occurrence is called *rate monotonic priority inversion* [5], which is comparable to the problem of arbitrary resource sharing as presented in [9].

IRQs and shared resources are not the only cause for priority inversions. If a task calls an OS related function (i.e., syscall), most OSs leave the so-called *user mode* and execute the syscall in the *kernel mode*, above all task priorities. Thereby, the switching into another mode leads to save and restore the task context. That is required for restoring the task on the preempted program point with all its immediate values if the CPU does not supply own registers for different modes. To support event-driven programming, the OSs has to support the set event syscall. Thereby, the OS is reactivating a task that is waiting for a corresponding event. However, the setting of the event could be postponed if a task waiting for that event has a lower priority than the currently running task's priority, due to the higher priority of the currently running task. However, in state-of-the-art OSs the set event syscall is immediately executed. Thereby the OS will preempt the currently running task although the event meant to trigger a lower prioritized task.

The ISR and OS cause kinds of priority inversions; thus, we call this phenomenon the *Operating System Priority Inversion (os-pi)* problem.

Next, we illustrate the os-pi problems in an example. Assuming four tasks τ_0, τ_1, τ_2, and τ_3 arranged in increasing priorities and an OS with the priority above all tasks. Apart, there are two events e_t, and e_s for which the tasks may wait. Fig. 1 shows the example execution flow and the description below explains the marked points:

- At time t_0, t_1, and t_2 the tasks τ_3, τ_2, and τ_1 start waiting for the events e_t, e_s, and e_t, respectively. Those waiting result in syscalls and the OS schedules the next highest prioritized ready task. Hence, task τ_0 is scheduled.
- At time t_3, the timer IRQ is triggered and the associated ISR is executed. The ISR sets the event e_t to notify the waiting task τ_3. Then, the OS is running and schedules task τ_3, because of the highest priority of all ready tasks (i.e., τ_0 and τ_3).
- At time t_4, once again the timer IRQ triggers, leading to the ISR and OS execution. Now, the priority of the currently running task is higher than the priority of the waiting task for this event (i.e., $p(\tau_1) < p(\tau_3)$). Thus, an os-pi occurs.
- At time t_5, task τ_3 is calling a syscall to set the event e_s (e.g., `setEvent(`e_s`)`). The event is meant to signal the lower prioritized task τ_2. The syscall causes a switch into the OS and leads to another os-pi.

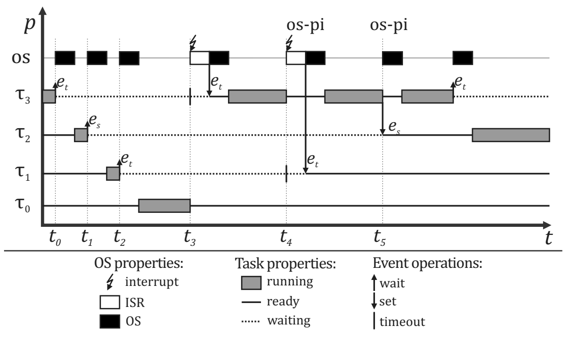


Figure 1. Illustration of the os-pi problem with four tasks and two events.

To counteract the above mentioned os-pi problems we propose to introduce task awareness into hardware. Before we present our hardware extension, we first define the underlying system and assumptions:

We are assuming a single core system with an OS. The OS is only executing in the privileged kernel mode, processes there the OS functionalities (i.e., syscalls), and maintains the scheduling of tasks. A task $\tau \in T$ is a part of an application which processes user programmed code. Each task τ possesses a static priority $p(\tau) \in \mathbb{N}$ and the OS schedules a task according to its priority and its task state. The task state can be either *running*, *ready*, or *waiting*. Running means that the task is executing on the CPU or scheduled for running, while the OS is executing. A task is ready, if it is allowed to be executed on the CPU, but its priority is lower than the currently running (scheduled) task. Waiting tasks are waiting for synchronization primitives, like an event or resource.

An OS event $e \in E$ is a synchronization primitive for that a task τ may wait. This enables the synchronization of tasks to each other, and for supporting the event-driven programming paradigm. If a task τ waits for an event e, then the task will be inserted into the corresponding event queue q_e:

$$\forall \tau_k, \tau_l \in T \wedge \tau_k \neq \tau_l \wedge p(\tau_k) \geq p(\tau_l):$$
$$q_e := (\ldots, \tau_k, \ldots, \tau_l, \ldots)$$

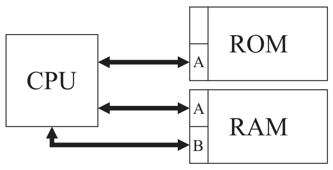


Figure 2. The system architecture to support the EventIRQ approach.

The event queue q_e is ordered according to task's priorities and the head task $h_e := q_{e,0}$ represents the highest prioritized task in that queue. The event e may be triggered by another task or IRQ, leading the OS to remove (only) the head task h_e from the event queue q_e and to insert it into the ready queue. This queue movement results in a task state change from waiting to ready, or even running if the task became the highest priority of all tasks in the ready queue.

The assumed system architecture is depicted in Fig. 2. Here, the CPU possesses a connection to the instruction memory (i.e., ROM) and twice times a connection to the data memory (i.e., RAM). The data memory possesses a dual-port memory connection, enabling a concurrent access to the data memory. The CPU controls both data channels independently to each other. Channel A is used for the common data memory access and channel B for OS-awareness including our proposed EventIRQ.

III. Event based Priority aware IRQ handling

A. Overview

The goal of our approach is to avoid the os-pi problem. This occurs, if the running task is interrupted by an IRQ or syscall triggering an event that is addressed to a lower prioritized task. Thus, the basic idea is to map the IRQs to OS events. Furthermore, the event notification has to be postponed until the os-pi problem no longer exists. Thereby, the CPU is extended by some OS-awareness and is aware of task priorities and triggered events. It is also aware of interpreting and modifying internal OS data structures. For each event $e \in E$ an event queue q_e is used. This event queue contains a double linked list of tasks, which are ordered by their priority that are waiting for a triggered event e. In order to realize the linked list, each task $\tau \in T$ contains a reference to the previous $prev_\tau \in T$ and to the next task $next_\tau \in T$ in the Task Control Block (TCB). If the event e is triggered, the head task h_e in the event queue q_e is removed and handled as proposed in the next sections.

B. Basic Concept

We are now looking into the basic EventIRQ concept, which only handles IRQs triggered by the hardware (e.g., USART, GPIO, etc.), except the OS timer IRQ for maintaining time awareness. The hardware extension keeps track of the currently running task's priority $p_{run} := p(\tau_{run})$ where task $\tau_{run} \in T$ is the currently running task on the CPU.

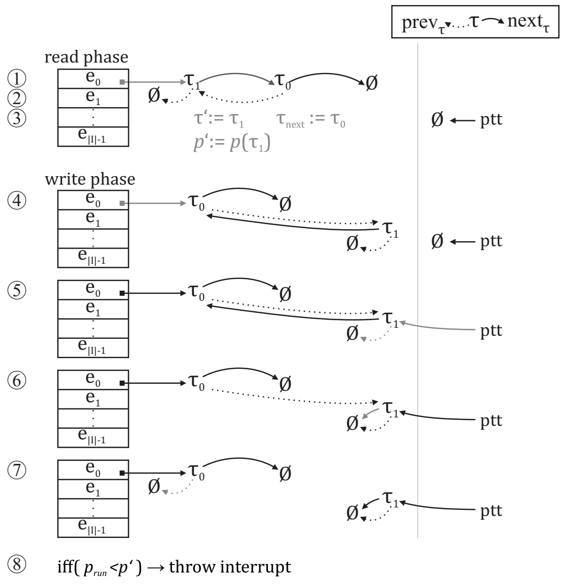


Figure 3. Example of the basic concept for a triggered IRQ $i_0 \in I$ (gray colored indicates a read or write access to the channel B data memory).

For each hardware IRQ $i \in I$, with $I := \{i_0, ..., i_{|I|-1}\}$, an event $e_i \in E$ is defined in software. All events are managed in a writable *event vector table*, similar to the interrupt vector table in common interrupt systems. The event $e_i \in E$ in the event vector table is pointing to the head task h_{e_i} of the event queue q_{e_i}. If no task is waiting for an event, the head task h_{e_i} is empty; thus, also the corresponding pointer in the event vector table is empty (i.e., $\emptyset$). If the event e_i triggers task τ at time

$$\tilde{t}_{\tau,e_i} := \{t \in \mathbb{N} \mid \tau \in T \text{ is triggered by } e_i \in E\},$$

the task τ, which is the head task h_{e_i} of the event queue q_{e_i}, is appended to the pending task list ϕ:

$$\forall \tau_k, \tau_l \in T \land \tau_k \neq \tau_l \exists e_k, e_l \in E \land \tilde{t}_{\tau_k,e_k} \leq \tilde{t}_{\tau_l,e_l} : \\ \phi := (\ldots, \tau_k, \ldots, \tau_l, \ldots).$$

This way the pending task list ϕ keeps track of the chronological order of triggered tasks. The pending task list tail pointer $ptt := \phi_{|\phi|-1}$ points to the last element in the pending task list ϕ, which is required for appending the tasks to that list.

Next, and in Fig. 3, we are presenting the steps that the hardware extension will perform if an IRQ is triggered. First, the hardware extension performs a read phase and afterwards a write phase:

① If an IRQ $i \in I$ is triggered, the hardware extension internally saves the triggered task $\tau\prime$, which is the head

104

task h_{e_i} in the event queue q_{e_i}. The EventIRQ accesses it with the event vector table.

$$\tau\prime := h_{e_i}$$

② The next waiting task for the event e_i (i.e., $q_{e_i,1}$) is read if the event queue q_{e_i} is not empty. We define the next waiting task $\tau_{next} \in T$ as:

$$\tau_{next} := \begin{cases} next_{\tau\prime} & \text{if } \tau\prime \neq \emptyset \\ \emptyset & \text{if } \tau\prime = \emptyset \end{cases}$$

③ The triggered task's priority $p\prime \in \mathbb{N}$ of the triggered task $\tau\prime$, which is located with a static offset in the TCB, is read. If no task was triggered ($\tau\prime = \emptyset$), the priority will be set to the lowest priority.

$$p\prime := \begin{cases} p(\tau\prime) & \text{if } \tau\prime \neq \emptyset \\ 0 & \text{if } \tau\prime = \emptyset \end{cases}$$

④ The event vector table of event e_i is updated with the next waiting task τ_{next} that becomes the new head of the event e_i.

⑤ Next, the triggered task $\tau\prime$ is appended to the pending task list ϕ if the triggered task $\tau\prime$ is not $\emptyset$: First, the previous pointer of the triggered task is updated with the old pending task list tail pointer:

$$prev_{\tau\prime} = ptt \text{ if } \tau\prime \neq \emptyset$$

Second, the pending task list tail pointer ptt is updated with the triggered task $\tau\prime$:

$$ptt = \tau\prime \text{ if } \tau\prime \neq \emptyset$$

⑥ The triggered task τ' is now the tail of the pending task list ϕ; therefore, the next pointer of it must be adapted:

$$next_{\tau\prime} = \emptyset \text{ if } \tau\prime \neq \emptyset$$

⑦ The event queue q_{e_i} now points to the next task τ_{next}, which is the new head of the event queue. Thus, the previous pointer of the next task τ_{next} must also be adapted:

$$prev_{\tau_{next}} = \emptyset \text{ if } \tau\prime \neq \emptyset$$

⑧ Finally, the running task τ_{run} will be preempted, if and only if (iff) the triggered task's priority $p\prime$ exceeds the currently running task's priority p_{run}:

$$\text{iff}(p_{run} < p\prime) \rightarrow \text{throw interrupt}$$

The OS has to perform some remaining steps to finalize the basic concept. On an interrupt, the OS has to handle it depending on its cause:

- If an interrupt is triggered by an IRQ, the OS removes the last appended triggered task $\tau\prime$ from the pending task list ϕ and inserts it into the ready queue. The interrupt is only triggered by the hardware extension if the last appended task to the pending task list ϕ has a higher priority than the currently running task τ_{run}. Therefore, the OS will also schedule the last triggered task $\tau\prime$ before the OS returns. To avoid race conditions on the pending task list ϕ (i.e., a hardware IRQ triggers while the OS is running), the hardware extension is only active if the CPU runs in the user mode.
- On a syscall, the execution mode is moved into the kernel mode running the OS. The syscall functionality probably changes the running task τ_{run}; and thus, the currently running priority p_{run}. Thereby, it is possible that some postponed tasks from the pending task list ϕ must be caught up. Therefore, the OS removes each task from the pending task list ϕ and insert them into the ready queue. The OS now schedules the tasks from the ready queue, according to its scheduling policy.

C. Extended Concept

Here, we show the extensions required to handle the timer for the OS and to avoid the os-pi problem by setting an event for a lower prioritized task.

1) Timeout: In an OS API without temporal awareness for syscalls, the basic concept would be sufficient. However, if a task is allowed to wait for an event up to a specific time t (e.g., `wait(`e,t`)`) or for a delay d (e.g., `sleep(`d`)`), the basic concept is not sufficient anymore. Thereby, if a task τ is waiting, because of the reasons mentioned above, the OS will save the reactivation time $\hat{t}_\tau \in \mathbb{N}$ in the TCB.

In our approach, we add time awareness to the OS by using an integrated hardware timer, which is supported by almost all of today's available CPUs.

The OS manages syscall timeouts with a double linked timeout queue q_{e_t}:

$$\forall \tau_k, \tau_l \in T \wedge \tau_k \neq \tau_l \wedge \hat{t}_{\tau_k} \leq \hat{t}_{\tau_l} : \\ q_{e_t} := (\ldots, \tau_k, \ldots, \tau_l, \ldots)$$

Thereby, the tasks are not sorted by priorities as in the event queues from the basic concept, but according to increasing timeouts.

The timer event e_t is located at the timer IRQ position in the event vector table. Now, tasks could be at the same time in the timeout queue q_{e_t} and in an event queue q_e, waiting for the timeout event e_t and a regular event $e \in E \setminus \{e_t\}$, respectively. To become member of the timeout event queue q_{e_t}, the TCBs possess two additional pointers referencing the previous timeout task $tprev_\tau \in T$ and the next timeout task $tnext_\tau \in T$.

The next part shows the extension to support timer events:

- In the read phase the reactivation time $\hat{t}$ of the next waiting task τ_{next} is read. This time sets the next compare time for triggering the timer IRQ, in order to signal the next timeout. Due to the ordered timeout queue, the next waiting task τ_{next}'s reactivation time $\hat{t}_{\tau_{next}}$ is the closest timeout of all tasks in the timeout queue q_{e_t}.

- In a configuration bit field `icfg` each IRQ is configured to indicate either a timeout event e_t or a regular event $e \neq e_t$. For the timeout event, instead of reading and updating $prev_\tau$ and $next_\tau$, the hardware extension is reading and updating $tprev_\tau$ and $tnext_\tau$, respectively.

To allow a task to wait simultaneously for a regular event and for a timeout event, the OS has to perform some additional work to maintain the correct functional behavior. In the basic concept, the OS catches up all tasks in the pending task list ϕ by removing them and inserting them into the ready queue. However, if a task τ is located twice in the pending task list ϕ (i.e., triggered regular event and timeout event), the OS would insert the task τ twice times into the ready queue. Further, it may overwrite some task information (e.g., return values). Therefore, the OS is traversing the pending task list ϕ from the head to the tail and catches up all triggered tasks. The OS checks if the task is also part of another queue/list. If so, the OS removes the task also from the second queue/list, which could be either an event queue, timeout queue, or the pending task list ϕ. For the latter, in case that the timeout event e_t was triggered before the regular event $e \in E \setminus \{e_t\}$, the head task h_e would consume this event instead. For that, a chronological order of the pending task list ϕ is required.

2) Set Event functionality: The basic and the timeout extension showed an approach to avoid the os-pi problem caused by IRQs. To avoid the os-pi problem for the set event syscall an internal process for handling software events has to be triggered. The software event $e_s \in E \setminus \{e_i \mid \forall i \in I\}$ is still pointing to the head task h_{e_s} of the event queue q_{e_s}. To set a software event e_s, a new hardware instruction or a memory mapped register uses the address of the software event e_s and triggers the process of the proposed basic concept. Instead of updating the event vector table, the passed address has to be updated.

D. Properties

In this section, we are analyzing some properties of our EventIRQ, with the assumption that an IRQ is triggered while the CPU runs in the user mode.

Lemma 1: A task is never interrupted by an event addressed to a lower prioritized task.

Proof: Our hardware extension adds priority awareness and it knows the priorities of the currently running task and the tasks waiting for a triggered event. By comparing both priorities, the hardware extension interrupts the currently running task iff the priority of the waiting task is higher. □

Theorem 2: The number of context switches in the EventIRQ approach are less than or equal to the traditional IRQ handling approach.

Proof: Follows from Lemma 1, a task will be interrupted only if the priority allows it. Thus, if an event is addressed to a lower prioritized task, the interruption, including a context switch, is avoided. If the priorities of the waiting tasks are always higher as the currently running task, each context switch will be executed. Therefore, the maximum number of context switches is bounded to the number of context switches in the traditional IRQ handling approach. □

Lemma 3: The response time of a single triggered IRQ is performed in a constant latency.

Proof: On an IRQ, our hardware extension is immediately executing the event handling in hardware, depicted in Fig. 3. It is a state machine and always executes the same sequences; thus, in a constant time. □

Theorem 4: On interleaving IRQs, the handling of the highest prioritized IRQ is started within an upper time bound.

Proof: Following Lemma 3, the IRQ is handled in a constant time. If IRQs are triggered on the same time, the higher prioritized IRQ will be handled first; thus, the handling time is the same as in Lemma 3. If the higher prioritized IRQ is triggered one cycle after the triggered lower prioritized IRQ, the handling of the lower prioritized IRQ is first finished in a constant time; and then, the higher prioritized IRQ is going to be handled immediately afterwards. □

IV. Implementation

We implemented the extended concept for our research platform *mosart*MCU running the *mosart*MCU-OS. Next, we show the relevant implementation issues.

A. mosartMCU

The *mosart*MCU[1] project aims to implement OS awareness into embedded, mainly real-time, and multi-core systems. The MCU is based on the open RISC-V [10] architecture, maintained by the University of California, Berkeley. They are offering an open source CPU implementation in Verilog, named vscale[2], which is the base for our *mosart*MCU. It implements the 32 bit integer and multiplication/division instructions [11]. The memory bus is 32 bit wide and the memory access is naturally aligned to the used data type. The implementation provides a register file with 32 registers, whereas the compiler does not touch the register `tp`. This register is thought for indicating the currently running task (they are using the term thread) by the OS. The specifier defines three different operating modes, whereas the *mosart*MCU supports only the non-privileged *user* mode and the privileged *kernel* mode. These operating modes define permissions for some instructions and for the control status registers (CSRs). These registers are used to get information from the CPU and to configure it.

[1]Multi-Core Operating-System-aware Real-Time MCU
[2]https://github.com/ucb-bar/vscale

B. The hardware extension

The hardware extension is able to operate all proposed approaches in parallel, avoiding the os-pi problem and to increase the overall performance. Thus, the extension is using an additional connection to the data memory, beside the datapath's data memory access through a dual-port memory.

To perform the proposed approach, the priority of the running task τ_{run} has to be known. To make this possible, on each change of the register `tp` the hardware extension automatically reads the priority of the task to which the register `tp` points (if it is not null), by knowing the address offset of the priority in the TCB.

On each IRQ, the interrupt controller sets the IRQ pending flag for the respective IRQ and passes the flag, along with the IRQ number, to our EventIRQ. With that information, the EventIRQ executes the proposed extended concept and the processed IRQ is acknowledged by clearing the pending flag. The IRQ is only handled by the hardware extension if the user operating mode is active. This prevents the handling of IRQs while the OS is executing. Thus, race conditions are prevented, because all the OS functionalities are only running in the kernel mode. Even in modern embedded systems, interrupt handling is disabled while the kernel is executing, because this would simplify the operating system and may improve the kernel because there is no need for synchronization among ISRs and the OS. Thus, if an IRQ is triggered while the kernel mode is active, the IRQ will not be handled until the kernel mode is finished.

The pending task list tail pointer ptt is accessible via the CSRs. An additional pointer, the pending task list head pointer $pth := \phi_0$ is added, pointing to the head of the pending task list ϕ. This pointer supports the OS to traverse easily the pending task list ϕ from the head to the tail.

All pointers in the *mosart*MCU are naturally aligned, what means that the two least significant bits (LSBs) are always zero for $32\,\mathrm{bit}$ addresses. Therefore, we can use these bits to add some additional information to the pointers. If an event is triggered, the configuration bit in the bit field `icfg` is read for the corresponding IRQ. This bit is set to the LSB of all updating pointers, while appending the triggered task $\tau\prime$ into the task pending list ϕ. This helps the OS to recognize if the task in the pending task list ϕ was triggered by the timer event or by a regular event.

To support the set event functionality, the basic Instruction Set Architecture (ISA) of the CPU was extended with an additional instruction. The new instruction is responsible for setting an event and uses one register as a parameter. The content of the register is the address of the event, which should be triggered by the hardware extension.

C. mosartMCU-OS

To support our extended IRQ handling approach, the OS must be extended to maintain the presented properties in Section III-D. The OS is only executed if a waiting task was triggered or the running task called a syscall and the priorities allow it. Firstly, the OS has to save the context of the currently running task. Secondly, if the OS was called by a higher prioritized waiting task, the OS removes only the last appended task in the pending task list ϕ. This would reduce the latency of the reactivation of a triggered high prioritized task. However, if the OS was called by a syscall, first the OS executes the syscall. Second, the OS catches up all the tasks located in the pending task list ϕ. Finally, the OS does the remaining administrative work and restores the context of the scheduled task according to the OS scheduling policy.

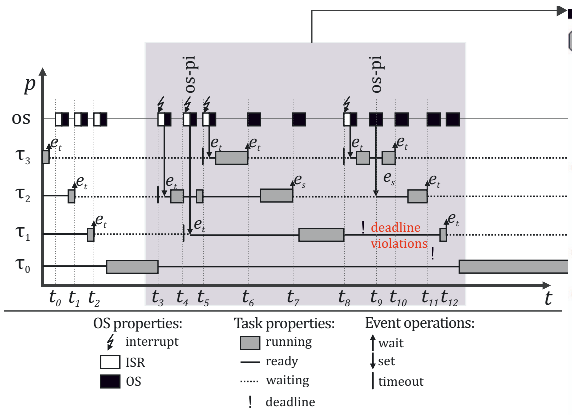
V. Evaluation

For our implementation, we investigated the behavior and the utilization of our EventIRQ. For the evaluation, we used the Nexys 4 DDR board from Digilent with a Xilinx Artix 7 Field Programmable Gate Array (FPGA).

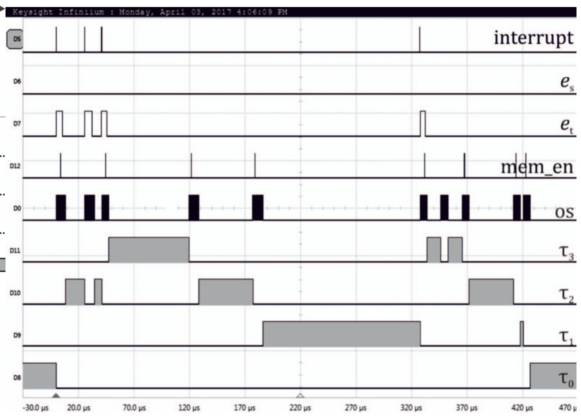
A. Behavior

The first part of the evaluation shows the behavior of a use case, shown in Fig. 4, with four tasks $T := \{\tau_0, \tau_1, \tau_2, \tau_3\}$ arranged in increasing priorities and two events $E := \{e_s, e_t\}$, running in the *mosart*MCU. Fig. 4a and Fig. 4b show the task scheduling of the traditional IRQ handling and set event approach. Fig. 4c and Fig. 4d show it for our proposed EventIRQ. Next, the market points for both approaches are explained.

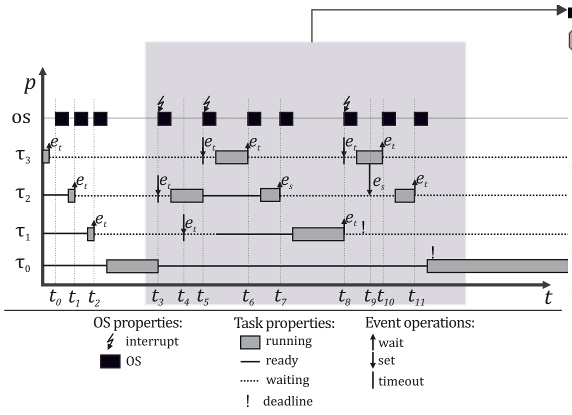
- At the times t_0, t_1, and t_2 the tasks τ_3, τ_2, and τ_1 are waiting for the timeout event e_t, respectively. Thus, task τ_0 is scheduled.
- At time t_3, the timer event e_t is triggered:
  Traditional: Therefore, the ISR and OS are executed and the OS schedules task τ_2.
  EventIRQ: Therefore, the OS is executed and schedules task τ_2.
- At time t_4, the timer event e_t is triggered, whereby this event is addressed to task τ_1 that is lower prioritized as the currently running task $\tau_{run} = \tau_2$:
  Traditional: The ISR and OS are called and preempt task τ_2, leading to an os-pi.
  EventIRQ: The EventIRQ logic adds the triggered task $\tau\prime = \tau_1$ to the pending task list ϕ and does not interrupt the currently running task. Thus, an os-pi is avoided.
- At time t_5, the timer event e_t is triggered. This is addressed for the higher prioritized task τ_3; therefore, the OS will schedule this task.
- At time t_6, task τ_3 waits for the timer event e_t; thus, the scheduler is scheduling task τ_2 because of the highest priority of all ready tasks.



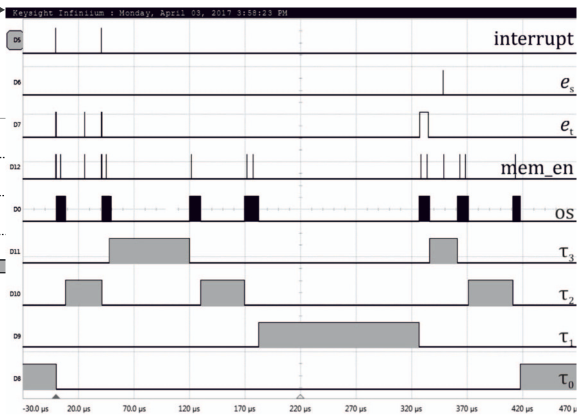
(a) Schematic execution trace of the traditional approach.



(b) Measured execution trace of the traditional approach.



(c) Schematic execution trace of the EventIRQ approach.



(d) Measured execution trace of the EventIRQ approach.

Figure 4. Use case example with four tasks and two events for the traditional and our proposed EventIRQ approach.

- At time t_7, task τ_2 starts waiting for the software event e_s. Task τ_1 is scheduled, and has to be finished within its deadline.
- At time t_8, the timer event e_t is triggered for which task τ_3 waits; thus, it will be scheduled by the OS.
  Traditional: Task τ_1 is not yet finished with its work and has to wait to be scheduled by the OS for finishing it.
  EventIRQ: Task τ_1 is finish with its work, and started simultaneously to the triggered event waiting for the timer event e_t.
- At time t_9, task τ_3 sets the software event e_s for that the lower prioritized task τ_2 is waiting for.
  Traditional: The syscall *setEvent* is called leading to a jump into the OS, which performs the work there. Further, task τ_1 is not scheduled; thus, its deadline is already violated.
  EventIRQ: The EventIRQ adds the triggered task $\tau\prime = \tau_2$ to the pending task list ϕ. The EventIRQ recognized that the triggered tasks priority $p\prime$ is less than the currently running task's priority p_{run}; thus, no jump into the OS is performed.
- At time t_{10}, task τ_3 starts to wait for the timer event e_t and the OS schedules task τ_2.
- At time t_{11}, task τ_2 starts to wait for the timer event e_t and the OS schedules the next task in the ready queue.
  Traditional: Task τ_1 is scheduled and has to finish its work, whereby its deadline is already violated.
  EventIRQ: Task τ_0 is scheduled and resumes, whereby its deadline is satisfied.
- At time t_{12}, task τ_0 is able to continue its work. However, its deadline is already violated because of the previous os-pi occurrence.

Fig. 4d and Fig. 4b depict internal FPGA control signals, which we made accessible for the oscilloscope. Thereby, we chose a unique priority for each task what allows us to detect the currently running task with the oscilloscope. The task signals are only active if the CPU is running in user mode; otherwise, the OS output signals the execution of the operating system. The signals e_s and e_t present the pending of the software event and the timeout event, respectively. The $interrupt$ signal shows the occurrence of an interrupt, which is triggered only through the correct priorities as mentioned in Section III. Finally, the signal mem_en shows the activity on the channel B data memory port. It shows an activity while tasks are running as well as the OS is running. First, the hardware extension is performing the mentioned steps on a triggered IRQ. On the latter, the head of the ready queue is changed, consequently also the register `tp`, which triggers the reading of the currently running task's priority p_{run} by the hardware.

B. Utilization

The second part of the evaluation addresses the logic utilization and maximal achievable frequency of the FPGA hardware as well as the memory requirement for the modified OS. Table I compares the original *mosart*MCU and *mosart*MCU-OS with the extended versions that includes the additional requirements for the dual-port RAM. Since, the used FPGA possess only dual-port memory, only the additional connections to the second port is required by the extended version.

Table I
RESOURCE UTILIZATION OF THE ORIGINAL AND THE EXTENDED VERSION.

Utilization	MCU			OS	
	LUTs	FFs	max. freq.	ROM	RAM
Original	2773	2019	73.180 MHz	6636 byte	20 byte
Extended	3546	2385	72.244 MHz	6680 byte	144 byte

The hardware extension increases the number of LUT slices by 773 and FF slices by 366. The LUT slices increase, because of calculations (e.g., offset addresses in the TCB) and the increase of FF slices is caused by the need of storing run-time information. For the timing, the extensions reduces the maximal achievable frequency from 73.180 MHz to 72.244 MHz witch is negligibly small. The OS ROM memory increases by 60 byte and 124 byte for the RAM. The software usage increases mainly because of the catch up code for the pending task list ϕ and the interrupt vector table is moved from the ROM section to the RAM section as the event vector table.

The above mentioned resource requirements stay constant, independent of the number of task and event instances, because the proposed extension does not statically assign tasks and events to the hardware. Only the number of interrupts would influence the size of the event vector table; and thus, the OS resource requirements. However, the number of interrupts would also influence the OS resources of the original implementation (i.e., interrupt vector table).

VI. Related Work

Renesas [12] offers an MCU with hardware support for almost all uC/OS III [3] API calls. The interface to the OS is reflected in memory mapped registers, and the API functions are called by reading and writing those registers. The scheduler, which is also implemented in hardware, executes a ready task with the highest priority. The action in case of an IRQ is configurable; thus, the hardware can choose to wake up a task, abort a waiting task, signal a semaphore, or set an event flag. Hence, a task is notified by the IRQ and scheduled by the hardware iff it has the highest priority. Otherwise, the task is inserted into a ready queue maintained by the hardware. This concept is constrained by the limited number of instances for tasks and events, which the hardware can handle due to the static memory mapped registers. Further, these instances are available at any time, and the MCU demands a lot of die space and energy even the application uses only a few of them.

Silicon OS [13], FASTCHART [14], and SEOS [15] are academic projects implementing parts of an OS into hardware. Nevertheless, these implementations are constrained to the number of OS instances; thus, limits the running of many independent developed tasks. However, in our approach the number of OS instances is independent; therefore, it supports a high flexibility number of instances.

Freescale offers an MCUs series with the XGATE technology [16]. Beside the main CPU, a co-processor for handling IRQs; thus, the handling for IRQs is outsourced, reducing the IRQ handling time in the main CPU. Scheler *et al.* [17] proposed to use a similar concept as the XGATE technology from Freescale, but they are adding priority awareness. The IRQ is still processed in the co-processor. After processing, a waiting task on the main CPU is activated by using semaphores. Thereby, the CPU is only notified for re-scheduling iff the currently running task's priority is lower than the just activated task's priority. However, compared to our approach, both mentioned solutions require an additional computation unit that has to be programmed, synchronized, and considered for real-time analysis.

In contrast, Leyva-Del-Foyo *et al.* [18] proposed to enable and disable (masking) each interrupt according to the currently running task, in a traditional interrupt controller. Thus, on each task switch, the OS adapts the interrupt controller, which only allows higher prioritized IRQs. However, the calculation of the mask consumes time on each task switch.

Gomes *et al.* [4] suggested adding task priority awareness to the interrupt controller. Thus, an interrupt will only be raised iff the running task's priority is lower than the priority of the task which is waiting for an IRQ. Similar interrupt controllers (e.g., [19]) in ARM MCUs support that functionality. However, the IRQ is still handled in an ISR and if the ISR is not handled while the next IRQ is triggered, that IRQ is missed. Consequently, for non-prioritized waiting queues (as for OS system timer), the mentioned concepts cannot avoid the os-pi problem, because for those IRQs the highest priority has to be assigned. Now with the EventIRQ also for the system timer the os-pi problem is avoided.

VII. Conclusion

The impossibility of predicting the occurrence of IRQs leads to severe problems in real-time systems. These are more amplified by the unadjusted way in which different system layers handle events and IRQs. In this paper, we showed that the traditional IRQ handling could lead to the os-pi problem. To solve that problem we presented a hardware extension with some knowledge of task priorities and event waiting queues. First, we presented the basic concept. Next, we showed the support for timeout events and for the setting of software events by avoiding the os-pi problem. Further, we showed some properties of our EventIRQ. Finally, the extended approach was implemented into our research platform showing the functionality behavior in a use case example and the evaluation of the resource requirement in the hardware (i.e., FPGA) and software.

The elimination of the os-pi problem increases the predictability of schedulability tests; thus, the next step would be to investigate the schedulability test of our proposed EventIRQ. Since, even more embedded systems are multi-cores; another work would be to extend the EventIRQ to multi-core systems.

Acknowledgment

This research project was partially funded by AVL List GmbH and the Austrian Federal Ministry of Sciences, Research and Economy (bmwfw).

References

[1] M. Baunach, "Dynamic hinting: Real-time resource management in wireless sensor/actor networks," in *Proc. of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug 2009, pp. 31–40.

[2] Real Time Engineers Ltd., "FreeRTOS," http://www.freertos.org/.

[3] Micrium, "uC/OS-III," https://www.micrium.com/.

[4] T. Gomes, P. Garcia, F. Salgado, J. Monteiro, M. Ekpanyapong, and A. Tavares, "Task-aware interrupt controller: Priority space unification in real-time systems," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 27–30, March 2015.

[5] L. E. L. del Foyo, P. Mejia-Alvarez, and D. de Niz, "Predictable Interrupt Management for Real Time Kernels over conventional PC Hardware," in *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, April 2006, pp. 14–23.

[6] M. J. Pont, *Patterns for Time-triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2001.

[7] "AUTOSAR," https://www.autosar.org/.

[8] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, jan 1973.

[9] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, Sep 1990.

[10] RISC-V Foundation, "RISC-V," https://riscv.org/.

[11] A. Waterman, Y. Lee, R. Avizienis, D. Patterson, and K. Asanovic, *The RISC-V Instruction Set Manual*, 2016.

[12] C. Stenquist, "HW-RTOS improved RTOS performance by implementation in silicon," White Paper, May 2014.

[13] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, and M. Imai, "Hardware implementation of a real-time operating system," in *Proc. of the 12th TRON Project International Symposium, 1995*. IEEE, Nov 1995, pp. 34–42.

[14] L. Lindh, "Fastchart-a fast time deterministic cpu and hardware based real-time-kernel," in *Proc. of the EUROMICRO '91 Workshop on Real-Time Systems*. IEEE, 1991, pp. 36–40.

[15] S. E. Ong, S. C. Lee, N. B. Z. Ali, and F. A. B. Hussin, "Seos: Hardware implementation of real-time operating system for adaptability," in *Proc. of the first International Symposium on Computing and Networking*. IEEE, 2013, pp. 612–616.

[16] *MC9S12XEP100 Reference Manual Covers MC9S12XE Family*, Freescale, 2013.

[17] F. Scheler, W. Hofer, B. Oechslein, R. Pfister, W. Schröder-Preikschat, and D. Lohmann, "Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system," in *Proc. of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '09. New York, NY, USA: ACM, 2009, pp. 167–174.

[18] L. E. Leyva-del Foyo, P. Mejia-Alvarez, and D. de Niz, "Integrated task and interrupt management for real-time systems," *ACM Trans. Embedded Computing Systems*, vol. 11, no. 2, pp. 32:1–32:31, jul 2012.

[19] *ARM Generic Interrupt Controller Architecture Specification - GIC architecture version 3.0 and version 4.0*, ARM Limited, 2016.

B. System-Aware Performance Monitoring Unit for RISC-V Architectures

Publication Information

T. Scheipel, F. Mauroner, and M. Baunach. System-Aware Performance Monitoring Unit for RISC-V Architectures. *In Proceedings of the 20th Euromicro Conference on Digital System Design (DSD)*. Vienna, Austria. August 2017.

Contribution

Co author. Supervised master thesis, with my core idea of combining the OS awareness concept with a performance monitoring unit.

2017 Euromicro Conference on Digital System Design

System-Aware Performance Monitoring Unit for RISC-V Architectures

Tobias Scheipel, Fabian Mauroner, and Marcel Baunach
Institute of Technical Informatics
Graz University of Technology
Graz, Austria
tobias.scheipel@student.tugraz.at, {*mauroner, baunach*}*@tugraz.at*

***Abstract*—Due to increasing complexity of software in embedded systems, performance aspects become much more important this days. This should happen early in the development process. Often execution times and events are not easily countable or measurable due to a lack of functionality in these systems. Execution time monitoring is also relevant in terms of reacting to internal and external events dynamically.**

Especially for systems using multiple tasks with internal or external resource dependencies, this is a major discipline. Another problem is that measurements during the development process are often done by interfering the system as a whole. This method leads to biases in the measurement results, because the finalized system gets deployed without these interfering functionalities and can therefore work more efficiently than the development system.

The scope of the present work is to develop a module in a hardware description language (HDL) which is able to measure execution times and events task-aware and unaware without interfering the system. The measurements of this module must be handed to the programmer through an easy accessible interface. The main focuses of the project are the scalability, platform independency concerning processor and operating system (OS), as well as easy extendibility. Also, reusability of counters during runtime is included in this work.

***Keywords*-Embedded systems, field programmable gate array, performance monitoring unit, hardware/software codesign**

I. Introduction

Determination and monitoring of execution times and events in an embedded system is getting more important nowadays. Most modern processors include a so-called Performance Monitoring Unit (PMU) to measure the performance of a system and be able to react accordingly. Intel [1] and ARM [2] define their own functionalities in their developer's manuals. The present work introduces these possibilities into the RISC-V [3] architecture, as used by the *mosartMCU* project [4]. This Instruction Set Architecture (ISA) was defined by the University of California, Berkeley, and was implemented by several different organizations and companies (e.g., [5], [6], [7] and [8]) in different hardware description languages (HDLs) as well as discrete processors. The ISA already defines a minimal set of performance monitoring functions, but none of them are included in any implementation yet. Furthermore, the specification [9] only contains simple counters for events and has no awareness of the system itself, as they are only triggerable by events within the processor.

The scope of this paper is to implement a PMU within a RISC-V processor that is aware of the software and hardware system and can therefore also access and measure tasks, events and other structures. This means that it has to know the software it is addressed and used by as well as the hardware system it is surrounded by. The main focus hereby lays on profiling tasks within a multitask software system in multiple ways by hardware.

Furthermore, the developed unit has to be scalable as far as number of hardware counters and configuration is concerned, easily extensible when new functionality like counter types are added as well as platform independent. It works as an external observer and does not interfere the system at all. Additionally, the PMU is capable of assigning hardware counters to more than one purpose without losing measure values. Due to this fact, buffering of counter values is important as well.

The paper is organized as follows. Section II shows existing works on performance monitoring counters. Section III illustrates the present structure of the processor. The following Section IV introduces the hardware structure of the newly developed PMU whereas Section V deals with the corresponding software interface. Section VI shows the experimental setup and provides measurements and investigations. Finally, Section VII discusses the developed PMU and Section VIII concludes the paper.

II. Related Work

Current high-end processors like Intel- or ARM-based include a proprietary PMU. Mostly, these units are simple counters which increment when a certain event occurs. They are configurable through specific registers. Intel calls them Model Specific Registers (MSRs) [1] and ARM reserves a certain block in its memory mapped area [2]. Both of them have in common that they only measure events, not execution times of tasks or other software constructs and they are only usable with a certain processor [10] [11].

PMUs for embedded system processors are not so wide spread. There are a few implementations for soft-cores that measure energy consumption or the overall performance of the system. Examples are the scalable Event Monitoring Unit for a multi-processor system [12] or the PMU for a LEON3 multi-core system [13].

All these systems are mainly built to measure events [14]. There are only a few methods to monitor tasks in hardware and those systems need a lot of hardware resources. Another disadvantage is that hardware counters are reserved once for a certain purpose and cannot be reused for other measurements without stopping the current assignment.

III. Present Processor Hardware and Architecture

The processor used in this work is a V-scale[1] RISC-V soft-core implemented in Verilog. It has a three-staged pipeline and follows the RV32IM specification [3] [15]. Hence, it supports a basic 32-bit integer unit alongside a multiplier/divider unit for integer values.

The main module in the hardware description is the pipeline, where all other modules like the control unit or the register file are connected, shown in Fig. 1. After analyzing the code of the core, the position determined best for the new PMU module is within the pipeline's hardware structure. This is due to the fact that all possible information can be gathered here without interfering the system itself. It is also possible to access the system memory from within this structure.

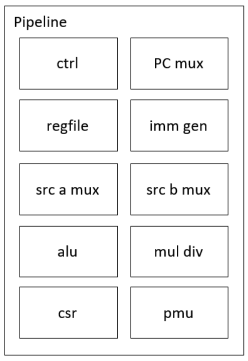


Figure 1. The pipeline of the V-scale with its modules.

To provide access to the counters of the unit, the Control and Status Registers (CSRs) are chosen. Thus, the user can read and write configuration or counter values directly with software structures provided by the RISC-V ISA. As far as platform independency is concerned, this interface can be easily exchanged for every architecture wanted. The further implementation as well as all the requirements of the unit itself will be illustrated in the following sections.

[1]https://github.com/ucb-bar/vscale

IV. PMU Hardware Structure

This hardware module contains features like several configurable counter registers, triggerable by events like change of the task pointer or the program counter, interrupts or external events. Hence, there must be different kinds of counters for global usage, task-aware global counters as well as counters belonging to a certain task. The PMU also must be configurable in terms of the number of counters itself and the number of configuration registers per counter. As far as task-aware counters are concerned, it must be possible to buffer counter values to the RAM at a task switch without delay. As the system clock of a RISC-V processor uses a 64-bit register, the same approach is taken within this work. This is due to the fact that overflows cannot be expected. A permanently incrementing 64-bit counter at 50 MHz clock reaches its maximum after approximately 12,000 years. Configuration registers however use the integer bit length of the processor itself, as the native integer length is the easiest to handle. In the present work, this bit length is 32 bit.

As stated before, the PMU is integrated directly into the pipeline module for several reasons. Due to the defined requirements, it has to have access to a certain amount of information from the system as well as a connection to the RAM. The memory link is needed to buffer values directly to the RAM and be able to reuse values later on. Furthermore, communication with the CSR file must be granted. The architecture of the PMU within the present hardware structure is shown in Fig. 2.

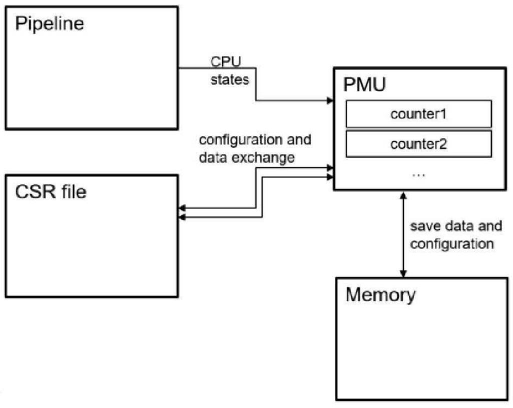


Figure 2. A rough scheme of the PMU and its connected modules in the hardware system. Connections not related with the PMU are not shown and arrows show data flow directions.

The following sections deals with a PMU of four counters with two configuration registers per counter. Regardless of this fact, the module stays scalable for whatever configuration required. At first, the configuration of the PMU is pointed out, followed by the combinatorial logic of a single

87

counter unit. Lastly, the buffering and reloading of values to and from the RAM is explained.

A. *Configuration and registers of the PMU*

The module itself consists of a main configuration register of 32-bit, in which every counter's type is defined. Those counter types are represented as 8-bit value and define how and when a counter has to measure. Additionally, every counter has its own two 32-bit configuration registers, in which further configurations to the certain type of counter can be applied. This means an overall number of nine configuration registers. Fig. 3 shows the configuration register's architecture.

A single counter value however consists of two discrete 32-bit registers, which leads to eight registers for this hardware constellation.

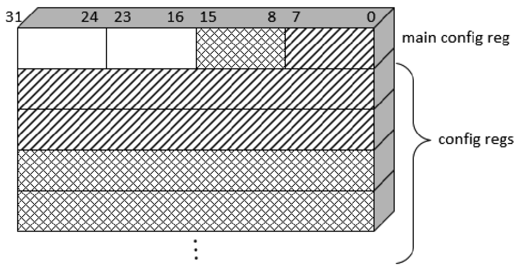


Figure 3. The main configuration register and the corresponding configuration registers of two counters.

Currently, there are three different groups of counter types:

- **Global counters**, measuring execution times and events independent of the current task. This configuration is globally valid and does not need to be buffered. An example in this case is a counter which records the time, a single task is running.
- **Task-aware counters**, measuring execution times and events for every running task. Therefore, the counter values are different from task to task. Hence, the counter values must be buffered at every task switch to guarantee valid values for each task. However, the configuration is persistent (e.g., a single counter that measures every task's individual execution time and buffers the values on task switches). Its configuration can be set once and is globally valid.
- **Counter belonging to a task**, measuring differently for every task. In this case, counter values as well as configuration values have to be buffered due to the fact that configuration is different from task to task. Here, a counter that measures if the program counter of a task is within a certain area is an example. The difference is that configuration has to be set within the task itself for every individual task and therefore must be saved and restored on each task switch.

The configuration registers for the counters itself are used to apply more configuration for the according type of counter if needed. For example, if a counter is configured to measure between two program counter values, these two addresses must be written into the counter's two configuration registers.

To access the registers mentioned above as well as the counter values, they all are mapped to CSR addresses. Hence, they are accessible via CSR operations defined in the ISA [3]. The read and write operations are passed from the CSR file to the PMU as soon as PMU-reserved addresses are recognized by the CSR file. Due to the fact that there are different address ranges defined for standard/non-standard read-only and read/write access within the CSRs [9], proper ranges must be chosen. To set configuration values, read/write addresses of the non-standard range, precisely these starting with 0x8F0, are defined. The counter values however must not be changed by the user. Furthermore, there is a certain address range predefined for hardware performance monitoring counters, starting with address 0xC03 for low-bytes and 0xC83 for high-bytes, respectively.

The module's access address ranges and logic scale with the configuration of the PMU. To ensure this functionality, the whole addressing is calculated by means of the hardware configuration's number of counters and registers and is handled by the module itself with separate address logic blocks for read and write rather than by the present CSR file.

B. *Combinatorial logic of the counters*

After proper configuration of a counter, its actual counting logic can be explained. Therefore, a single counter unit consists of a byte within the main configuration register, two dedicated configuration registers, a counter register as well as a combinatorial logic block to enable the current incrementation of the counter register. This circumstance is illustrated in Fig. 4.

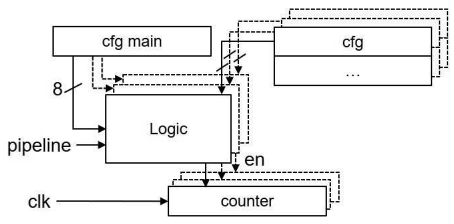


Figure 4. A simplified schematic of the combinatorial logic to enable a counter.

The logic block generates an enable flag to enable the sequential clock-triggered increment of the counter value.

88

Inputs for the logic block are the configuration registers as well as the current processor information gathered by inputs directly from the pipeline module. As this happens only by reading from certain wires, the system is not interfered by this approach.

The logic block itself contains sub-blocks for every different counter type and its own constraints. To implement an extensible structure, a logic of combinatorial blocks with a final or-gate of multiple inputs is chosen, as shown in Fig. 5.

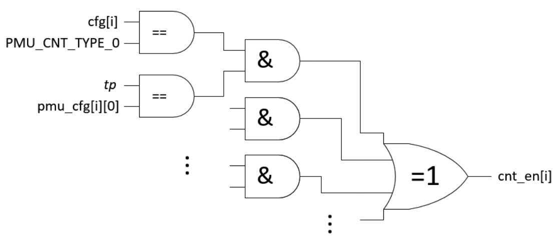


Figure 5. Generalized logic to determine a counter's enable flag.

It illustrates the part of logic for generating a counter's enable flag (*cnt_en[i]*) with *PMU_CNT_TYPE_0* as counter type in its main configuration (*cfg[0]*). Therefore, this counter measures if the current task pointer of the pipeline (*tp*) has the configured value in *pmu_cfg[i][0]*, which is the first configuration register of the current counter *i*. Further counter types must be added at the inputs of the final or-gate at the end.

In this work, different usable counter types divided by the three groups are:

- For **global** counters, there are: a counter which measures all the time (i.e. system clock), one for the processor user mode execution time, a single task execution time counter, an overall task execution time counter, a counter for all but a specific task, an overall interrupt counter, a counter for missed interrupts by a specific task and an overall external port pattern match counter.
- The group of **task-aware** counters, which save all counter values separately for every task in memory, contains: an overall task execution time counter, a task interrupt counter and an external port pattern match counter.
- There is only one counter of which counter and configuration values must be buffered. The task part counter that measures if the program counter is between two values, has another special feature. In order to start the counter at a certain program counter value and end it at another one, the PMU uses the least significant bit of the configuration registers to mark whether or not the program counter already reached the configured value. This sequential logic must be executed due to the fact that it can be possible that a task is preempted while the counter is incrementing. To carry on measuring when the task is continued, this flag must be set and reset accordingly.

C. Buffering of counter and configuration values

To ensure the explained functionality, buffering of counter and configuration values is compulsory. Hence, there has to be a memory interface within the logic and the hardware has to be aware of the operating system's (OS) task concept. In particular, the current task pointer must be known in order to integrate task awareness into the unit. On a task switch, all the register's values must be saved somewhere belonging to the suspended task in the RAM and the buffered values of the new task must be loaded into the registers, respectively.

One of the main problems concerning buffering is that memory access collisions must be avoided. As the present hardware structure uses a dual port RAM [4], one of the two memory ports is reserved for PMU purposes. As the memory allows access to a memory cell from the two channels simultaneously, consistent values are guaranteed. A memory like this is also placed within the used Artix-7 Field Programmable Gate Array (FPGA) [16].

The current task pointer of the OS must be stored in the *tp* register [3] of the processor. This allows the PMU to detect task switches and initiate the memory swap process on its own by a state machine. As long as the unit's memory access cycle is faster than the task context switch of the software system, valid values in the registers can be ensured. The data structure of a task's control block (TCB) within the OS is shown in Fig. 6.

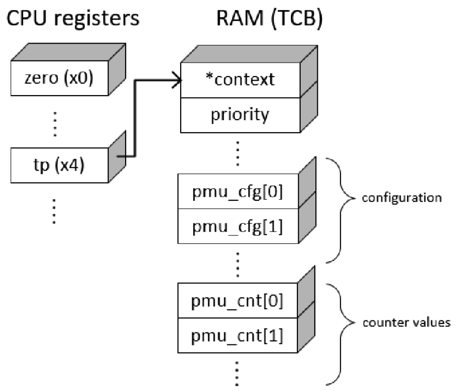


Figure 6. TCB in the software system.

With this knowledge, the module is able to calculate the memory addresses by adding certain offsets to the task pointer stored in the CPU register *tp*. To ensure scalability, these calculations take the hardware configuration into account. The state machine handles the memory swap by swapping out all the current register values into the memory cells of the to be suspended task and reading data from the new task's TCB into the PMU registers. Task-aware and task belonging measurements must be stopped during this process. In order not to lose measuring cycles during this work, every counter type must have its own compensation. It is also important to differentiate between counter types, as far as overwriting of configuration and counter values are concerned. A task-aware counter's configuration must not be overwritten during this activity, for example. However, the swap from the registers into the TCB is made, no matter which counter type is configured. This is to ensure consistent data in the memory, in order for the OS to use valid data of a suspended task from within another task to profile it.

V. Software Interface

To use the functionality mentioned above in an embedded system, a simple OS is needed. For this purpose, the *mosartMCU*-OS is used. This OS is a minimalistic system consisting of a simple task and resource management system for a RISC-V architecture. Furthermore, there are several syscalls to start the OS itself, create tasks and use the system's resources and events.

A. Hardware/Software interface

The implemented scheduler manages each task as far as starting and preempting is concerned. Therefore, the task pointer of the instantaneous running task is stored directly in the register *tp* of the CPU. By this approach, as mentioned, the simple interface between hardware and software is created. This register must be maintained by the scheduler at all time to ensure proper performance measurements. In order to signal a task switch as soon as possible for the hardware, the new value must be set into the register at the very beginning of all actions related to it. Hence, the PMU starts its swapping process at the same time the OS executes its register context switch.

Additionally, the system's data structure must be equivalent to the one used in the buffering process of the PMU to ensure consistent and valid data storage. Due to this, the OS must be well aware of the hardware configuration of the unit, as far as number of counters and configuration registers is concerned.

B. Register access

The present system knows two types of register access:

- read counter words
- read/write configuration values

As the registers are represented by certain addresses within the CSR file, they can also be accessed via simple instructions defined by the RISC-V ISA [3]. Due to the fact that the addresses of the counter values are aligned with the RISC-V definition for hardware performance counters [9], the keywords `hpmcounter`N and `hpmcounter`N`h` can be used where N is defined within the interval $[3, 31]$.

The user must be aware that the PMU starts measuring at the time instant the main configuration register is set to a valid value. To stop and reset the unit, 0x0000 must be set. Additionally, configuration of counters belonging to a task must be done within the context of the corresponding task itself to ensure proper measurements.

C. Task profiling

All the accesses to registers provide the current values in the counter registers of the instantaneous executed task. As these values get buffered into the RAM at every task switch, they are also stored within the TCB. This on the contrary provides value snapshots of every past task execution. These can be used to easily profile all tasks from within another dedicated task or the OS itself in order to gain information concerning execution times or events. Based on these results, new schedules or the schedulability can be calculated for instance [17].

VI. Measurements and Investigations

This section deals with measurements and investigations concerning the resource usage in the Artix-7 FPGA with different hardware configurations on the one hand and validations of simulations and measurements of different test cases on a certain system on the other hand. All the illustrated studies use one setup explained as follows.

A. Experimental Setup

The setup for experimental purposes consists of a Nexys-4[2] development board of Digilent based on a Artix 7[3] FPGA of Xilinx. To measure data and validate the results, a digital oscilloscope – the PicoScope 2205 MSO [18] – is used. It has two analog alongside 16 digital channels, which can all be used simultaneously. To display results, it must be connected to a PC via USB. It has a maximum input frequency of 100MHz.

For measuring purposes, the PMU is extended by a special 16-bit measurement port that is wired directly to an output port of the development board. On this port, the oscilloscope's digital inputs are connected. It can be used to display signals like enable flags of certain counters as well as values currently stored in registers. Hence, each test case can implement its own measurement port configuration to achieve exact results.

The hardware system therefore consists of a *mosartMCU* based on a V-scale RISC-V core, a RAM, a ROM, several peripherals like GPIO ports and UART as well as the designed PMU module. It operates at a clock of 50MHz.

[2]https://reference.digilentinc.com/reference/programmable-logic/nexys-4/start

[3]https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html

B. Different hardware configurations

Our reference model is a PMU with four counters and two configuration registers per counter (4x2). Taking the main configuration register into account, this leads to 17 32-bit registers overall. To synthesize a full RISC-V core with peripherals and a PMU, Xilinx Vivado 2016.2[4] is used. The first column of Table I shows the primitive statistics, net boundary statistics and clock report of the reference model.

By varying the hardware configurations by means of increasing the number of counters and/or the number of configuration registers per counter, the results shown in Table I are achieved. As values for MULT and OTHERS remain unchanged, they are not shown or discussed in the following section.

Table I
RESOURCE USAGE IN DIFFERENT HARDWARE CONFIGURATIONS.

	4x2	*4x4*	*4x8*	*8x2*	*16x2*
FLOP_LATCH	682	941	1453	1235	3141
LUT	1955	2380	2735	3994	9865
MUXFX	40	163	328	317	1095
CARRY	198	197	208	371	840
NETS	513	515	502	661	1473
CLK Inst	684	943	1455	1237	3113

When firstly looking at the change of values for increasing the number of configuration registers at a constant number of counters, Fig. 7 shows the trend. For this, the configurations 4x2, 4x4 and 4x8 are considered. The first digit in these configurations marks the number of counting registers, the second the number of configuration registers per counter (e.g., 4x2 is a PMU with four counter units and two configuration registers per counter). It shows a linear growth for all values except NETS and CARRY. This can be explained by the fact that configuration registers are not wired that complicatedly within the module and are optimized by the synthesis tool. The linear growth of the other values is due to the increase of registers itself. More Lookup tables (LUTs) and multiplexers (MUXFXs) are required to address the new added configuration registers. The number of flip flops (FLOP_LATCH) approximately follows the amount of clock instances (CLK Inst) for the reason that every latch needs its own clock input.

Secondly, the number of counters is increased by constant number of configuration registers per counter. This circumstance is shown in Fig. 8. In this case, the number of LUTs grows by a square function. A similar growth can be seen with flip flops and clock instances. This can be explained due to the fact that by increasing the number of counters, the overall number of configuration registers rises. Further, the addressing of the registers gets more complicated which can be seen at the increase of multiplexers.

[4]https://www.xilinx.com/products/design-tools/vivado.html

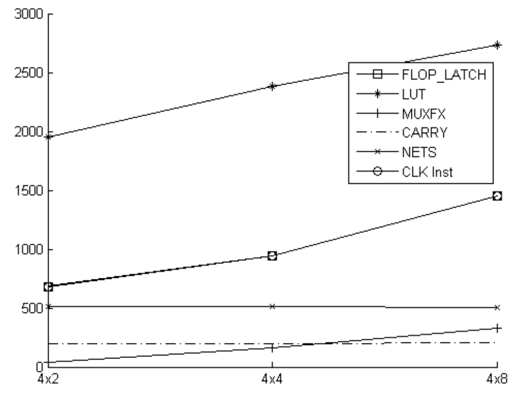


Figure 7. Growth of certain parameters when varying the number of configuration registers.

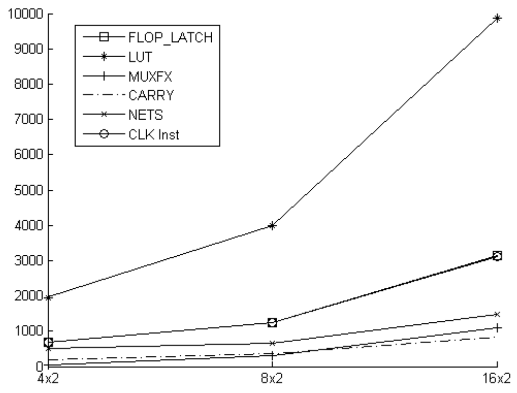


Figure 8. Growth of certain parameters when varying the number of counters.

C. Test cases

All the following test cases build on a 4x2 configured PMU and a simple *mosartMCU*-OS setup. As the system measures clock cycles in most times, the actual time can be calculated by

$$t = Cnt \cdot \frac{1}{f_{CPU}} = Cnt \cdot \frac{1}{50MHz}, \qquad (1)$$

where Cnt is the value within a counter register and f_{CPU} is the clock frequency of 50MHz. The resulting time t presents time in seconds. This means, that the expected maximum deviation ε is

$$\varepsilon = \frac{1}{50MHz} = 20ns. \qquad (2)$$

The test cases are divided into a simulation part showing the expected result followed by a measurement with the oscilloscope. All simulations are carried out with Vivado's built in simulation tool.

Table II
COUNTER SIMULATION AND MEASUREMENT RESULTS.

Counter type	*Sim result*	*Time*	*Measurement*
(1)	6736	$134.72\mu s$	$134.7\mu s$
(2)	14	$280ns$	$280.4ns$
(3)	8183	$163.66\mu s$	$163.8\mu s$
(4)	8815	$176.30\mu s$	$176.38\mu s$
(5)	709	$14.18\mu s$	$14.13\mu s$
(6)	8218	$164.36\mu s$	$164.4\mu s$
(7)	15	$300ns$	$309.7ns$
(8)	25	$500ns$	$509.7ns$
(9)	9411	$188.22\mu s$	$188.6\mu s$
(10)	946	$18.92\mu s$	$19.099\mu s$
(11)	10830	$216.6\mu s$	$216.8\mu s$
(12)	9411	$188.22\mu s$	$188.6\mu s$

1) Global counters: The first test case illustrates the usage of global counters. In fact, the first counter (1) of the setup measures all the time, whereas the second (2) measures all the user mode time of the processor, the third (3) is set to measure the overall task time and the fourth (4) the overall task time except for the idle task. Table II lists the simulation and measurement results of this test case at different time instances.

2) Simple task counters: The next test cases deal with a global counter for a single task (5), one for the overall task-aware task time (6) as well as a counter belonging to a task, which measures areas between two program counter values. Two results at different time instances are shown as (7) and (8) in the results table II. The results in (7) and (8) show the usage of one hardware counter in two different tasks. The values in the registers are only valid for the instantaneous running task.

3) Task-aware test case: At last, this test case shows a global overall task time counter, two global task time counters for single tasks, configured to two different tasks and a task-aware task time counter.

At the first time instant at which the global overall task time counter results in the value (9), the single task counter for the first task as well as the task-aware counter yield in value (10), which proofs correctness.

At the second time instant, the other task is investigated. The overall task time counter has the value (11) and the corresponding values for the single task counter for the second task and the task-aware counter again deliver the same value (12).

All these results show that all the global counters work properly within certain measurement uncertainties and result in the same values within ε as expected through simulation. Also, the final counting values in the registers match the expected values.

VII. DISCUSSION

The developed PMU has some main advantages to other present works. Compared to approaches which measure task constructs within the OS, this hardware solution is able to measure execution times without overhead or lost cycles. Software approaches slow down the whole system and can yield falsified results due to interferences. Additionally, the developed module can stay within the hardware structure rather than be removed after optimizing software structures.

The module is also able to measure a lot of different things with very few hardware counters. This is due to the fact, that a single counter can run a certain configuration for each task.

As far as other hardware solutions [12] are concerned, this module's scalability leads to minimal hardware increase, as hardware counters can be reused rather than having to add new ones to the system. It can also be used in multi core systems, as every core has its own PMU. In this case, only measurements for the certain core can be made, as the module is connected directly to it. Future aims towards an additional overall unit to include more information concerning multi core operations [13].

VIII. CONCLUSION

The present paper shows a new approach on how to measure execution times and events of an embedded system by integrating a PMU by means of hardware/software co-design approaches. It is constructed to be aware of the hardware and software system it is embedded in. Furthermore, the unit is able to reuse counter elements at runtime to minimize hardware complexity. It is also reconfigurable, scalable and easily portable to other hardware systems, as it is implemented in a HDL and has a simple and easily adjusted interface to the CPU itself.

Moreover, it could be shown that re-used task-aware counters yield the same results as multiple dedicated counters each reserved for a single task. At the same time hardware usage is minimized through this approach. Due to this fact, a more detailed performance profile can be created, as task awareness is built in. It is not only possible to consider the system as a whole, but also profile certain parts of it.

As it is non-intrusive, it can be used for worst case execution time analysis at an early development stage. Moreover, if deployed to an FPGA-based embedded system, even its software works alongside the system and therefore no altering in any execution state is expected.

REFERENCES

[1] Intel, "Intel Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide," Intel, Tech. Rep. 253669-061US, Dec. 2010.

[2] ARM, "Cortex-A5 Technical Reference Manual," ARM, Tech. Rep., 2016. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0433c/DDI0433C_cortex_a5_trm.pdf

[3] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, “The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-118, May 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html

[4] F. Mauroner and M. Baunach, “Event based and Priority aware IRQ handling for Multi-Tasking Environments,” 2017, Euromicro DSD (accepted).

[5] VectorBlox, “ORCA FPGA-Optimized RISC-V,” 2016, http://riscv.org/wp-content/uploads/2016/01/Wed1200-2016-01-05-VectorBlox-ORCA-RISC-V-DEMO.pdf.

[6] A. Traber, F. Zaruba, S. Stucki, A. Pullini, G. Haugou, E. Flamand, F. K. Grkayank, and L. Benini, “PULPino: A small single-core RISC-V SoC,” 2015, http://iis-projects.ee.ethz.ch/images/d/d0/Pulpino_poster_riscv2015.pdf.

[7] A. Traber, “RI5CY Core: Datasheet,” ETH Zrich, Tech. Rep., Feb. 2016. [Online]. Available: http://www.pulp-platform.org/wp-content/uploads/2016/02/datasheet_RI5CY.pdf

[8] Y. Lee, A. Ou, and A. Magyar, “Z-scale: Tiny 32-bit RISC-V Systems,” 2015, https://riscv.org/wp-content/uploads/2015/06/riscv-zscale-workshop-june2015.pdf.

[9] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovic, “The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9.1,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-161, Nov 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-161.html

[10] D. Patil, P. Kharat, and A. K. Gupta, “Study of Performance Counters and Profiling Tools to Monitor Performance of Application,” *21st IRF International Conference*, 2015.

[11] A. Singh, A. Buchke, and Y.-H. Lee, “A Study of Performance Monitoring Unit, perf and perf_events subsystem,” -, 2012.

[12] J. A. Ambrose, V. Cassisi, D. Murphy, T. Li, D. Jayasinghe, and S. Parameswaran, “Scalable Performance Monitoring of Application Specific Multiprocessor Systems-on-Chip,” *2013 IEEE 8th International Conference on Industrial and Information Systems*, Aug. 2013.

[13] N. Ho, P. Kaufmann, and M. Platzner, “A hardware/software infrastructure for performance monitoring on LEON3 multicore platforms,” *24th International Conference on Field Programmable Logic and Applications*, 2014.

[14] B. Sprunt, “The Basics of Performance-Monitoring Hardware,” *IEEE Micro (Volume: 22, Issue: 4, Jul/Aug 2002)*, 2002.

[15] T. Chen and D. A. Patterson, “RISC-V Geneology,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-6, Jan 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-6.html

[16] Xilinx, *7 Series FPGAs Memory Resources*, ug473 (v1.12) ed., Xilinx, Sep. 2016. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf

[17] F. Dittman and S. Frank, “Hard real-time reconfiguration port scheduling,” *Design, Automation and Test in Europe Conference and Exhibition*, Apr. 2007.

[18] *PicoScope 2205 MSO Mixed Signal Oscilloscope*, Pico Technology, 2016. [Online]. Available: https://www.picotech.com/download/datasheets/PicoScope2205MSODatasheet-en.pdf

C. StackMMU: Dynamic Stack Sharing for Embedded Systems

Publication Information

F. Mauroner and M. Baunach. StackMMU: Dynamic Stack Sharing for Embedded Systems. *In Proceedings of the 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Limassol, Cyprus. September 2017.

Contribution

Main author

StackMMU: Dynamic Stack Sharing for Embedded Systems

Fabian Mauroner and Marcel Baunach
Institute of Technical Informatics
Graz University of Technology
Graz, Austria
Email: {mauroner, baunach}@tugraz.at

***Abstract*—Real-time multi-tasking systems may require an individual stack for each task to fulfill all hard real-time requirements. However, these stacks may consume a huge memory space, even if not all stacks are simultaneously fully utilized. Thus, sharing currently unused stack space may improve memory utilization as possible with Memory Management Units (MMUs). However, an MMU introduces temporal jitter to memory accesses, influencing the real-time behavior. In this work, we propose a new concept to share dynamically the complete available stack space across tasks. Thereby, every stack operation executes in a deterministic time, by giving the Microcontroller Unit (MCU) Operating System (OS)-awareness.**

I. Introduction

In the past, embedded systems were primarily designed for one certain scenario. Thus, hardware and software were aligned to a particular purpose. Statically designed software applications were optimized to fit into an embedded system's program and data memory. However, this programming paradigm prevents or even limits the adding and replacing of software parts, called *tasks*. Even worse, the task's memory consumption cannot be dynamically adapted to the new environment requirements. The Internet of Things (IoT), Industry 4.0, or automotive software are examples were dynamically updating and adaptation of tasks at run-time are required. Therefore, the practice of writing static software and bringing them together in one linking step will no longer be applied; and thus, to optimize statically the code (including memory usage) for a particular purpose.

A way to handle dynamically memory space is the well-known concept of address virtualization [1]. The virtualization is performed by a hardware component, the Memory Management Unit (MMU). It is responsible for translating all Virtual Memory (VM) addresses into Physical Memory (PM) addresses. However, using an MMU is not so common in real-time embedded systems. There is the need for low energy consumption and predictable memory access times. These needs cannot be satisfied with a conventional MMU, resulting in the use of simple Microcontroller Units (MCUs) without an MMU.

Memory is a scarce resource in MCUs, because of the significant space required in a die. Thus, more memory would also increase the costs of the MCU; and thus, also of the product. To counteract that issue, the developers often select the smallest sufficient memory in an MCU.

978-1-5090-6505-9/17/$31.00 © 2017 IEEE

A task usually uses a static memory space, where all the static variables and constants of the task are placed. Furthermore, a dynamically growing and shrinking memory exists for function calls, storing scope variables, and storing other temporary data. The mentioned dynamic memory is called *stack*. The *stack frame* is the total available stack space in which the *stack pointer* is pointing into the stack frame. It is used to access stack memory with an offset and indicates the threshold of valid and non-valid stack data. There are different approaches to handle stack frames in embedded systems: First, allocation of an individual stack frame for each task [2], [3]. Second, a common stack frame [4], [5] shared between all tasks; finally, a combination of both approaches [6]–[8].

The shared stack approach may reduce the overall memory space for stacks. However, it may also restrict the schedulability of individual tasks, because of nested stacks. A stack could not growth its stack memory, if the stack memory on the top was not owned by that task. Otherwise, it would overwrite stack memory of another task. Further, the stack shrinkage operation would lead to stack memory defragmentation, resulting in a huge management overhead by the Operating System (OS). Thus, tasks are not allowed be preempted and this fact may reduce the overall system performance and may violate real-time constraints especially for dynamically changing environments. Therefore, in most real-time embedded systems the OS allocates an individual stack for each task.

An alternative way to reduce stack memory are preemption thresholds [9], found in the ThreadX [10] OS. The OS uses the task's nominal priority for scheduling. If a task is scheduled, its priority will raise to its threshold priority, which must be defined above its nominal priority. On suspension, its priority falls back to its nominal priority. Thus, all other tasks prioritized with a lower nominal priority as the threshold priority of the currently running task cannot be scheduled. This approach could form non-preemptive task groups, where those groups can use a shared stack memory together, because they do not preempt to each other. Nevertheless, it is not possible to share the whole stack memory space among non-preemptive task groups and/or preemptive tasks if they do not use its maximum stack memory at a time.

A. Motivation

In an MMU-equipped embedded system, application developers do not have to define statically the required stack frame size for a task. The virtualization of the memory addresses allows the task to use a stack space up to the full VM address

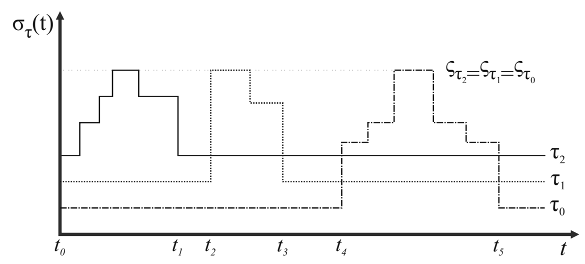


Fig. 1. Example stack consumption of three tasks $T = \{\tau_0, \tau_1, \tau_2\}$.

width. It is approximately consuming only that memory space that the task requires at a certain moment in time. If a task requires more stack memory, the MMU signals the OS to assign more memory to the requesting task [11]. If the task does not require that space anymore, the OS frees the non-used memory and makes it available for other tasks.

However, in MMU-less embedded systems, the memory is not virtualized, and the tasks are directly accessing the PM addresses. Thus, for the individual stack frame approach, developers have to define a static stack frame for each task with a specific size. If the required stack exceeds the assigned stack frame, a *stack overflow* will occur and could crash the whole system by overwriting non-owned data in the memory. To avoid stack overflows, in MMU-less multitasking systems, developers have to know the maximum required stack space ς_τ for each task $\tau \in T$, with T being the set of tasks.

The maximum required stack ς_τ can be analyzed with a static code analyzer tool, as for instance in [12]. These tools are aware of the stack usage $\sigma_\tau(t) \in \mathbb{N}_0$ of task τ at time $t \in \mathbb{N}_0$. However, if the entire system is unknown, because of a compositional developed system with some unknown software parts (e.g., libraries), the required stack size must be estimated by the developer. Thus, the risk to run into stack overflows is given. Stack overflows can also be detected at run-time by checking on each procedure call if a stack overflow occurred. However, this requires a manually inserting of checks or an automatically inserting by the compiler, leading to more program memory usage and longer run-time executions [13].

In a common MMU-less system with non-shared stacks, an individual stack frame is assigned for each task τ with its maximum required stack space ς_τ. Therefore, the totally assigned stack space $\tilde{S} \in \mathbb{N}_0$ in that system is computed as:

$$\tilde{S} := \sum_{\tau \in T} \varsigma_\tau = \sum_{\tau \in T} \max\{\sigma_\tau(t) \mid t \in \mathbb{N}_0\}$$

For each task the full stack frame is assigned; although, the stack frames are not completely used simultaneously. Thus, the individual stack frames are not utilized in an efficient way. Next, we illustrate this on an example, depicted in Fig. 1. Let's assume a multitasking embedded system with three tasks $T := \{\tau_0, \tau_1, \tau_2\}$ that are executing always in the same sequence, because of dependencies to each other (e.g., events). From time t_0 to t_1, task τ_2 is executing and uses its individual stack frame. From time t_2 to t_3 task τ_1 is executing and task τ_0 is executing from time t_4 to t_5. In the remaining time the OS is running, which is for clarity not depicted in the Fig. 1. It shows that for all three tasks $\tau \in T$ the maximum required stack space ς_τ of the individual stack is the same. Therefore, the developers have to assign an individual stack frame to each task, with at least the size of the maximum used stack. However, the individual stack frames are not simultaneously fully utilized, which leads to a worse stack memory utilization.

B. Dynamic Stack Sharing

Our concept aims to solve the issue of the wasteful stack space utilization of an individually assigned stack frame to each task. This is achieved with the assistance of a hardware extension, the *StackMMU*. It enables the possibility to share the whole stack space among all tasks with a minimum intervention of the OS by using VM addressing for the stack, leading to the following properties:

- The whole stack space is shared by all tasks. Depending on the task scheduling, the totally assigned stack space may be temporary reduced, compared to the traditional individual stack frame allocation approach (i.e., $\tilde{S}$).
- The assignment and deassignment of stack memory to a task and the memory accesses are executed by the hardware extension in a predictable time.
- If a task runs into a stack overflow, the StackMMU detects this and forwards the handling of this issue to the OS.
- If all tasks together require too much stack space, the StackMMU also forwards this issue to the OS, which is responsible to handle this problem.

Section II introduces the design of our concept and its implementation details are shown in Section III. Section IV analyses the properties theoretically and with a use case study, which is compared with other related works in Section V. Section VI discusses the applicability of the concept; and finally, Section VII concludes this work.

II. Design

The key idea of our design is to eliminate the statically individually assigned stack frame to a task; however, it will be assigned stack memory only on task's demand. If a task requires more stack memory, the hardware extension automatically allocates more stack memory in a predictable time, without the influence of the OS. If the task does not require the stack memory anymore, the hardware extension automatically deassigns the corresponding memory. Consequently, the totally assigned stack space $S \in \mathbb{N}_0$ leads to the next equation under the assumption that no additional memory is required for handling the StackMMU:

$$S := \max\left\{\sum_{\tau \in T} \sigma_\tau(t) \mid t \in \mathbb{N}_0\right\}$$

This means, that the totally assigned stack space S is bounded to the maximum used stack of all tasks at a time. Furthermore, stack overflows and out of stack conditions are detected by the hardware extension and are forwarded to the OS to handle those issues.

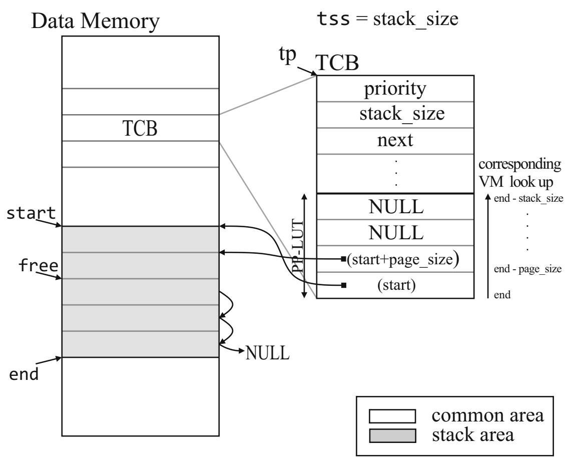


Fig. 2. Structure of our proposed StackMMU approach.

A. Structure

Now we are looking into the structure of our dynamic stack sharing approach, depicted in Fig. 2.

First, the data memory is divided into two areas. One area is the common area used with PM addresses as in common MMU-less embedded systems. The second area is the *stack area* used with VM addresses. The stack area is again divided into same sized *pages*, where the size is configurable as a power of two. The pages are handled as a linked list, whereas the register *free* points to the top of non-used pages. Further, the start and the ending of the stack area are defined with the registers *start* and *end*, respectively. The OS must initialize all of the mentioned registers at start-up. Furthermore, it has to initialize each page in the stack area with the address of the next available page. For indicating the last page, it is initialized with `NULL`.

Second, the OS stores the task control blocks (TCBs) into the common memory area. The TCB contains information about its associated task (e.g., priority, queue pointers). The number of task control items is defined by the OS and is thereby constant. However, some additional content is appended at the end of the TCB, namely the *Page Pointer Look Up Table* (PP-LUT), which contains the *page pointers*. Page pointers are used to access the pages in the stack area by the VM address.

During the creation of the tasks, the OS appends as much spaces as required for the PP-LUT, in the TCB, to support the developer's maximum defined stack size. Further, the task's stack pointer has to be initialized to the same address as the `end` register's value, which is the highest VM address in the system. In the control part of the TCB, task's maximum stack size must be stored, allowing the MCU to be aware of the maximum stack size of the currently running task. On a context switch, the MCU writes the stack size into the MCU's register *task stack size* `tss`. The register `tss` is used to detect a stack overflow and to look up the correct page pointers in the PP-LUT for stack access operations.

B. Stack Access

On common architectures, the stack pointer serves the purpose to point to the top of valid stack memory. It enables the software to have relative references to items in the stack memory [11]. However, the computation unit may also access the items in the stack with an absolute address, depending on the compiler and the architecture.

Therefore, our hardware extension detects a memory access in the stack area if the memory address is between `start` and `end`. This range indicates that the memory access is a VM address; then, the StackMMU triggers the memory access through the PP-LUT for the currently running task.

The next equation shows the calculation of the page base address pm_addr_base, which represents the page base address of the corresponding VM address of a task's stack access:

$$pm\_addr\_base = \text{PP-LUT}\left(\left\lfloor\frac{\texttt{tss} + vm\_addr - \texttt{end}}{\texttt{page\_size}}\right\rfloor\right)$$

In mostly all computer systems, with stack pointers, the stack pointer grows from high addresses to lower addresses. Thus, the PP-LUT also grows from the last page pointer to the first page pointer (see Fig. 2).

To access a stack item in a page, the page address offset pm_addr_offset is calculated with the following equation:

$$pm\_addr\_offset = vm\_addr \bmod \texttt{page\_size}$$

Finally, the two above mentioned equations can be combined together for calculating the PM address pm_addr located in the stack area:

$$pm\_addr = pm\_addr\_base + pm\_addr\_offset$$

Whereas a memory access into the common area represents a PM address and is still handled as usual.

C. Stack Pointer Change

Next, we show how the hardware extension assigns and deassigns a page to a task on its stack pointer `sp` change. A stack pointer `sp` change can be performed through register addition and subtraction instructions (e.g., `addi sp,sp,-16`) or through push and pop instructions. In most computer systems, these stack pointer modification have no side effects. However, in our approach the stack pointer `sp` has the additional purpose to assign and deassign a stack page to and from the PP-LUT, respectively. The hardware extension behaves differently, depending on the stack address' change:

1) If the stack pointer `sp` address grows or shrinks and does not traverse a page boundary, the hardware extension does nothing.
2) If the stack pointer `sp` grows over a page boundary, a new page is assigned by the hardware extension to the currently running task. The hardware extension stores the address of the first free page (by using the `free` register) into the page pointer specific location in the PP-LUT. Additionally, the address of the next free page is read and the `free` register is updated with that address.

3) If the stack pointer shrinks over a page boundary, the hardware extension removes the last valid page in the PP-LUT, by reading the corresponding page address. In this page, the reference to the next free page is updated to the value of the `free` register and the `free` register is updated with the address of the recently freed page, simultaneously.

The tasks let grow and shrink the stack pointer `sp` (e.g., function call) and are not overwriting the stack pointer `sp` directly with a new value. However, the OS does. To prevent the triggering of the stack assignment and deassignment, the hardware extension is only enabled if the OS is not running.

D. Stack Overflow Detection

For MMU-less embedded systems, one of the developer's jobs is to find the required stack size for each task. Underestimating the stack size could lead to unexpected run-time behavior. To detect this underestimation at run-time, the StackMMU checks if the stack pointer grows over the maximum defined stack size of a task. This means, if

$$\texttt{start} \leq \mathit{vm\_address} < \texttt{end} - \texttt{tss}$$

holds, the hardware extension raises an exception and the OS has the responsibility to handle this issue.

E. Out of Stack Detection

For all tasks running in an MCU, the developer defines with the registers `start` and `end` the stack area that holds the completely available stack space. The `page_size` register divides this area into pages that may be allocated to tasks. Therefore, the number of pages are limited. If tasks require more pages than available, the hardware extension will raise an exception. As for the stack overflow detection, that issue has to be handled in the OS. The hardware extension detects the out of stack condition, if the `free` register points to NULL and a task requires a new stack page.

F. Compiler Changes

To support our dynamic stack sharing approach, some modifications in the compiler might be needed. In some instruction set architectures (ISAs) a single instruction can modify the stack pointer arbitrarily, growing or shrinking over more than one stack page. However, to add or remove a new stack page in predictable time, the growth and shrinkage size has to be restricted to the size of one stack page.

III. Implementation

We implemented the proposed approach into our research project *mosart*MCU. In the following sections, we describe the relevant points of the *mosart*MCU, our own developed OS *mosart*MCU-OS, and the implementation details of the hardware extension.

A. mosartMCU

The *mosart*MCU[1] project aims to implement more OS awareness into embedded multi-core systems. The MCU core is based on the open RISC-V [14] architecture, maintained by the University of California, Berkeley. The offered open source Verilog implementation, named vscale[2], is the code base for our *mosart*MCU. It implements 32 bit integer and the multiplication/division instructions [15], executing the instructions in a three stage pipeline. The implementation provides 32 registers; whereas, the compiler does not touch the register `tp`, which is only maintained by the OS. This register indicates the TCB of the currently running task. The TCB contains information about the task including its priority. We extended the basic implementation, with an automatic read operation triggered by the hardware if the register `tp` is changed. Therefore, the hardware is always aware of the currently running task's priority. The automatic read operation is done in parallel to the normal execution flow by using an additional connection to the data memory through a dual-port memory. This dual-port memory is also used by some other extensions. Further, the MCU specification defines four different operating modes, whereas the *mosart*MCU supports only the non-privileged *user*-mode and the privileged *machine*-mode. These operating modes define permissions for some instructions and for accessing control status registers (CSRs), which are hardware registers used to configure and to get information from the MCU.

B. mosartMCU-OS

The dynamic stack sharing concept is based on pages. Supporting the hardware extension to find free pages, the *mosart*MCU-OS has to do some initialization steps in the initialization phase: The `free` CSR is set to the same address as the `start` CSR, which presents the first address in the stack area. The `end` CSR is also initialized to indicate the end of the stack area. Further, the initialization code has to set the next free page to the first address in each page. For the last page, the first address has to be set to `NULL`, supporting the hardware extension to detect an out of stack condition. These are the one and only steps that the OS has to perform to support our concept.

C. Dynamic Stack Sharing

The *mosart*MCU uses the register `tp` to point to the currently running task's TCB. The change of the register `tp` triggers the hardware to read automatically the priority and the stack size of the task. The stack size is stored into the CSR `tss`, which is used to look up page pointers for the VM addresses. To avoid an arithmetic division in hardware for calculating the final PM address, the page size `page_size` must be aligned to the power of two. Thus, the division can be replaced with a simple shift operation.

In addition to the task pointer `tp`, modifying the stack pointer `sp` triggers a functionality in our hardware extension. It may perform a stack page assignment or deassignment for the task. These operations are divided into two phases. Fig. 3 shows an example of the two different phases for a stack

[1]multi-core operating-system-aware real-time MCU

[2]https://github.com/ucb-bar/vscale

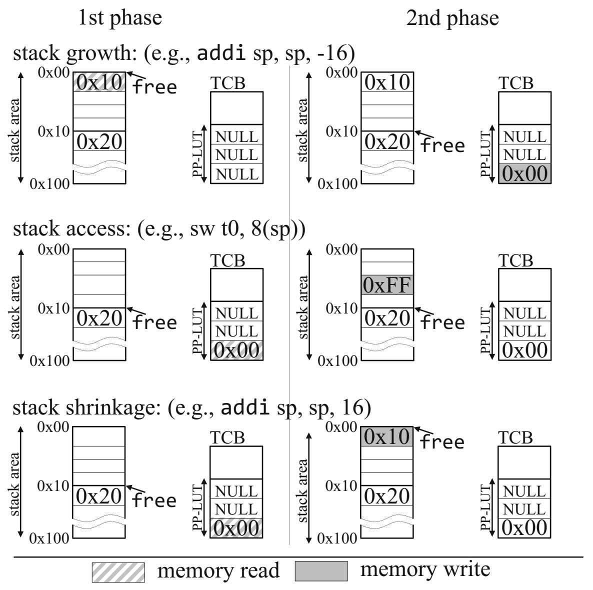

1st phase
2nd phase
stack growth: (e.g., addi sp, sp, -16)
stack access: (e.g., sw t0, 8(sp))
stack shrinkage: (e.g., addi sp, sp, 16)
stack area
0x00
0x10
0x100
0x10
0x20
0xFF
0x00
free
TCB
PP-LUT
NULL
memory read
memory write

Fig. 3. Dynamic stack sharing phases for stack growth, access, and shrinkage.

growth, access, and shrinkage operation:
Stack growth: First, the hardware extension reads the address of the next free page from the page where the CSR `free` is pointing (i.e., $0x10$). Second, it stores the CSR `free` value (i.e., $0x00$) into the corresponding PP-LUT location and concurrently updates the CSR `free` with the read address of the next free page.
Stack access: First, the hardware extension reads the page base address pm_addr_base from the PP-LUT; second, it uses the read page base address pm_addr_base and the page offset pm_addr_offset to access the PM address pm_addr of the corresponding VM address.
Stack shrinkage: The hardware extension has to free the last valid page in the PP-LUT. First, it reads the corresponding page base address pm_addr_base from the PP-LUT (i.e., $0x00$). Second, the hardware extension updates the reference of the next free page with the CSR `free` value in the corresponding page. Concurrently, the CSR `free` is updated with the read page base address pm_addr_base. In the PP-LUT the old page pointer is not changed; however, it is not used anymore, but will be overwritten on a new stack growth operation on this at this page pointer.

In all three operations, the first phase performs a read access in the memory and in the second phase a read or write access into the memory. No further memory accesses are required. Hence, if the memory access is executed in a constant time, also all three operations are executed in a constant time leading to a predictable stack growth, access, and shrinkage operation.

D. Control Unit adaption

All of those mentioned operations are implemented into the control unit of the *mosart*MCU. The control unit is responsible to operate control signals for the instructions, as for instance the memory control signals. If the control unit detects a stack pointer change or a memory access in the stack area while the MCU runs in the user-mode, the StackMMU is involved. First, it will change the memory control signals to perform the first phase of reading the required information. Second, it will use the read data to perform the second phase. Thereby, the control unit operates the execution flow of the instruction; it is also able to stall it. Thus, the control unit stalls the execution flow if required, as for instance done for reading data from the common memory area, too.

IV. Evaluation

Next, we discuss the evaluation of the StackMMU. First, we theoretically analyze the stack memory usage and the overhead. Second, we analyze the timing overhead theoretically and in our testbed. Third, we investigate the implementation utilization, timing behavior, and power consumption in the Xilinx Artix 7 Field Programmable Gate Array (FPGA) assembled on a Digilent Nexys4 DDR board. Finally, an experimental evaluation is shown.

A. Memory usage and Overhead

The dynamic stack sharing concept is based on pages. Thus, the stack space grows and shrinks in multiples of the page size `page_size`. Consequently, the sum of all currently used stack spaces $U(t, \texttt{page\_size}) \in \mathbb{N}_0$ at time t is calculated as follows:

$$U(t, \texttt{page\_size}) = \sum_{\tau \in T} \left\lceil \frac{\sigma_\tau(t)}{\texttt{page\_size}} \right\rceil \cdot \texttt{page\_size}$$

With the CSRs `end` and `start` in the data memory, the developer defines the size of the totally assigned stack space S':

$$S' := \texttt{end} - \texttt{start}$$

Whereby, the totally assigned stack space S' must be a multiple of the page size `page_size`; otherwise, an unexpected runtime behavior would occur.

To avoid out of stack conditions, the currently used stack space U is not allowed to exceed the totally assigned stack space S':

$$\forall t \in \mathbb{N}_0, \quad U(t, \texttt{page\_size}) \leq S'$$

To manage the StackMMU, it requires some additional memory space for storing the page pointers in the PP-LUT. Furthermore, if the currently stack usage σ_τ of task τ does not fit exactly into a page, some space in the page remains empty. Thus, the data memory overhead O for the whole system is calculated as follows:

$$O(t, \texttt{page\_size}) :=$$
$$\sum_{\tau \in T} \left(\underbrace{\left\lceil \frac{\varsigma_\tau}{\texttt{page\_size}} \right\rceil}_{\text{PP-LUT size}} + \underbrace{(\sigma_\tau(t) \bmod \texttt{page\_size})}_{\text{remaining free space in a page}} \right)$$

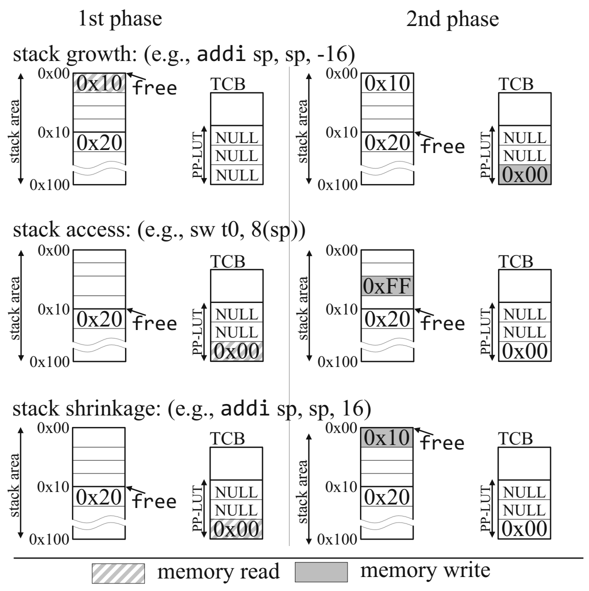

1st phase
2nd phase
stack growth: (e.g., addi sp, sp, -16)
stack access: (e.g., sw t0, 8(sp))
stack shrinkage: (e.g., addi sp, sp, 16)
stack area
0x00
0x10
0x100
0x10
0x20
0xFF
0x00
free
TCB
PP-LUT
NULL
memory read
memory write

Fig. 3. Dynamic stack sharing phases for stack growth, access, and shrinkage.

growth, access, and shrinkage operation:
Stack growth: First, the hardware extension reads the address of the next free page from the page where the CSR `free` is pointing (i.e., $0x10$). Second, it stores the CSR `free` value (i.e., $0x00$) into the corresponding PP-LUT location and concurrently updates the CSR `free` with the read address of the next free page.
Stack access: First, the hardware extension reads the page base address pm_addr_base from the PP-LUT; second, it uses the read page base address pm_addr_base and the page offset pm_addr_offset to access the PM address pm_addr of the corresponding VM address.
Stack shrinkage: The hardware extension has to free the last valid page in the PP-LUT. First, it reads the corresponding page base address pm_addr_base from the PP-LUT (i.e., $0x00$). Second, the hardware extension updates the reference of the next free page with the CSR `free` value in the corresponding page. Concurrently, the CSR `free` is updated with the read page base address pm_addr_base. In the PP-LUT the old page pointer is not changed; however, it is not used anymore, but will be overwritten on a new stack growth operation on this at this page pointer.

In all three operations, the first phase performs a read access in the memory and in the second phase a read or write access into the memory. No further memory accesses are required. Hence, if the memory access is executed in a constant time, also all three operations are executed in a constant time leading to a predictable stack growth, access, and shrinkage operation.

D. Control Unit adaption

All of those mentioned operations are implemented into the control unit of the *mosart*MCU. The control unit is responsible to operate control signals for the instructions, as for instance the memory control signals. If the control unit detects a stack pointer change or a memory access in the stack area while the MCU runs in the user-mode, the StackMMU is involved. First, it will change the memory control signals to perform the first phase of reading the required information. Second, it will use the read data to perform the second phase. Thereby, the control unit operates the execution flow of the instruction; it is also able to stall it. Thus, the control unit stalls the execution flow if required, as for instance done for reading data from the common memory area, too.

IV. Evaluation

Next, we discuss the evaluation of the StackMMU. First, we theoretically analyze the stack memory usage and the overhead. Second, we analyze the timing overhead theoretically and in our testbed. Third, we investigate the implementation utilization, timing behavior, and power consumption in the Xilinx Artix 7 Field Programmable Gate Array (FPGA) assembled on a Digilent Nexys4 DDR board. Finally, an experimental evaluation is shown.

A. Memory usage and Overhead

The dynamic stack sharing concept is based on pages. Thus, the stack space grows and shrinks in multiples of the page size `page_size`. Consequently, the sum of all currently used stack spaces $U(t, \texttt{page\_size}) \in \mathbb{N}_0$ at time t is calculated as follows:

$$U(t, \texttt{page\_size}) = \sum_{\tau\in T}\left\lceil\frac{\sigma_\tau(t)}{\texttt{page\_size}}\right\rceil \cdot \texttt{page\_size}$$

With the CSRs `end` and `start` in the data memory, the developer defines the size of the totally assigned stack space S':

$$S' := \texttt{end} - \texttt{start}$$

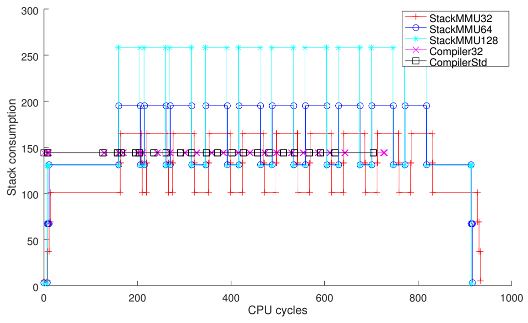
Whereby, the totally assigned stack space S' must be a multiple of the page size `page_size`; otherwise, an unexpected runtime behavior would occur.

To avoid out of stack conditions, the currently used stack space U is not allowed to exceed the totally assigned stack space S':

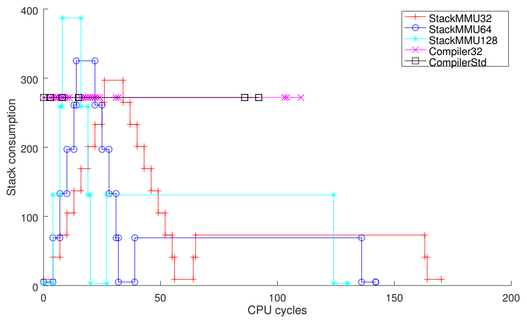
$$\forall t \in \mathbb{N}_0, \quad U(t, \texttt{page\_size}) \leq S'$$

To manage the StackMMU, it requires some additional memory space for storing the page pointers in the PP-LUT. Furthermore, if the currently stack usage σ_τ of task τ does not fit exactly into a page, some space in the page remains empty. Thus, the data memory overhead O for the whole system is calculated as follows:

$$O(t, \texttt{page\_size}) := \sum_{\tau\in T}\left(\underbrace{\left\lceil\frac{\varsigma_\tau}{\texttt{page\_size}}\right\rceil}_{\text{PP-LUT size}} + \underbrace{(\sigma_\tau(t) \bmod \texttt{page\_size})}_{\text{remaining free space in a page}}\right)$$



(a) Run-time behavior of a quicksort algorithm.



(b) Run-time behavior of a temporary buffer usage with 256 Bytes and following a bubblesort algorithm is executed.

Fig. 4. Stack consumption and required execution times for the StackMMU approach with different page sizes (32 Bytes, 64 Bytes, 128 Bytes), static individual stack frame allocation with the adapted compiler (i.e., compiler32) for page sizes of 32 Bytes and the standard compiler (i.e., compilerStd).

and temporary array usage with the bubblesort algorithm, respectively. The quicksort algorithm uses the divide and conquer technique; thus, the sorting algorithm works recursively. A recursive algorithm leads to many function calls and leaves; therefore, a function performs many stack changes at its beginning (prologue) and ending (epilogue). For each recursive function call the size of stack growing and shrinking remains constant, with a maximum of 144 Bytes. Thus, the stack consumption of the StackMMU concept is highly depending on the stack page size, because stack memory can be allocated only as a multiple of the stack page size. As a result, a stack page size with 128 Bytes consumes much more memory as all other approaches. However, it is the fastest one compared to the other StackMMU execution times, because the required instructions for stack growth and shrinkage is lesser. This timing influence can be seen isolated for the adapted compiler (i.e., `Compiler32`), which bounds the stack grows and shrinks to 32 Bytes. Compared to the standard compiler (i.e., `CompilerStd`), it requires some additional cycles because of splitting up the stack change. The execution of the StackMMU generally consumes more CPU cycles, because the StackMMU adds a cycle for stack growth, shrinkage, and access.

The second use case shows the same behavior as the first use case, whereas the maximum required stack is needed only for the first CPU cycles. For the other times, the stack consumption is only around one third compared to the static assigned stack. There, the free stack memory is available for other tasks.

V. Related Work

For real-time embedded systems, reducing temporal jitter on memory accesses is a challenging requirement to avoid an unknown real-time behavior. Further, memory is also a scarce resource in these systems; therefore, sharing unused memory is targeted.

Compared to the Context Save Areas (CSAs) and Context Lists of the Infineon's TriCore [16] our dynamic stack sharing approach is similar. However, the TriCore uses the CSAs for storing a part of the task context on function calls and interrupts. Their aim is to improve the context switch performance. Hence, there is no way to store local variables; therefore, the architecture is still using a stack pointer, which is pointing to a stack to handle locally allocated variables.

Schamani *et al.* [17] proposed a configurable MMU on a FPGA with pages. The results of the implementation show that the MMU adds temporal jitter to the memory access. On a hit, the PM base address is found in a cache, the access time is 5 cycles. On a miss, the PM base address is not in the cache, after 11 cycles the OS is notified to update the internal translation tables. In the MMU implementation from Hu-Cheung Ng *et al.* [18], on a cache hit the memory access time is 2 cycles and on a table update 16 cycles. Apart, a cache miss consumes 600 to 227 000 cycles depending on the OS that handles the miss. Further, the power consumption showed, that the MMU consumes more than 50 % of the overall power consumption.

The MMU implementation in the MC68851 [19] and the ARM Cortex A5 [20] are using a tablewalker in hardware instead of the software based searching in translation tables. The tablewalker searches, beginning on a base address in a register, translation item in the main memory. If the tablewalker does not find the correct one, an exception is raised to notify the OS to handle this problem in software. In addition, the tablewalker consumes some cycles that introduce jitter to the memory access.

Chu *et al.* [21] proposed a stack virtualization based on software. They are combining binary translation and a kernel to achieve a VM addressed stack on an MMU-less embedded system. Getting, setting, and reallocating the stack pointer consumes 45, 94, and 2326 cycles, respectively. It is an enormous impact on the system performance, which they showed with a use case.

In [22], Middha *et al.* propose a stack sharing solution without VM addresses. Thereby, a task uses unused stack space of statically assigned stack frames of another task. Thus, on each function the approach has to check for a stack overflow. If it occurs, a fixed sized page is allocated in another task's free stack space. If the stack consumption exceeds the page, another page is allocated in another free stack frame. Without optimization, the run-time and energy consumption increases up to 23 % and 24 %, respectively. With code analysis optimization, they reduced both to 3 %. However, the code analysis optimization restricts programming features (e.g., recursive functions, function pointers).

Yi *et al.* showed in [23] an approach to allocate stack to a task on demand. On compile time, a tool calculates the stack usage of each function call and manipulates the C code. The C code will call their *on demand stack library* with the calculated stack usages at the start and end of a function. The library internally uses a linked list, pointing to the next allocated stack area for a function. The run-time stack usage showed that the implementation uses only that memory that a task requires; nevertheless, for their use case the execution time of the application increases about 10 % over a long time.

VI. Discussion

Section IV showed theoretically and with a use case the evaluation of the StackMMU. It showed that the memory consumption and time overhead highly depends on the behavior of the work. For traditional MMU-less approaches the totally assigned stack space $\tilde{S}$ is the sum of all individual stack frames. Each individual stack frame of a task has to be the same size or greater as the task's maximum required stack space ς_τ to avoid stack overflows (see Section I-A). However, not all tasks usually use the full assigned stack frame at a time, resulting in a worse stack memory utilization. Whereas, in the StackMMU concept the unused stack memory is shared, may resulting in a better stack memory utilization. The utilization depends on the stack usage $\sigma_\tau(t)$ over time of a task τ, the page size `page_size`, and the scheduling dependency of all tasks to each other.

Therefore, if the task maximum required stack space is needed for the whole time, the memory saving of the StackMMU may be not there or even worse. However, if the required stack space possess only a short peak, the StackMMU takes advantage of sharing the unused stack memory. Here, the `page_size` influences the memory consumption and performance. On one side, a too small `page_size` increases the PP-LUT in the TCB and may increase the execution time, because of many required stack change instructions. On the other side, a too big `page_size` allocates big pages, which may be worse utilized; thus, it increases the memory overhead with a better performance. The same issue has to be handled when using an MMU. In this case, the page size also influences the memory utilization and the memory access times. The MMU introduces jitter to a memory access due to the usage of a cache. In our implementation, we are not using a cache; and thus, the latency for stack memory access, growth, or shrinkage introduces an additional constant cycle, which is faster and predictable compared to common MMUs.

Furthermore, the scheduling dependency of all tasks to each other impacts the sum of all currently used stack spaces $U(t, \texttt{page\_size})$ influencing the required totally assigned stack space S'. Thus, the developer must try to minimize the sum of all currently used stack spaces $U(t, \texttt{page\_size})$ by scheduling the task in a suitable way. In MMU-equipped embedded systems, the same approach must be used to improve the memory utilization.

The StackMMU implementation increases the power consumption by 2%. Compared to the related works in Section V this is a small additional required power consumption. Thus, our proposed hardware extension will suit well for real-time embedded systems that require low power consumption, dynamic stack sharing, and predictability as the IoT, Industry 4.0, or automotive industry.

Whereas, our proposed concept does not prevent tasks $\tau \in T$ to consume their maximum required stack space ς_τ on the same time t leading to no stack memory minimization. However, with the support of the OS' scheduling policy and our proposed StackMMU it will allow us to find a new solution.

VII. Conclusion and Outlook

MMUs are widely used in non real-time systems. In real-time systems, MMUs introduce temporal jitter and may result in real-time constraint violations. Therefore, MMUs are not used for those systems. However, the introduction of virtual memory supports the sharing of unused memory, which would be useful for memory constrained embedded systems. Therefore, in this paper we presented a new dynamic stack sharing concept based on a hardware extension and some knowledge of the internal OS-data structures (i.e., TCB). Our solution supports the virtualization of the stack addresses and only adds one cycle for stack growth, shrinkage, and access operations, resulting in a predictable operation time. The hardware extension assigns and deassigns a stack page automatically from a pool of stack pages and stores the references into the TCB of the requiring task. On stack access, the references in the TCB are read before the hardware extension performs the stack memory access on the computed address. The OS is not involved in these operations, but it has the responsibility to initialize the hardware extension properly at start-up. The handling of these operations was implemented into the *mosart*MCU project running in an Artix 7 FPGA. In the FPGA, the hardware extension consumes additionally 1066 LUT slices, 184 register slices, and 2 % more power, independent of the number of tasks and other hardware extensions. Further, the evaluation showed that the handling of the stack pages required some additional memory space for managing the pages. Nevertheless, depending on the task scheduling, in a use case scenario we showed that the stack area can be shared between tasks and the totally assigned stack space may be reduced compared to the traditional approaches, which waste memory by allocating individual stacks to each task.

As potential future work, we will investigate a new scheduling algorithm based on the idea of fixed-priority preemption threshold. However, instead of using priority thresholds we are planning to consider the required stack space for the next part of code as the threshold. Through the operating system awareness of our *mosart*MCU, it may support the online scheduler to find a feasible schedule in a way to satisfy still all the real-time requirements while minimizing the stack memory usage.

Acknowledgment

This research project was partially funded by AVL List GmbH and the Austrian Federal Ministry of Sciences, Research and Economy (bmwfw).

References

[1] J. Fotheringham, “Dynamic storage allocation in the atlas computer, including an automatic use of a backing store,” *Commun. ACM*, vol. 4, no. 10, pp. 435–436, Oct. 1961.

[2] Micrium, “uC/OS-III,” https://www.micrium.com/.

[3] Real Time Engineers Ltd., “FreeRTOS,” http://www.freertos.org/.

[4] Contiki, “Contiki: The Open Source OS for the Internet of Things,” http://www.contiki-os.org.

[5] T. P. Baker, “A stack-based resource allocation policy for realtime processes,” in *Proc. of the 11th IEEE Real-Time Systems Symposium (RTSS)*, Dec 1990, pp. 191–200.

[6] P. Gai, G. Lipari, and M. Di Natale, “Stack size minimization for embedded real-time systems-on-a-chip,” *Design Automation for Embedded Systems*, vol. 7, no. 1, 2002.

[7] R. Ghattas and A. G. Dean, “Preemption threshold scheduling: Stack optimality, enhancements and analysis,” in *Proc. of the 13th Real Time and Embedded Technology and Applications Symposium (RTAS)*, April 2007.

[8] C. Wang, Z. Gu, and H. Zeng, “Global fixed priority scheduling with preemption threshold: Schedulability analysis and stack size minimization,” *IEEE Trans. on Parallel and Distributed Systems*, vol. 27, no. 11, pp. 3242–3255, Nov 2016.

[9] Y. Wang and M. Saksena, “Scheduling fixed-priority tasks with preemption threshold,” in *Proc. of the 6th Int. Conference on Real-Time Computing Systems and Applications (RTCSA)*, 1999.

[10] Express Logic Inc., “ThreadX,” http://rtos.com/products/threadx/.

[11] T. A. Andrew S. Tanenbaum, *Structured Computer Organization*, 6th ed. Pearson, 2013.

[12] M. Eslamimehr and J. Palsberg, “Testing versus static analysis of maximum stack size,” in *Proc. of the 37th Annual Int. Computer Software and Applications Conference (COMPSAC)*, July 2013, pp. 619–626.

[13] S. Biswas, M. Simpson, and R. Barua, “Memory overflow protection for embedded systems using run-time checks, reuse and compression,” in *Proc. of the Int. Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. New York, NY, USA: ACM, 2004, pp. 280–291.

[14] RISC-V Foundation, “RISC-V,” https://riscv.org/.

[15] A. Waterman, Y. Lee, R. Avizienis, D. Patterson, and K. Asanovic, *The RISC-V Instruction Set Manual*, 2016.

[16] *TriCore V1.6*, Infineon, May 2012.

[17] F. Shamani, V. F. Sevom, J. Nurmi, and T. Ahonen, “Design, implementation and analysis of a run-time configurable memory management unit on fpga,” in *Proc. of the Nordic Circuits and Systems Conference (NORCAS): NORCHIP International Symposium on System-on-Chip (SoC)*, Oct 2015, pp. 1–8.

[18] H. C. Ng, Y. M. Choi, and H. K. H. So, “Direct virtual memory access from FPGA for high-productivity heterogeneous computing,” in *Proc. of the Int. Conference on Field-Programmable Technology (FPT)*, Dec 2013, pp. 458–461.

[19] B. Cohen and R. McGarity, “The Design And Implementation of the MC68851 Paged Memory Management Unit,” *IEEE Micro*, vol. 6, no. 2, pp. 13–28, April 1986.

[20] *Cortex-A5 MPCore Technical Reference Manual*, ARM Limited, 2009.

[21] R. Chu, L. Gu, Y. Liu, M. Li, and X. Lu, “Sensmart: Adaptive stack management for multitasking sensor networks,” *IEEE Trans. on Computers*, vol. 62, no. 1, pp. 137–150, Jan 2013.

[22] B. Middha, M. Simpson, and R. Barua, “Mtss: Multitask stack sharing for embedded systems,” *ACM Trans. Embedded Computer Systems*, vol. 7, no. 4, pp. 41:1–46:37, Aug. 2008.

[23] S. Yi, S. Lee, Y. Cho, and J. Hong, *OTL: On-Demand Thread Stack Allocation Scheme for Real-Time Sensor Operating Systems*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4490, pp. 905–912.

D. Task Priority aware SoC-Bus for Embedded Systems

Publication Information

F. Mauroner and M. Baunach. Task Priority aware SoC-Bus for Embedded Systems. *In Proceedings of the 19th International Conference on Industrial Technology (ICIT)*. Lyon, France. February 2018.

Contribution

Main author

Task Priority aware SoC-Bus for Embedded Systems

Fabian Mauroner and Marcel Baunach
Institute of Technical Informatics
Graz University of Technology, Graz, Austria
Email: {mauroner, baunach}@tugraz.at

***Abstract*—System-on-a-Chip (SoC)-buses are designed to communicate from masters to slaves. In a multi-master system, the masters combat for accessing the slaves. In the past, different arbitration algorithms have been invented to grant access to the slaves. However, these algorithms are not aware of the currently running task's priority that wants to access the slaves; thus, a lower prioritized task could access the slave first. Other solutions with priority awareness use additional wires and support only a few priority levels that cannot be mapped from a task priority. In this paper, we present a SoC-bus that is suitable for hard real-time systems, where the highest combating prioritized task immediately gets access to the addressed slave. The lower prioritized tasks are stalled, what is managed by the interconnect logic. We implemented our proposed approach into a Field Programmable Gate Array (FPGA) and we show the required hardware resource consumption. Further, we demonstrate the considerations of the task priorities in a use case scenario.**

Keywords—Embedded systems; Multi-Core; SoC-Bus; Priority-Awareness; FPGA implementation

I. Introduction

The complexity of embedded systems continues to grow. In the past, embedded systems were developed for a specific application; but today, they must be able to adapt to the outside world. This requires a huge computation power, to handle all these dynamics in software. One way to handle that huge computation power is to use more computation units, so-called multi-core systems. However, there the access to shared resources (e.g., memories, peripherals, etc.) is critical and must be executed as efficient as possible, otherwise it becomes the bottleneck of the whole system. Particularly for hard real-time systems, as automotive or avionic systems, the access to shared resources is crucial, because it may influence accuracy, schedulability, and the costs of the systems.

The aim of a real-time system is to fulfill all the time boundaries [1], with an upper time bound and/or a lower time bound. The former, defines that a functionality must be finished within the upper time bound. The latter, defines the earliest time on which a task could start. A violation of these time boundaries could result in a dramatic situation, where machines could be destroyed or even human lives could be influenced. A system with such strong time boundaries is called *hard real-time system*.

In the past, there have been investigated a lot of new techniques to improve hard real-time systems and to prove the strict observance of all real-time requirements. On the software layer, the Operating System (OS) is responsible for scheduling all the tasks, which are parts of an application, in a suitable way. Thus, tasks are scheduled in a way to satisfy all real-time requirements. To prove if all real-time requirements are satisfied, the Worst Case Execution Time (WCET) of each task is required. The WCET represents the maximum time that a task requires to execute for its work. By contrast, the Best Case Execution Time (BCET) and Average Case Execution Time (ACET) are the shortest and the average times that a task requires to finish its work, respectively.

In contrast to the software layer, also the hardware layer has been researched to support hard real-time applications (e.g., pipelining, interrupts, caches, memories, or System-on-a-Chip (SoC) buses). These adaptations may affect the BCET and the WCET.

To prove the schedulability of hard real-time systems, the schedulability analysis [2] takes into account the WCET, despite the ACET is far away from it. Therefore, for real-time systems, the hardware layer should be designed to bring the WCET closer to the BCET. Thus, the reduced jitter results in a more realistic schedulability analysis. Especially, the SoC-bus is a crucial part, which must be handled in an efficient and predictable way.

The master is always admitted to access the slaves if there is only one master in a SoC-bus (e.g., single core systems). There is no other master that would block or interleave its access to the slaves. In a multi-master system (e.g., multi-core) a master could be blocked or interleaved by another master. There exist many different bus topologies for accessing slaves by masters. Each approach has its advantages and disadvantages [3]. The most common topologies for SoC-buses are the *linear bus* and the *crossbar switch*. If two or more masters combat to access the same slave, the bus has to solve that conflict, what is named *arbitration* [3].

In this paper, we present a priority aware SoC-bus including an arbitration policy that uses the currently running task's priority, leading to a predictable SoC interconnect bus.

The next sections are structured as follows: Section II investigates related works. Section III discusses fundamentals to get the basics for following the proposed task priority aware SoC-bus presented in Section IV. The evaluation of our approach is shown in Section V. Lastly, Section VI concludes this work.

II. Related Work

Priority awareness in SoC-bus communication is only required for multi-master bus systems. Each single-master could be isolated to work in a multi-master system. Nevertheless, single-master bus protocols as the ARM AMBA AHB [4] or the Altera Avalon [5] are not specifying priorities in their specification. The open Wishbone [6] specification does not specify a priority, too. The specification passes the arbitration technique to the developer.

The ARM AMBA specification [7] specifies the usage of additional wires connected to an arbitration logic. However, the specification limits the number of masters to 16 and does not specify how to set the priorities of each master.

IBM specifies the Processor Local Bus (PLB) [8] with priority awareness. The PLB is responsible for admitting access to the master with the highest priority. The priority contains four levels, for which two additional lines from the master to the arbitration logic must be used. Thus, the priority levels are limited to four, which is not feasible for task priorities. Moreover, for the priority level additional wires are required.

Many known and widely used arbitration algorithms are not deterministic, therefore not suitable for hard real-time systems. Further, they do not utilize the full available bandwidth: For static assigned priorities [9], the highest prioritized master gets always access to the slave and could starve other masters. All this is independent of the task priorities, which at the end are performing the operations with the slave. Thus, a lower prioritized task could get access instead of the higher prioritized task, what would lead into a priority inversion.

Another approach is Time Division Multiple Access (TDMA) [10], where each master gets a time slot for communicating with the slave. This approach enables predictability, but the communication cannot be used consecutively; thus, the WCET of a task may increase. Here, other masters are interleaving the highest prioritized master, so the whole system performance may be reduced.

Similar to TDMA, round robin [9] passes the access to a slave to the next accessing master. Thereby, the next master is statically chosen and no priorities are considered. The properties of this approach are the same as for TDMA, but here the ACET may improve.

The lottery [11] arbitration is a probabilistic one, which considers the last requesting master and a random number to grand access to an accessing master. However, this approach does not have an upper time bound for an admitted access; consequently, it is not predictable for hard real-time systems.

Compared to the mentioned related works, our approach adds task priority awareness to the SoC-bus that avoids priority inversion and leads to predictable task executions and full bus utilization without using additional wires for the priorities.

III. Background

In this section, we introduce fundamentals of the proposed priority aware SoC-bus.

A. System Structure

In a system with more masters, the access has to be controlled by considering the priorities of the masters. In our approach, the arbiter uses the current priority $\pi_m \in \Pi \subseteq \mathbb{N}$ of the master $m \in M$. Whereas, M is the set of all masters that has access to the slaves $s \in S$. The m's current priority π_m maps the priority of the in master m currently running task $\tau_{m,run} \in T_m$, which is known by the *mosart*MCU. Thereby, the task priorities are spread among all tasks on different masters and each priority indicates the importance of the respective task (e.g., Rate Monotonic (RM) [2]).

B. mosartMCU

The *mosart*MCU[1] [12] project implements OS awareness into embedded, real-time, and multi-core systems. The Microcontroller Unit (MCU) is based on the open RISC-V [13] architecture, maintained by the University of California, Berkeley. They offer an open source Verilog implementation, named vScale[2], which is the base for the *mosart*MCU. It implements the 32 Bit integer and multiplication/division instructions [14]. The SoC-bus is 32 Bit wide and a memory access is naturally aligned to the used data type. The implementation provides a register file with 32 registers, whereas the compiler dose not touch the register `tp`. This register is thought for indicating the currently running task $\tau_{m,run}$ in the OS. By using the register `tp` and the knowledge of the internal OS structure, the MCU becomes aware of the currently running task $\tau_{m,run}$'s priority, which is mapped to m's current priority π_m.

C. mosartMCU-OS

In each core of the *mosart*MCU runs an instance of our own developed *mosart*MCU-OS (i.e., partitioned RM scheduling [15]). Its main design decision is that the whole OS functionalities (e.g., syscall, kernel, etc.) are running in kernel-mode. The tasks are running in the user-mode, and may call OS related functionality by calling a syscall with a special instruction. The instruction leads to the kernel-mode and jumps into the OS; there, the context is saved and the syscall is executed. After handling the syscall, the scheduler may select a new task by changing the register `tp`. After some additional administrative work, the context of the scheduled task is restored and the Central Processing Unit (CPU) continues in the user-mode by executing task $\tau_{m,run}$'s program code. Thereby, the context restore phase restores all in memory saved registers back into the registers. Further, this phase performs some non-memory related instructions. This property is used later on to ensure the proper task priority assignment to the master m.

D. Bus Signals

Here we demonstrate the bus signals used by the *mosart*MCU. The SoC-bus is designed to get along without three-state signals; thus, it can be directly implemented into a Field Programmable Gate Array (FPGA). First, we present the signals directed from master to slave:

EN	The 1 Bit enable signal starts a read or write transfer.
WEN	The 1 Bit write enable signal defines a read or a write transfer if it is `LOW` or `HIGH`, respectively.
SIZE	On a bus transfer, the transfer can be either a byte, half-word, or a word access. Therefore, the 2 Bit size signal distinguishes between the three bus wide transfers.
ADDR	This 32 Bit signal selects the address in the memory area for the read or write operation.
WDATA	This 32 Bit signal updates the content on address ADDR at a bus write operation.

[1]Multi-Core Operating-System-aware Real-Time MCU
[2]https://github.com/ucb-bar/vscale

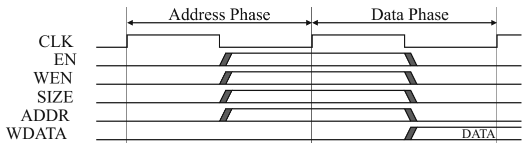


Fig. 1. Write data transfer demonstrating the address and the data phases.

Second, we present the remaining bus signals directed from the slave to the master, what reflects the answer of the initiated bus transfer:

RDATA	This 32 Bit signal represents the read content at a bus read operation on address `ADDR`.
WAIT	The access to the address `ADDR` may take more than one cycle. Thus, this waiting signal notifies the master, that the slave is busy and not even ready to undertake the `WDATA` or to provide the `RDATA` for a write or read operation, respectively.
BAD	An active BAD signal notifies the master that the address `ADDR` is invalid.

The mentioned signals are the minimum required signals to perform a read or write bus transfer for the *mosart*MCU. It is possible to add more signals to support even more features, as for instance the AMBA 3 AHB-Lite [16] does: burst mode, master lock, or protection mechanisms. However, these must be handled properly.

E. Bus Transfer Protocol

The bus transfer protocol specifies the relationship and timing of all bus signals. The used bus protocol follows the address and data phase approach used in the ARM's AMBA AHB buses [4]. In the address phase, the address is chosen with the memory size `SIZE` and write enable `WEN` signals. In the data phase, the data is transferred. Thereby, the data is send from a master to a slave by the `WDATA` signal and from a slave to a master by the `RDATA` signal on a write or read access, respectively. Fig. 1 depicts a write transfer, with the two phases. The two missing control signals (i.e., `WAIT` and `BAD`) are not depicted. However, both signals would only be relevant in the data phase. The `BAD` signal notifies the master if the address was invalid, and the `WAIT` signal will prolong the data phase if the slave is busy. Thus, the data transfer signals (`WDATA` or `RDATA`) are not overtaken as long as the `WAIT` signal is active. If the `WAIT` signal is deasserted, the data signals are overtaken completes the transfer.

The division of an address and data phase enables a pipelined bus transfer. Thus, an address phase may start while the data phase of the previous address phase is ongoing. Thereby, the bus access requires two cycles (if the slave is not busy) but the maximal throughput is the full bus width times the bus frequency.

With this bus transfer protocol, the bridge between our bus protocol specification and the ARM AMBA 3 AHB-Lite is straightforward and no internal registers for holding control or data signals are required. Therefore, the bus bridge can be implemented efficiently only with combinational logic.

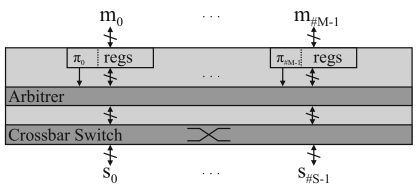


Fig. 2. Isolated masters connected to the interconnect.

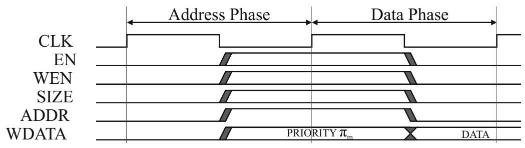


Fig. 3. Write data transfer example with priority transfer.

IV. Task Priority Aware SoC-Bus Communication

Here we present the task priority aware SoC-bus. First, we present the underlying structure and the extension of the bus transfer protocol. Second, we show the arbitration strategy; and last, some properties of our approach are shown and proved.

A. System Structure

The fundamental bus protocol is designed to simplify the bus transfer protocol. This leads to a minimalist and easy to implement design. The drawback of this simple protocol is that the protocol does not support multi-masters. Therefore, we propose to use an interconnect that isolates all masters from each other, as depicted in Fig. 2. This mentioned structure connects all masters to the interconnect enabling support for multi-master accesses. Further, it can locally perform all the required steps for the arbitration, leading to priority aware bus accesses on concurrent slave accesses.

The interconnect possesses for each master m instead of a 1 Bit enable signal a $\#S$ Bit enable signal $\nu_m := s$ that selects a slave s. Thus, the address resolving for a slave is done in the masters and not in the interconnect logic.

B. Bus Transfer Protocol Extension

To add priority awareness to the fundamental bus protocol, the `WDATA` signal is used. In the fundamental protocol, the `WDATA` signal is not used while the first address phase is executed. There, our proposed bus transfer protocol applies the priority, as depicted in Fig. 3. Unless an address phase was performed in the previous cycle, the `WDATA` signal represents the m's current priority π_m. Otherwise, the data phase of the previous address phase is concurrently performed with the new address phase. Thus, in a pipelined access, only the first address phase contains the priority on the `WDATA` signal. For the remaining bus accesses the priority is internally stored in the interconnect logic to preserve the priority awareness.

C. Arbitration strategy

In a multi-master system, more than one master m may select the same slave s. Thereby, the priority π_m

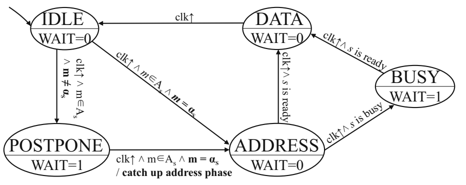


Fig. 4. Interconnect finite state machine.

of a master m shows the importance for accessing the slave s. Thus, the highest prioritized master must grant access to it and the lower prioritized masters have to wait, to avoid a priority inversion [17]. This leads to different interconnect states that a bus communication possesses: $\Gamma := \{\text{IDLE}, \text{POSTPONE}, \text{ADDRESS}, \text{BUSY}, \text{DATA}\}$. Thereby, each master $m \in M$, connected to the interconnect, possesses an interconnect state $\gamma_m \in \Gamma$.

The arbiter admits access to a slave s to one of the requesting masters A_s:

$$A_s := \{m \in M \mid (\gamma_m = \text{IDLE} \land \nu_m = s) \lor \gamma_m = \text{POSTPONE}\} \quad (1)$$

The requesting masters A_s for slave s set contains all masters that are accessing the slave s or are postponed for accessing it. Out of this set, only the master with the highest priority gets access to the slave s. Thus, the slave s accessing master α_s is defined as:

$$\alpha_s := \left\{a \in A_s \mid \pi_a = \max_{\forall x \in A_s} \{\pi_x\}\right\} \quad (2)$$

Fig. 4 depicts the state-transitions of a master m connected to the interconnect. In the state IDLE the master m is not using the bus. If the master m initiates a slave access, m's interconnect state γ_m will be moved to POSTPONE or ADDRESS. In the state POSTPONE, the bus signals are not put through, because a higher prioritized master accesses the same slave. In the state ADDRESS the address phase is executed by putting trough the control signals from the master m. The interconnect puts through only the control signals and not the data signals; therefore, the master m's current priority is not passed through. After the state ADDRESS follows the state DATA, which performs the data phase if the slave is not busy. Otherwise, the master will stay in the state BUSY as long as the slave s is busy.

In the transaction from POSTPONE to ADDRESS the address phase signals of the master are caught up. Thus, the interconnect internally stores the address phase signals on master's address phase for catching up later. Furthermore, in the state POSTPONE the `WAIT` signal to the master is asserted, what leads to a stalled master.

Due to the used task priorities, equal priorities may occur. For that, either it is forbidden to assign same priorities on different masters or the arbitration has to use a backup policy. Many different backup policies are applicable (e.g., static priority, round robin, etc.). Each policy will influence the access times, why it has to be chosen carefully.

D. Properties

The arbitration policy leads us to the next properties:

Theorem 1: The master priority π_m in the interconnect is always equal master m's currently running task priority $\tau_{m,run}$.
Proof: If the master m interconnect state γ_m is POSTPONE, the `WAIT` signal is asserted by the SoC-bus. Thus, master m is stalled and the priority cannot change. If the bus is constantly used in pipelining mode, only the first bus access transfers the master priority π_m. The syscall is executed immediately after a context save; thus, the priority remains the same for the syscall. Afterwards, the kernel may change the task, including the priority, and then restores the context. The context restore requires instructions, which are not accessing the memory; thus, the pipelined access is stopped at least one time. Therefore, the priority of the new task is applied at the next address phase before returning to the user-mode. □

Theorem 2: The highest prioritized master in the requesting master set A_s for slave s immediately gets the permission to the slave s.
Proof: The requesting master set A_s for slave s contains all masters that are requesting access to the slave s. If the master m in the set A_s is the highest prioritized master, then $\alpha_s = m$ and the interconnect puts these signals through. Otherwise, the master $m\prime$, with $\alpha_s \neq m\prime$, changes its interconnect state $\gamma_{m\prime}$ to POSTPONE and remains there as long as $\alpha_s \neq m\prime$. □

Theorem 3: The arbitration is deadlock free.
Proof: The arbiter immediately allows access to the higher prioritized master; and therefore, the bus is a preemptive resource. Thus, not all four Coffman conditions [18] hold, which are required to lead into a deadlock. □

V. Evaluation

In this section, we are examine the implementation of our proposed arbitration approach. For that, we used the *mosart*MCU platform that is running in an FPGA. First, we will show a behavior example in a simulation and a use case scenario. Afterwards, we will investigate the slices utilization and the timing behavior of the synthesized hardware, which was done with the Xilinx Vivado 2017.3 WebPack toolchain for a Xilinx Artix-7 [19].

A. Behavior

This part of the evaluation shows three masters $M := \{m_1, m_2, m_3\}$ that want to access the slave $s_0 \in S$, concurrently. Thereby, the master priorities, which are recognized from their currently running tasks, are set as follows:

$$\pi_{m_2} > \pi_{m_3} > \pi_{m_1} \quad (3)$$

Fig. 5 depicts the access flow to the slave s_0 in a simulation output and shows the priority inversion avoidance on the SoC-bus. In the next list, the relevant times are emphasized:

- At time t_0, all three masters wanted to access the slave s_0. The address and control signals of the highest prioritized master (i.e., master m_2) are passed to the slave. For the other two masters the address and control signals are internally stored.

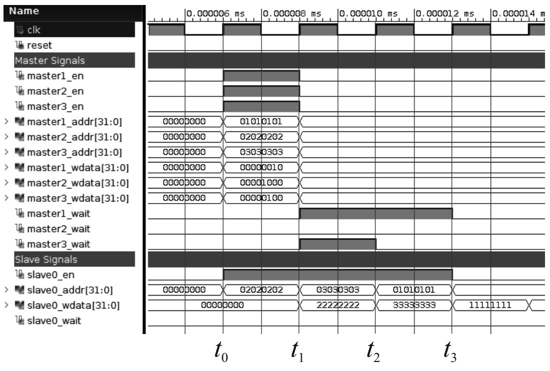


Fig. 5. Simulation example of three concurrently accessing masters with $\pi_{m_2} > \pi_{m_3} > \pi_{m_1}$ to the same slave s_0.

- At time t_1, the data phase of master m_2 starts and the address phase of master m_3 is caught up. For the two postponed masters (i.e., state POSTPONE) the `WAIT` signals are asserted.
- At time t_2, master m_3's data phase is performed and the master m_2 address phase is caught up.
- At time t_3, the data phase of the lowest prioritized master m_1 is performed.

B. Use Case

In this use case scenario, we investigate the measured WCET and the measured Worst Case Response Time (WCRT) of four tasks distributed among a dual-core system. Both cores are connected to the SoC-bus as a master and communicate to the shared global memory (i.e., $s \in S$). On masters m_0 and m_1 the tasks $T_{m_0} := \{\tau_{m_0,4}, \tau_{m_0,1}\}$ and $T_{m_1} := \{\tau_{m_1,3}, \tau_{m_1,2}\}$ are released on the same time, respectively. Thereby, the task identifier defines their priority and a higher number defines a higher priority. The priorities are distributed according to the RM scheduling policy; whereby, shorter deadlines possess a higher priority. Table I lists the WCETs and WCRTs for static allocated priorities to the masters (i.e., $\tilde{\pi}_{m_0} < \tilde{\pi}_{m_1}$) and the task priorities approach. Fig. 6 plots the execution flow of the four tasks in the task priority aware approach.

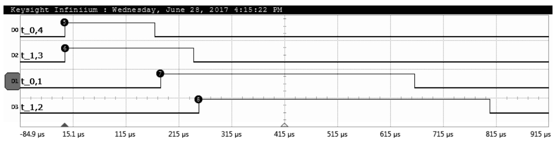


Fig. 6. Use case task executions.

TABLE I. USE CASE WCETS AND WCRTS.

Task	static priority ($\tilde{\pi}_{m_0} < \tilde{\pi}_{m_1}$)		task priority	
	WCET	WCRT	WCET	WCRT
$\tau_{m_0,4}$	155.20 µs	155.20 µs	145.30 µs	145.30 µs
$\tau_{m_1,3}$	240.32 µs	240.48 µs	242.34 µs	242.50 µs
$\tau_{m_0,1}$	440.34 µs	630.96 µs	439.68 µs	620.40 µs
$\tau_{m_1,2}$	560.32 µs	811.22 µs	560.32 µs	813.24 µs

In the static priority use case, master m_1 is higher prioritized than master m_0. Therefore, the highest prioritized task $\tau_{m_0,4}$ is being stalled while task $\tau_{m_1,3}$ is accessing the shared memory, concurrently. This leads to a priority inversion on the SoC-bus. However, in our proposed approach the priority inversion is avoided; and thus, the WCET and WCRT of the highest prioritized task $\tau_{m_0,4}$ are improved.

For the tasks $\tau_{m_1,2}$ and $\tau_{m_0,1}$ the WCETs remains almost equal, because the order of the priorities remains the same for both use cases. Whereas, for the task $\tau_{m_0,1}$ the WCRT is shorted, because task $\tau_{m_0,4}$ is not stalled by the SoC-bus.

C. Slices Utilization

For FPGA developers, the slices utilization is a metric to investigate the resource utilization in the FPGA. Thereby, in a FPGA the slices utilization can be distinguished into Flip-Flop (FF) and Lockup Table (LUT) slices. The FF slices are D-FFs, which are used to implement sequential logic. The LUT slices are used to implement combinational logic. The LUTs are tables, which map their input values to a defined output value. In Artix-7, the LUTs are implemented as $6\,\mathrm{Bit}$; thus, they save 64 output values for all 64 input combinations.

Fig. 7a shows the FF slices utilization depending on the number of masters and slaves. It shows the utilization with $32\,\mathrm{Bit}$ and $16\,\mathrm{Bit}$ wide priorities and with static assigned priorities. Fig. 7b depicts the same for LUT slices.

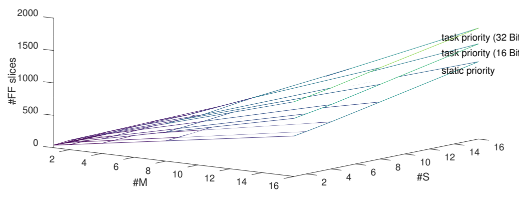
In both figures, the utilization increases with the number of increasing masters and slaves. Further, the figures show that the complexity of the implementation is $\mathcal{O}(\#M \cdot \#S)$, what reflects the complexity of a crossbar switch. If we compare the implementation of assigned static priority with the proposed task priority approach, we see that the latter requires more slices depending on the priority width. Nevertheless, the resource increase is in the same complexity class.

The cause of increasing FFs is the catch up logic of the address phase. The LUT slices are influenced by comparing the priorities of the masters.

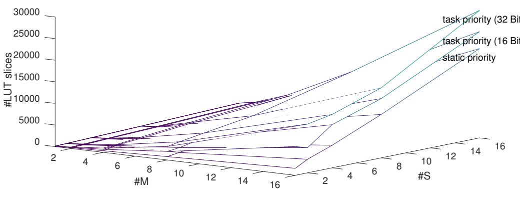
D. Timing Results

The last part of the evaluation covers the timing behavior depending on the number of masters and slaves. The timing behavior is significant to figure out the maximum achievable frequency that an implementation could run in a FPGA. If the maximum achievable frequency is exceeded, a setup time or hold time could be violated resulting into an unexpected run-time behavior. For the timing evaluation, we used the Xilinx Artix-7 with the lowest speed grade.

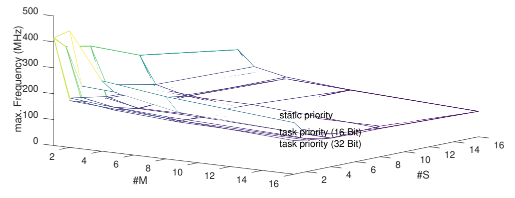
Fig. 7c depicts the maximum achievable frequency with different number of masters and slaves. The figure compares static priorities and our task priority aware approach. All configurations show a similar frequency behavior; whereby, increasing masters and slaves increase also the number of possible master to slave connections. Hence, more signals have to be switched through. The static allocated priority solution reaches a higher frequency than our task priority aware approach. However, with an increasing number of slaves both frequencies converge to the same maximum achievable frequency; thus, the frequency gap will be even smaller.



(a) FF slices utilization.



(b) LUT slices utilization.



(c) Maximal achievable frequency.

Fig. 7. Resource utilization and maximum achievable frequency depending on the numbers of masters and slaves of the SoC-bus.

VI. Conclusion

In this paper, we presented a new task priority aware SoC-bus for embedded systems. Thereby, we extended the basic SoC-bus protocol with priority awareness without adding new wires for the priority. This approach would work on all other bus protocols that distinguish between address phase and data phase, and where the address phase does not use the data signal (e.g., ARM AMBA 3 AHB-Lite). The basic protocol is not multi-master capable; therefore, we used an interconnect to isolate all masters. The arbitration policy is implemented into the interconnect, which is able to catch up the address phases if a master had to wait because its priority was lower than the one of the currently admitted master. This priority awareness always admits an access to a slave for the highest prioritized, namely the most important, master and avoids priority inversion on the SoC-bus layer. Thereby, we used the *mosart*MCU platform where the master's priority is equal to the currently running task in it. Further, in the evaluation we showed a simulation output, a use case, and the resource utilization in the FPGA. The results showed that the priority awareness in the SoC-bus consumes more slices and the maximum reachable frequency decreases; nevertheless, compared to the basic SoC-bus the priority awareness requirement is small and the use case showed that the whole system behaves according to the task priorities as required by real-time systems.

Acknowledgment

This research project was partially funded by AVL List GmbH and the Austrian Federal Ministry of Sciences, Research and Economy (bmwfw).

References

[1] International Organization for Standardization, "ISO/IEC 2382:2015 Information technology," International Organization for Standardization, Geneva, CH, Tech. Rep., 2015.

[2] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, jan 1973.

[3] A. S. Tanenbaum and T. Austin, *Structured Computer Organization*, 6th ed. Prentice Hall, 2009.

[4] *ARM AMBA 5 AHB Protocol Specification*, ARM Limited, 2015.

[5] *Avalon Interface Specifications*, Altera Corporation, 2015.

[6] *Wishbone B4 - WISHBONE System-on-Chip (SoC) Interconnection Architecturefor Portable IP Cores*, OpenCores, 2010.

[7] *AMBA Specification*, ARM Limited, 1999.

[8] *Processor Local Bus (PLB) Arbiter Design Specification*, Xilinx Inc., 2002.

[9] A. Shrivastava and S. K. Sharma, "Various Arbitration Algorithm for On-Chip(AMBA) Shared Bus Multi-Processor SoC," in *IEEE Students' Conf. on Electrical, Electronics and Computer Science*. IEEE, Mar. 2016.

[10] H. Shah, A. Raabe, and A. Knoll, "Priority division: A high-speed shared-memory bus arbitration with bounded latency," in *Proc. of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, Mar. 2011.

[11] R. C. Bhawna Tiwari and N. Goel, "Comparative Analysis of Different Lottery Bus Arbitration Techniques for SoC Communication," in *Proc. of the Int. Conference on Computational Techniques in Information and Communication Technologies (ICCTICT)*. IEEE, Mar. 2016.

[12] F. Mauroner and M. Baunach, "EventIRQ: An Event based and Priority aware IRQ handling for Multi-Tasking Environments," in *Proc. of the 20th Euromicro Conference on Digital System Design (DSD)*, Aug 2017, pp. 102–110.

[13] RISC-V Foundation, "RISC-V," https://riscv.org/. Last Visited: 01.12.2017.

[14] A. Waterman, Y. Lee, R. Avizienis, D. Patterson, and K. Asanovic, *The RISC-V Instruction Set Manual*, 2016.

[15] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Barua, "A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms," *Handbook on Scheduling Algorithms, Mehtods, and Models*, 2004.

[16] *AMBA 3 AHB-Lite Protocol*, ARM Limited, 2006.

[17] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. on Computers*, vol. 39, no. 9, pp. 1175–1185, Sep 1990.

[18] E. G. Coffman, M. J. Elphick, and A. Shoshani, "System Deadlocks," *ACM Trans. on Computing Surveys*, vol. 3, no. 2, pp. 67–78, 1971.

[19] Xilinx Inc., "Artix7," https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html.

E. Remote Instruction Call: An RPC approach on Instructions for Embedded Multi-Core Systems

Publication Information

F. Mauroner and M. Baunach. Remote Instruction Call: An RPC approach on Instructions for Embedded Multi-Core Systems. *In Proceedings of the 19th International Conference on Industrial Technology (ICIT)*. Lyon, France. February 2018.

Contribution

Main author

Remote Instruction Call: An RPC approach on Instructions for Embedded Multi-Core Systems

Fabian Mauroner and Marcel Baunach
Institute of Technical Informatics
Graz University of Technology, Graz, Austria
Email: {mauroner, baunach}@tugraz.at

***Abstract*—Remote Procedure Calls (RPCs) are a well-known approach on the software layer, but not on the hardware layer. In this paper, we present, to the best of our knowledge, the first Remote Instruction Call (RIC) approach, which supports the execution of an instruction in another Central Processing Unit (CPU). This leads to an instruction execution in a different address room. RIC can perform some operations that are otherwise only possible with a memory access into another CPU's address room. With RIC no synchronization is required, what may avoid performance penalties as in common approaches. In this work, we present the idea of RIC, an implementation into the *mosart*MCU project, and the performance evaluation, which is implemented in a Field Programmable Gate Array (FPGA). We discuss some potential improvements and extensions, which may make this work the base for future works.**

***Index Terms*—Embedded Systems; Multi-Core; Remote Instruction Call; FPGA implementation**

I. Introduction

Remote Procedure Call (RPC) is a well-known and widely used approach to execute a procedure on another machine. More than 30 years ago [1], there was established the idea of transferring the arguments and the identifier of the procedure to a remote machine where the procedure is executed. In the meantime, the caller is waiting for the return value, which is returned by the remote machine. RPCs enable the execution of procedures in a completely different address room. If a procedure uses some internal values, the procedure may return different values compared to a local execution. CORBA [2], DCOM [3] from Microsoft, or RMI [4] in Java are RPC specifications that are defining how the RPC execution works: One part is about how the arguments are passed; this can be binary, or marshaled. Another part specifies where and how the procedures are referenced on a remote machine and how the data is transferred.

RPCs are not limited in the execution on distributed machines; they can also be locally applied on a System on Chip (SoC). A SoC may contain many computation units (i.e., Central Processing Units (CPUs)), which may communicate between each other to exchange information or to perform work as RPCs. For embedded systems, which are often real-time systems, already exist real-time RPC approaches. Nevertheless, these approaches require computation time to encode, find the destination machine, transfer, and decode the arguments. For the return value, once again the same steps have to be performed. For constrained embedded systems, this sophisticated approach could be too complex and may even shorten their batteries lifetime. Therefore, on embedded multi-core systems, the common way to implement an efficient RPC is to use shared memory and a synchronization primitive as an event to notify another task. The shared memory requires a synchronization mechanism to avoid race conditions. However, for real-time systems, the synchronization mechanism is still an intense research topic, and results are either pessimistic or difficult to implement (i.e., [5]–[7]). Another approach to perform computation work on another CPU is to use an Inter-Process Communication (IPC). There exist approaches based on shared memory, which are leading to the same problems as mentioned before. However, there are also approaches that get along without shared memory; instead, a dedicated hardware performs the transfer. There exist co-processor (e.g., [8], [9]), Direct Memory Access (DMA) (e.g., [10]), and hardware module (e.g., [11], [12]) approaches to perform an IPC. However, these approaches either restrict the number of IPC channels or do not consider real-time requirements, what leads to deadline violations.

To counteract these problems, we present in this paper the Remote Instruction Call (RIC) approach. It works similar to RPCs on the software layer, but instead of procedures, instructions are remotely executed on the hardware layer. Thereby, the arguments are passed to the destination CPU, where the instruction should be executed. Then, the CPU returns the return value to the caller CPU.

The rest of the paper is organized as follows. Section II shows the specification of the, to the best of our knowledge, first approach that moves an RPC down into the hardware layer for performing an instruction in another CPU. Section III demonstrates an example implementation of RIC in a Field Programmable Gate Array (FPGA). The evaluation, in Section IV, shows the performance of the implementation and the synthesize results in an FPGA. Section V discusses RIC's issues and the capability in futures embedded multi-core systems. Section VI concludes this work.

II. Remote Instruction Call

The RIC approach is a technique that works similar to an RPC, but instead of a remote procedure, a remote instruction is executed. To enable that, a connection between the sender and the receiver, which is performing the computation work, must be possible. The arguments of the instruction and its return value are transmitted over the connection. For RIC, we are assuming a multi-core system with Reduced Instruction Set Computer (RISC) CPUs, whereby the instructions possess a 3-address format. This means, that the instructions may have up to two source registers (`src1` and `src2`) and up to one destination register (`dst`). The first source register `src1` has the restriction that it must contain an address. The address defines the destination CPU. This leads to the assumption, that there exists a unique address space, whereby each CPU is assigned to a specific address range in the unique address space. Further, the address contains the addressed Operating System (OS)'s construct in the remote CPU. On this OS construct, the remote instruction will perform the instruction's work, with the data of the possible second argument.

To execute a remote instruction, the common memory interface is used. Beside the control signals for starting a memory access (i.e., `en`) and the control signal for defining a write or read operation (i.e., `wen`), RIC requires an additional control signal, namely `type`. In the base version of the memory interface specification, it defines the data width; thus, it defines if the memory transfer is either a double word, word, half-word, or byte. We extended this control signal with the identification of the remote instruction. Beside the definition of the memory width, it defines the type of instruction that has to be performed remotely. Over the address (i.e., `addr`) and the write data signals (i.e., `wdata`), the first and second arguments of the instruction are transferred, respectively.

The interconnect fabric transfers the instruction type and the two arguments to the destination CPU as a common remote memory access. The memory access is forwarded by the interconnect fabric through the recognition of the address.

RIC builds on an extended *memory controller* that every CPU in the multi-core system possesses. The memory controller is responsible for forwarding the memory accesses from the CPU to the addressed component, according to the address. Thus, it could be addressed a local component or a remote component. Here, a component can be either a memory device or a peripheral. To support RIC, the memory controller detects a remote address and initiates a remote memory access that the interconnect fabric forwards to the destination CPU. The remote memory controller detects a remote instruction and forwards it to the CPU through the *remote interface*, depicted in Fig. 1.

The remote interface is based on a handshake protocol, similar to the AMBA AXI protocol [13], with a valid signal for each instruction and a ready signal indicating if a remote instruction can be performed. Additionally to the control signals, the remote interface possesses the `rem_src1_dst` port with the first argument or the return value of the response. The `rem_src2` port transmits the second argument. With the `rem_ID` port, the destination CPU is aware of the requesting CPU. If the instruction is specified with a return value, the requesting CPU waits for the return value while the remote instruction is executed.

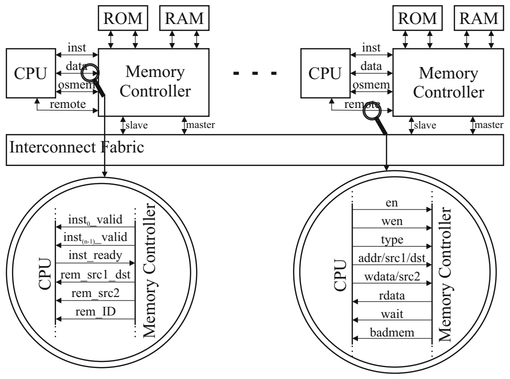


Fig. 1. Architecture structure for supporting the RIC approach.

III. Implementation

We implemented the proposed approach into the research platform *mosart*MCU. The *mosart*MCU is implemented into a Xilinx Artix-7 FPGA, which is assembled on a Nexys 4 DDR board from Digilent. In our project, we implemented two RIC instructions, one without a return value, and one with a return value. Before we explain the functionality of the RIC instructions, we introduce the *mosart*MCU and two local instructions, which are then extended to RICs.

A. mosartMCU

To realize the proposed architecture, we started the *mosart*MCU[1] project, that implements OS awareness into embedded multi-core systems (e.g., [14], [15]). The open RISC-V [16] Verilog implementation vScale[2] from the University of California, Berkeley, is the code base for the *mosart*MCU project. vScale implements all 32 Bit integers and the multiplication/division instructions from the Instruction Set Architecture (ISA) specification [17] and executes the instruction in a three stage pipeline. The specification defines 32 registers; whereas, the compiler does not use the register `tp`. This register indicates the Task Control Block (TCB) of the currently running task, which contains information about the task including its priority. We extended the basic implementation with an automatic read operation triggered by the hardware if the register `tp` is changed. Therefore, the hardware is always aware of the currently running task's priority. Concurrently to the normal execution, a read operation is automatically performed by using the additional connection (i.e., *osmem*) to the data memory through a dual-port memory. This dual-port memory is also used by some other OS-awareness extensions. Further, the CPU specification defines three different operating modes, whereas the *mosart*MCU supports only the non-privileged *user*-mode and the privileged *kernel*-mode. These operating modes define permissions for

[1]Multi-Core Operating-System-aware Real-Time MCU
[2]https://github.com/ucb-bar/vscale

some instructions and for accessing Control Status Registers (CSRs), which are hardware registers used to configure and to get information from the CPU.

To support predictable memory access times in a multi-core environment, we use an interconnect fabric with priority awareness [18]; there the arbitration of the memory accesses depends on the tasks' priorities. The underlying protocol operates in pipeline mode; thus, an address phase is performing while a data phase of the previous address phase is executing, as in the ARM's AMBA AHB-Lite bus [19] specification.

B. EventIRQ

EventIRQ [14] is a hardware extension of the basic *mosart*MCU. The EventIRQ is an Interrupt Request (IRQ) handling approach for avoiding the unpredictable interruptions due to IRQs. To achieve that, all the interrupts are mapped to OS events, which a task is waiting for. Therefore, the software for the Interrupt Service Routine (ISR) is moved from the ISR to a task. On an IRQ, the hardware extension accesses the TCB of the triggered task by knowing the internal OS data structures. All these operations are simultaneously performed to the normal execution flow. At the end of the TCB access, EventIRQ is aware of the currently running task's priority and the priority of the triggered task. Thus, the currently running task is only interrupted iff the priority of the triggered task is higher than the one of the currently running task. Otherwise, the task is appended to a list, which will be caught up by the OS later on. With this postponing of the IRQ handling, the response time of the IRQ may increase; however, the unpredictable interruption of a high-prioritized task is avoided and no rate-monotonic priority inversion (part of Operating System Priority Inversion (OS-PI)) occurs.

For all IRQs, expect the system timer, the tasks waiting for the event are ordered by priority. Thus, on a set event, the highest prioritized task consumes the event. However, for the system timer, all the tasks are ordered by its timeout. To handle the timeout queue properly and to avoid the OS-PI issue, the EventIRQ additionally sets the new system timeout of the next waiting task in hardware.

To avoid the OS-PI, caused by setting a software event that is directed to a lower prioritized task, EventIRQ extends the base ISA with a set software event instruction `sev src1`. This instruction performs the same operations as the mentioned process for an IRQ, but instead of updating the event table (i.e., interrupt vector table for the EventIRQ that is stored in data memory) it updates the triggered event.

C. EventQueue

The EventQueue approach is an IPC realization in the *mosart*MCU. EventQueue enables tasks, with the support of a hardware extension, to transfer data to another task. The base technologies of EventQueue are the before mentioned EventIRQ approach and the message queue mechanism.

For transferring data from one task to another, the *mosart*MCU ISA is extended with a queue write instruction `qwr dst,src1,src2`. The instruction triggers in the CPU a process for adding data into the queue. There, EventQueue checks the size in the queue's OS data structure and checks if the buffer is already full. This state is then passed into the destination register. Thus, the instruction is stalled as long as the return value is generated. Afterwards, EventQueue operates simultaneously to the current program flow, and the hardware may trigger the task that is waiting for some incoming data by setting the event. The setting of the event follows the EventIRQ mechanism and avoids the OS-PI issue as mentioned in Section III-B.

The instruction's arguments are the address of the referencing queue and the passing data. Consequently, the sender task does not accesses the memory of the queue (i.e., only referencing). EventQueue can also be used to implement a secure IPC with the combination of a Memory Protection Unit (MPU). There, the MPU protects the queue's OS data structure and EventQueue adds the data into the queue.

D. RIC extension

The mentioned EventIRQ and EventQueue approaches work only locally; thus, for single-core systems. To support these approaches for multi-core systems, we use RIC as proposed in this paper.

The CPU executes all its standard instructions as the base implementation does. However, if one of the two proposed extended instructions is detected by the instruction decoder, the instruction decoder checks if the instruction is addressed locally or to a remote CPU. In case that it is addressed to the local CPU, the instructions are executed as mentioned before. Otherwise, the CPU transfers the content of the two sources register to the local memory controller. Here it is important to emphasize, that for the transfer the `type` is set to apply the remote instruction. The memory controller detects that the memory access is addressed to a remote CPU and forwards the memory access to the interconnect fabric.

The interconnect fabric forwards the memory access to the remote CPU. Thereby the interconnect fabric considers the task's priorities and arbitrates the accesses according to their priorities. The remote CPU's memory controller detects an RIC, due to the control signal `type`, and forwards all the required information to the CPU over the remote interface.

In case of a set event instruction `sev`, the first argument represents the address of the event instance in the remote CPU. This triggers the EventIRQ mechanism, as mentioned in Section III-B. Here, the instruction does not use a return value; therefore, the sender CPU immediately resumes its work and the remote CPU executes, simultaneously to its usual work, the EventIRQ mechanism.

For the queue write instruction `qwr`, the first argument represents the address of the queue instance in the remote CPU and the second argument represents the data to transfer. Here, the execution of the queue write instruction works similar to the set event instruction `sev`, but for the return value, some additional work has to be performed. After receiving the queue write instruction, the CPU executes the instruction, what consumes some cycles. If the EventQueue produces the

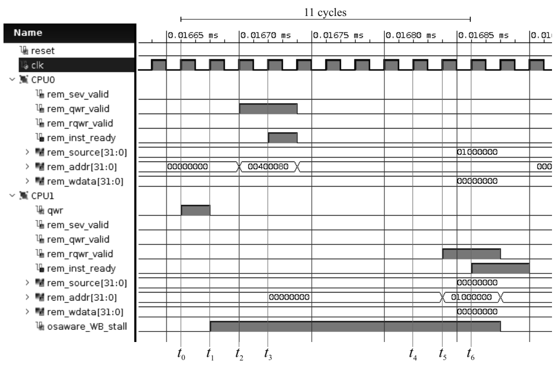


Fig. 2. Example of a remotely executing queue write instruction.

return value, EventQueue initiates a new memory access with a specific type. This memory access contains the return value, generated by EventQueue. The memory access is transferred as before, over the interconnect fabric to the CPU, what triggered a queue write instruction. The remote identifier `rem_ID` is used for the replay address. The initiating CPU stalls its pipeline as long as the return value is replayed by the destination CPU.

IV. Evaluation

In this section we show the performance evaluation of the remotely execution of a write queue instruction and the synthesize results in the Xilinx Artix-7 FPGA, which is reported by the Xilinx Vivado 2017.3 synthesizer.

A. Performance Evaluation

For the local queue write instructions `qwr`, the CPU stalls as long as the return value is not returned by the EventQueue. In addition to the instruction execution, for RICs the interconnect structure influences the execution time of the remote instruction. Fig. 2 depicts involved signals of a remotely executing write queue instruction on the *mosart*MCU. In the following, each marked time is discussed in detail:

- The CPU1 initiates, at time t_0, a queue write instruction `qwr`. The instruction is addressed to the remote CPU0.
- At time t_1, the memory controller detects that a RIC has to be performed and forwards the memory access to the interconnect. Furthermore, the CPU starts to stall until the return value is responded.
- The interconnect contains a register for forwarding the memory access, leading to one additional cycle for the interconnect. Therefore, at time t_2, the memory access signals are forwarded to the CPU0.
- At time t_3, CPU1 is ready to take over the two arguments and starts the EventQueue process.
- The EventQueue execution consumes 5 cycles; and at time t_4, the memory access for transferring the return value is initiated. The interconnect needs an additional cycle to forward the memory access to CPU1, and at time t_5, the signals are asserted.
- At time t_6, the return value is received and CPU1 resumes its execution on the next cycle.

On the *mosart*MCU platform, a remote queue write operation stalls the CPU for 10 cycles. Hence, the execution of the write queue instruction `qwr` for a remote CPU consumes 11 cycles. On the destination CPU, the EventQueue is executed simultaneously to its normal execution flow. Thus, the CPU is not blocked through another CPU and it will notify the waiting task about new data in the queue according to the OS-PI avoidance policy of EventIRQ.

TABLE I
Synthesize results with and without RIC support.

	FF	LUT	max Frequency	Power
CPU with RIC	2773	4130	69.26 MHz	19 mW
CPU without RIC	2701	4044	77.32 MHz	21 mW
Mem. Contr. with RIC	129	376	323.33 MHz	1 mW
Mem. Contr. without RIC	117	307	323.33 MHz	1 mW

B. Synthesize Results

In this section, we investigate the synthesize results of involved hardware modules with and without RIC support. Thereby, we considered the number of Flip-Flops (FFs), Look Up Tables (LUTs), the maximal achievable frequency, and the dynamic power consumption.

The FFs are D-FFs, which are used to implement sequential logic; by contrast, LUTs are used for implementing combinational logic. The LUTs are tables, which map their input values to a defined output value. The Xilinx Artix-7 architecture possesses a 6 Bit LUT; thus, it saves 64 output values for all 64 input combinations. The maximal achievable frequency is another metric of an FPGA. This metric is influenced by the longest path that a signal has to pass through the combinational logic. If the maximum achievable frequency is violated, an unexpected run-time behavior is the result. This is because a setup time or hold time is violated. We investigated this metric with the lowest speed grade of the Artix-7 FPGA. The last metric is the dynamic power consumption that the modules consume by changing their internal states. Table I lists all mentioned metrics for the CPU and for the memory controller. For the CPU and the memory controller the FFs and LUTs slightly increase with RIC support. The FFs are required for additional information that must be stored internally, and the LUTs for feeding the additional FFs with information.

The maximal achievable frequency of the CPU is reduced with the support of RIC because the instruction decode is executed together with the instruction execution, which starts the memory transfer. For supporting RICs, we had to extend the instruction decode, which had already the longest path in the CPU. Thus, an additional pipeline stage might mitigate the frequency reduction impact of the RIC extension. For the memory controller the maximal achievable frequency stays constant.

The power consumption depends on the required FFs and LUTs; thus, the power consumption of the CPU will also increase with RIC support. The additional required resources for the memory controller are that little, that the power consumption increase is less than the report output accuracy.

V. Discussion

In traditional multi-core systems, the execution of work with remote information results in remote memory accesses or usage of shared memory. However, these accesses must be synchronized with resource management protocols, which introduces management overhead and are often very pessimistic. Other techniques in multi-core systems as notifying a task of another CPU is mostly done with an IRQ. However, an IRQ introduces issues as the OS-PI problem. RIC triggers the execution of an instruction on a remote CPU, similar to the RPC for procedures at software layer. Thus, the CPU locally has the potential to handle local issues while performing a remote instruction; thus, no global synchronization is required.

The bus arbitration for the remote CPU is done by the interconnect fabric. In the *mosart*MCU this is realized with a crossbar switch that uses the task priorities of each accessing task for the arbitration. RIC is not limited to a crossbar switch; it can also be implemented by using a Network on Chip (NoC) or other bus topologies. The only requirement is that the interconnect fabric must be able to transfer the instruction type and that the remote CPU is aware of the sending CPU, for replaying the return value (e.g., for the `qwr`).

RIC has also the potential to be implemented more deeply into the CPU. One extension would be to execute the remote instruction in an out-of-order execution manner as in High-Performance Computing (HPC) systems or to share execution resources of the CPU as in Intel's hyper threading [20] approach.

VI. Conclusion

In this paper, we presented, to the best of our knowledge, the first approach to perform an RIC, which enables the execution of instructions in a remote CPU. The idea is similar to an RPC; however, instead of operating on software layer, RIC works on instructions; thus, on the hardware layer. Today's 3-address RISC architectures suite well to transfer the two source register over the common data bus to the remote CPU, where the instruction is then performed. This enables the execution of instructions in a completely different address room and avoids the demand of using shared memory concepts that must be synchronized. We implemented RIC into our research platform *mosart*MCU and investigated the performance of a remotely executing instruction and the synthesize results in an FPGA implementation. RIC needs more cycles, because of the transfer over the interconnect. Nevertheless, if the interconnect fabric and the instruction itself are predictable, RIC is performed in a predictable time. For real-time systems the predictable computation time is intended to get accurate real-time analyzes. Further, we discussed the benefits of RIC and some potential future works that may base on the proposed RIC approach.

Acknowledgment

This research project was partially funded by AVL List GmbH and the Austrian Federal Ministry of Sciences, Research and Economy (bmwfw).

References

[1] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. on Compututer Systems*, vol. 2, no. 1, pp. 39–59, Feb. 1984.
[2] *Common Object Request Broker Architecture (CORBA) Specification, Version 3.3*, Object Management Group, Nov. 2012.
[3] R. Sessions, *COM and DCOM: Microsoft's Vision for Distributed Objects*. New York, NY, USA: John Wiley & Sons, Inc., 1998.
[4] J. Waldo, "Remote procedure calls and Java Remote Method Invocation," *IEEE Trans. on Concurrency*, vol. 6, no. 3, pp. 5–7, Jul 1998.
[5] R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-time Synchronization Protocols for Multiprocessors," in *Proc. of the 9th IEEE Real-Time Systems Symposium (RTSS)*. IEEE, Dec 1988, pp. 259–269.
[6] P. Gai, M. D. Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca, "A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform," in *Proc. of the 9th IEEE Proc. of the Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, May 2003, pp. 189–198.
[7] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson, "A Flexible Real-Time Locking Protocol for Multiprocessors," in *Proc. of the 13th IEEE Int. Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, Aug 2007, pp. 47–56.
[8] U. Ramachandran, M. Solomon, and M. Vernon, "Hardware Support for Interprocess Communication," in *Proc. of the 14th Annual Int. Symposium on Computer Architecture (ISCA)*, ser. ISCA '87. ACM, 1987, pp. 178–188.
[9] J.-M. Hsu and P. Banerjee, "A Message Passing Coprocessor for Distributed Memory Multicomputers," in *Proc. of the 1990 ACM/IEEE Conference on Supercomputing (SC)*, ser. Supercomputing '90. IEEE Computer Society Press, 1990, pp. 720–729.
[10] S. Srinivasan and D. B. Stewart, "High Speed Hardware-assisted Real-time Interprocess Communication for Embedded Microcontrollers," in *Proc. of the 21st IEEE Conference on Real-time Systems Symposium (RTSS)*, ser. RTSS'10. IEEE Computer Society, 2000, pp. 269–279.
[11] *PrimeCell Inter-Processor Communications Module (PL320)*, ARM Limited, 2004.
[12] K. Johnson, "QorIQ Platform: Architecture Advantages," Presentation, Oct. 2013.
[13] *AMBA AXI and ACE Protocol Specification*, ARM Limited, 2011.
[14] F. Mauroner and M. Baunach, "EventIRQ: An Event based and Priority aware IRQ handling for Multi-Tasking Environments," in *Proc. of the 20th Euromicro Conference on Digital System Design (DSD)*, Aug 2017, pp. 102–110.
[15] ——, "StackMMU: Dynamic Stack Sharing for Embedded Systems," in *Proc. of the 22nd IEEE Int. Conference on Emerging Technologies and Factory Automation (ETFA)*, Sept 2017.
[16] RISC-V Foundation, "RISC-V," https://riscv.org/. Last Visited: 01.12.2017.
[17] A. Waterman, Y. Lee, R. Avizienis, D. Patterson, and K. Asanovic, *The RISC-V Instruction Set Manual*, 2016.
[18] F. Mauroner and M. Baunach, "Task Priority aware SoC-Bus for Embedded Systems," in *Proc. of the 19th International Conference on Industrial Technology (ICIT)*, Feb. 2018.
[19] *ARM AMBA 5 AHB Protocol Specification*, ARM Limited, 2015.
[20] D. Koufaty and D. T. Marr, "Hyperthreading Technology in the Netburst Microarchitecture," *IEEE Trans. on Micro*, vol. 23, no. 2, pp. 56–65, March 2003.

F. CoStack: Collaborative Stack Sharing for Embedded Real-Time Systems

Publication Information

F. Mauroner and M. Baunach. CoStack: Collaborative Stack Sharing for Embedded Real-Time Systems. *In Proceedings of the 13th International Conference on Systems (ICONS)*. Athens, Greece. April 2018.

Contribution

Main author

ICONS 2018 : The Thirteenth International Conference on Systems

CoStack: Collaborative Stack Sharing for Real-Time Embedded Systems

Fabian Mauroner
Institute of Technical Informatics
Graz University of Technology
Graz, Austria
Email: mauroner@tugraz.at

Marcel Baunach
Institute of Technical Informatics
Graz University of Technology
Graz, Austria
Email: baunach@tugraz.at

***Abstract*—Embedded real-time systems are targeting for economical stack memory usage and predictable execution flows, what is challenging to unify. In this paper, we propose CoStack, a collaborative stack sharing approach across tasks. CoStack allows defining a collaborative stack memory that can be used by a higher prioritized task if the stack runs out-of-memory. Thus, CoStack virtually reduces the stack memory consumption, leading to a lower memory requirement, and concurrently remains predictable, what is desired for real-time systems. This paper presents an experimental evaluation of CoStack, the synthesized results in a Field Programmable Gate Array (FPGA) and some implementation details of CoStack.**

Keywords–Embedded Systems; Stack handling; Operating-System-Awareness; FPGA implementation

I. Introduction

Modern real-time embedded systems require, due to their growing complexity and flexibility, evermore memory to fulfill all their challenging requirements. However, embedded system's memory is a restricted resource; thereby, it has to be used in an efficient way.

Global variables are always available for the complete embedded system's life, but local variables are only available and allocated on demand. Therefore, local variables are dynamically allocated on a specific space in the memory, named *stack frame*. The *stack pointer*, indicating the threshold of valid and non-valid data in the stack frame, grows if a local variable is allocated or if Central Processing Unit (CPU) registers are temporally stored (e.g., on function prologue and epilogue). The stack pointer shrinks if the local variables or temporally stored registers are not needed anymore.

In state-of-the-art real-time Operating Systems (OSs) [1][2] for each task an own individual stack frame is allocated; although, it is not fully utilized simultaneously. Therefore, [3][4] show approaches to use a common shared stack frame among all tasks. This reduces the overall stack memory consumption, but restricts the schedulability of all tasks. The reason therefor is that a stack cannot grow if the task's stack pointer is not on the top of the common stack frame. Otherwise, it would destroy data from another task in the common stack frame. Therefore, an individual stack frame is assigned to the tasks in real-time systems by accepting an increasing overall memory consumption. The reason therefor is, for real-time systems, the schedulability and satisfaction of real-time constraints is an essential requirement. This has to be improved to reduce the required computation power and CPU frequency, to reduce the costs and power of the developed embedded real-time system.

Consequently, both, efficient stack memory and schedulability must be unified to improve the memory consumption and to fulfill all the real-time requirements.

Observations showed [5] that not each task fully utilizes its individual stack frame simultaneously, wherefore the approach to share stack memory space among all tasks is used. To avoid the restriction of the schedulability, it must be avoided or at least timely bound that a task is blocked for requesting stack memory. That is possible with the concept of address virtualization [6]. However, this concept requires an additional hardware component, namely the Memory Management Unit (MMU). It translates all Virtual Memory (VM) addresses into Physical Memory (PM) addresses. However, an MMU is only rarely found in embedded systems, because it requires a lot of power and it introduces non-predictable memory accesses. That has to be avoided for real-time systems, which aim for predictability. Therefore, in [7] we presented a dynamic stack sharing approach, which uses VM addresses only for the stack memory and ensures a predictable stack memory access and stack pointer adjustment if the underlying memory architecture behaves predictable, too.

Since the memory is a scarce resource, the software developer has to use it sparsely. However, sometimes it is not possible to optimize the code (e.g., to use an algorithm with less stack memory consumption). Thus, to avoid an out-of-stack, the software developer must guarantee enough stack memory spaces for all tasks at any time. Otherwise, a task would be unpredictably blocked and deadlines may be violated.

A. *Related work*

In the recent years, there has been done a lot of research to reduce the stack memory consumption, with completely different approaches:

Wang *et al.* proposed the preemptive threshold scheduling in [8], which is used in the ThreadX real-time OS [9]. It is based on Rate Monotonic (RM) scheduling and extends each task with a threshold priority, beyond its nominal priority. If a task is scheduled, its threshold priority is the new priority that must be exceeded to preempt the task by another task's nominal priority. This solution leads to non-preemptive task groups and these are able to use a common shared stack frame. Further, it may improve the schedulability compared to the standard RM scheduling. Nevertheless, sharing the stack memory is not possible between non-preemptive task groups and/or preemptive tasks; thus, sharing the stack memory is limited.

In [10], Chu *et al.* proposed to use a binary translation and a specific kernel by using VM addressed stacks in embedded systems. Thus, their approach allocates only that memory that is required, but the authors showed that for each stack pointer access and adaptation the required run-time is enormous.

Yi *et al.* [11] showed an approach, where a tool analyzes the stack consumption of each task and modifies the developed code on compile time. The modified code calls the *on demand stack* library at each prologue and epilogue of a function. In their use case scenario, the stack memory consumption indeed reduces; however, the execution time increases by $10\,\%$.

Other solutions proposed to allocate the stack memory on the heap. In older works, as for instance [12], on each function prologue and epilogue the heap allocation and deallocation is called, respectively. However, these calls lead to a large run-time overhead for each function.

Works as [13][14] also allocate the stack on the heap. With analyses at compile time, they are able to reduce the large run-time overhead for allocating and deallocating stack memory on the heap. Nevertheless, run-time checks are still required to allocate and deallocate the stack memory.

Middha *et al.* [5] propose to allocate an individual stack frame to each task. If a task overflows (i.e., out-of-stack) its individual stack frame, their approach allocates unused stack memory in another task's individual stack frame. Without code optimization, their run-time consumption increases by around $23\,\%$; with an optimization about $3\,\%$. However, the optimized solution restricts programming features as function pointers and recursive functions.

None of the mentioned works is able to free stack memory voluntarily for a higher prioritized task if no stack memory is available. In [15], Baunach proposed CoMem that is a collaborative memory management in the heap memory for dynamic memory. There, each memory block (in the heap) is handled as a system resource. If a task requests the memory block and is used by another lower prioritized task, the lower prioritized task will be informed. That task has then the control to free the memory block or not.

CoMem enables a collaborative usage of memory blocks in the heap but not for the stack memory. Therefore, in this paper we present *CoStack*, a collaborative stack sharing concept based on a hardware extension, which enables the reduction of the whole stack memory consumption by defining parts in the source code in which the stack memory is collaborative. With the state-of-the-art approaches, there is no possibility to give a higher prioritized task the advantage to use the stack memory instead of a lower prioritized task that owns collaborative stack memory that might voluntarily be released. CoStack ensures the allocation of the required stack memory for higher prioritized tasks by freeing collaborative stack memory from lower prioritized tasks if an out-of-stack condition would result. CoStack does not require a code analysis at compile time; therefore, it is possible to use our approach also in highly dynamic environments, as the Internet of Things (IoT), Industry 4.0, or automotive applications.

The rest of the paper is organized as follows: First, Section II describes in detail the fundamentals of CoStack. Second, Section III analyses the memory improvement and shows the schedulability analysis. Next, Section IV demonstrates implementation aspects in our development platform. Section V shows the CoStack evaluation with an example and the synthesized results for a Field Programmable Gate Array (FPGA). Last, we summarize this work in Section VI.

II. Collaborative Stack Sharing

The collaborative stack sharing approach is based on StackMMU presented in [7]. First, we introduce the terminology and the system assumption. Second, we introduce the fundamentals of StackMMU. Third, we show the extension of the basic StackMMU, which is needed for providing the required information to the hardware. Last, the collaborative stack sharing approach, CoStack, is described in detail.

A. Terminology and Assumption

For CoStack we assume a Reduced Instruction Set Computer (RISC) load/store single-core CPU with Control Status Registers (CSRs), which are CPU registers accessible by specific instructions. Further, we define a multi-tasking system with $\tau \in T$ as a task in the set of tasks T. For each task τ we define, the priority p_τ, the Worst Case Execution Time (WCET) with C_τ defining the time executing on the CPU, and the period or minimum interarrival time with T_τ for periodic or sporadic tasks, respectively. The longest time, in which a priority inversion [16] occurs (i.e., a lower prioritized task runs instead of a higher prioritized task) is denoted as the blocking time B_τ of task τ. Further, we assume that the context switch in the OS and the OS itself consumes no computation time. The stack usage $\sigma_\tau(t) \in \mathbb{N}_0$ defines the stack memory consumption of task τ at time $t \in \mathbb{N}_0$, what can be analyzed with static code analyzers.

B. StackMMU

The StackMMU [7] uses VM addresses for the stack memory. Thus, each task's stack pointer points to a virtual address and StackMMU translates that address to a PM address. For that, the memory is divided into a common area and into a stack area, as depicted in Figure 1. In the common area,

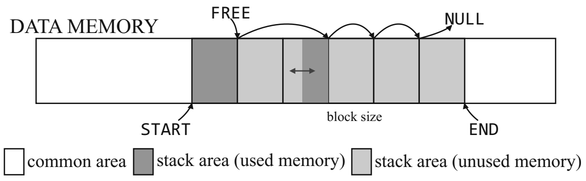


Figure 1. Memory layout of the StackMMU.

all the global data of a task is stored and accessed by PM addresses. The memory for the stack is located, in the stack area. Thereby, the stack area is again divided into blocks, called *pages*. Each page has the same size and is configurable including the start and end of the whole stack area at startup. A linked list contains all available pages, whereby the `FREE` register points to the first free available page. Thus, if a task requires more stack memory and exceeds the available memory in the last appended page, the StackMMU assigns a new page to that task. Thereby, the hardware uses the register `FREE` and updates the Task Control Block (TCB) of the task with the address of the new allocated page. On a stack memory access, first, the hardware reads the PM base address in the

TCB; and second, it accesses the content in the stack area. If the stack page is not required anymore, the hardware frees the page and updates all the required pointers (i.e., FREE and the pointer to the next free page in the page). Before a specific stack operation (i.e., growth, access, or shrinkage) is executed, all three operations are performing a read memory operation leading to an additional memory access. However, if the memory access is deterministic, as in many embedded systems, the whole stack operation is also executed in a predictable time as desired for real-time systems.

C. MultiStackMMU

StackMMU restricts the size of the stack pointer change for growing and shrinking in one instruction to the size of a stack page. That limitation is handled by a modified compiler. However, we purged that limitation with MultiStackMMU.

Here, if a stack grows or shrinks more than one page, the hardware extension performs the assignment or deassignment of pages one after another, respectively. This leads to a longer run-time to perform these operations; nevertheless, the execution time remains predictable if the memory accesses itself are deterministic.

Through this extension, the modified compiler, which limited the stack pointer change to one stack page size, is not required anymore. Besides, the hardware is now aware of the number of required pages on a stack memory request, necessary for performing our collaborative stack sharing approach CoStack.

D. CoStack

As long as in the stack area are available more unused pages than required by a task, the pages are properly assigned to the task by the MultiStackMMU approach. However, if there are not enough available pages, an out-of-stack condition occurs. An out-of-stack condition is handled by the OS; nevertheless, the task will be unpredictably blocked until stack memory is available. This would lead to a reduction of the system performance and may lead to real-time constraint violations. Thus, to reduce the possibility of an out-of-stack condition, enough stack memory must be provided.

Instead of blocking the task as long as not enough stack memory is available, CoStack deallocates stack memory from tasks that define their used stack memory as collaborative.

If CoStack detects that the required stack memory pages exceed the free available pages, a *collaborate exception* is triggered and the OS must handle it. Thereby, the stack growth is aborted by the hardware and the OS saves the context of the task. After rescheduling that task, the stack growth instruction will be repeated (i.e., the program counter of the stack growth instruction is stored in the TCB). In the exception handler, the OS searches a lower prioritized task that provides collaborative stack memory. Then, the OS modifies the program counter and schedules the collaborative task. The collaborative task frees the collaborative stack memory and yields itself to return to the OS. There, the scheduler selects the highest prioritized runnable task, which may be the same as the previous one. If a task, that requires stack memory, is scheduled and enough unused pages are available, the required pages are assigned to the task. Otherwise, once again a collaborate exception is triggered and the OS iterates through the tasks as before to search a collaborative task. In case that there are no lower prioritized tasks with collaborative stack memory, the OS has to define a handling strategy to handle this failure as in standard out-of-stack approaches.

Figure 2c demonstrates how a code part can be tagged with collaborative stack. The macros in Figure 2a, Figure 2b, and Figure 2d generate the code for handling CoStack properly. Additionally to the tagged code, which uses a collaborative stack memory, a handler is defined. The handler is only executed if the collaborative stack memory was deallocated for performing some clean-up work, similar to the catch primitive in programming languages such as Java or C++.

Figure 2a shows how the collaborative stack mechanism is performed. First, it checks if the currently running task already defines a collaborative code part. If so, the collaborative code part is already a part of an upper tagged collaborative code part and the `try` part is immediately executed. Otherwise, the handler address and the collaborate frame pointer are stored in the task's TCB. Additionally, the callee saved registers are stored on the stack, to restore them if the collaboration must be performed. Afterwards, the `try` part is executed. If there was no other task requiring the collaborative stack, the callee saved registers on the stack are discarded because the program flow was not manipulated. Otherwise, the program counter continues at the label COLLABORATE after a context switch. There, the callee saved registers are restored and the collaborative stack memory space is freed, by using the stored collaborate frame pointer. After that, the collaborate frame pointer in the TCB is cleared and the syscall `yield()` is called for a self-preemption to allow the OS to schedule a higher prioritized task that may wait for stack memory.

III. Analysis

In this section, we analyze the memory utilization of the collaborative stack sharing approach CoStack and compare it with the StackMMU approach. Further, we show the schedulability analysis of CoStack.

A. Memory Consumption

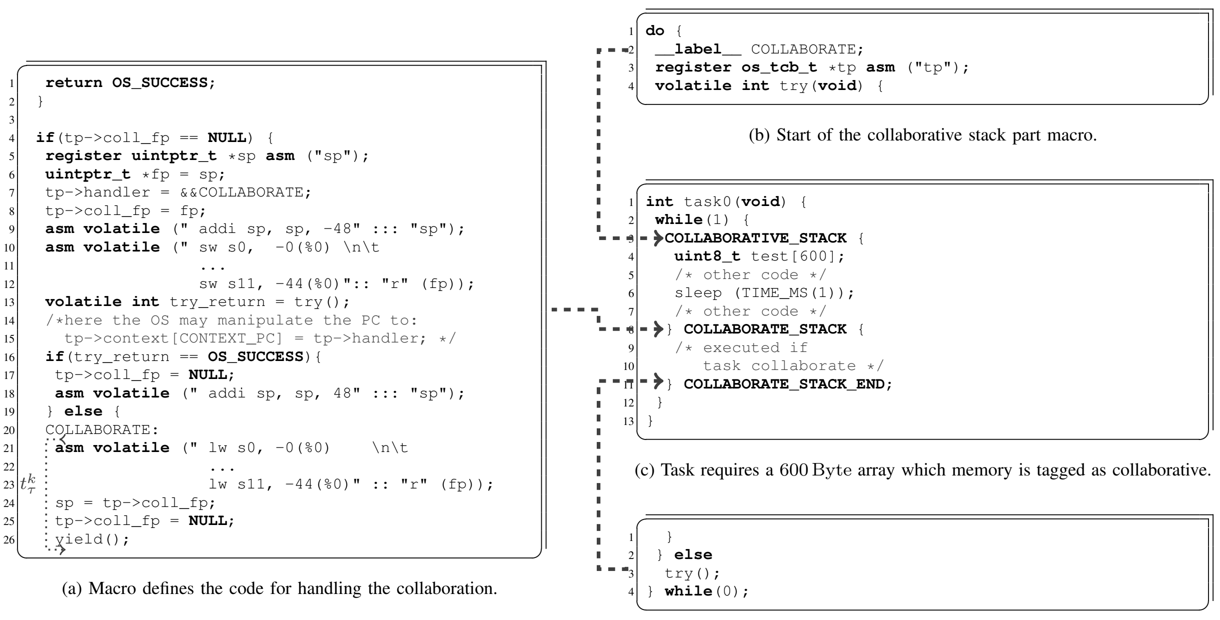
In the StackMMU approach, the stack grows and shrinks with the stack page size page_size. There, the size of the stack changes over time t, depending on the required memory $\sigma_\tau(t)$ by a task τ at time t. Thus, the stack memory consumption of the whole system U is calculated as follows:

$$U(t, \texttt{page\_size}) = \sum_{\tau \in T} \left\lceil \frac{\sigma_\tau(t)}{\texttt{page\_size}} \right\rceil \cdot \texttt{page\_size} \tag{1}$$

Let S denote the totally assigned stack space. To avoid an out-of-stack condition, the stack memory consumption U is not allowed to exceed the totally assigned stack memory S. Otherwise a task would be unpredictably blocked what restricts the schedulability:

$$\forall t \in \mathbb{N}_0 : \quad U(t, \texttt{page\_size}) \leq S \tag{2}$$

With the introduction of the collaborative stack sharing approach, each task τ may define some collaborative stack memory $\kappa_\tau(t)$ at time $t \in \mathbb{N}_0$. This collaborative stack memory κ is available for all higher prioritized tasks, leading



```
  return OS_SUCCESS;
 }

 if(tp->coll_fp == NULL) {
  register uintptr_t *sp asm ("sp");
  uintptr_t *fp = sp;
  tp->handler = &&COLLABORATE;
  tp->coll_fp = fp;
  asm volatile (" addi sp, sp, -48" ::: "sp");
  asm volatile (" sw s0,  -0(%0) \n\t
                  ...
                  sw s11, -44(%0)":: "r" (fp));
  volatile int try_return = try();
  /*here the OS may manipulate the PC to:
    tp->context[CONTEXT_PC] = tp->handler; */
  if(try_return == OS_SUCCESS){
   tp->coll_fp = NULL;
   asm volatile (" addi sp, sp, 48" ::: "sp");
  } else {
  COLLABORATE:
   asm volatile (" lw s0, -0(%0)    \n\t
                  ...
                  lw s11, -44(%0)" :: "r" (fp));
   sp = tp->coll_fp;
   tp->coll_fp = NULL;
   yield();
```

(a) Macro defines the code for handling the collaboration.

```
do {
 __label__ COLLABORATE;
 register os_tcb_t *tp asm ("tp");
 volatile int try(void) {
```

(b) Start of the collaborative stack part macro.

```
int task0(void) {
 while(1) {
  COLLABORATIVE_STACK {
   uint8_t test[600];
   /* other code */
   sleep (TIME_MS(1));
   /* other code */
  } COLLABORATE_STACK {
   /* executed if
      task collaborate */
  } COLLABORATE_STACK_END;
 }
}
```

(c) Task requires a 600 Byte array which memory is tagged as collaborative.

```
  }
 } else
  try();
} while(0);
```

(d) End of the collaborative stack part.

Figure 2. Macros for the collaborative stack sharing approach with a usage example.

to the available collaborative stack pages K_τ at time t for task τ.

$$K_\tau(t) = \sum_{\forall i: p_{\tau_i} < p_\tau} \left\lfloor \frac{\kappa_{\tau_i}(t)}{\texttt{page\_size}} \right\rfloor \cdot \texttt{page\_size} \quad (3)$$

As mentioned in Section II, if a higher prioritized task requires non-available stack memory but collaborative stack memory is available, the collaborative task deallocates its collaborative stack memory to make it available for the higher prioritized task. This means that the stack memory consumption is virtually reduced, leading to the next equation that must be hold to avoid an out-of-stack condition:

$$\forall t \in \mathbb{N}_0, \forall \tau \in T: \quad U(t, \texttt{page\_size}) - K_\tau(t) \le S \quad (4)$$

Thus, CoStack contributes to the reduction of the totally assigned stack memory S by collaboratively sharing stack memory as shown by comparing the equation (2) with (4).

B. Schedulability Analysis

The freeing of the collaborative stack memory is performed in the respective lower prioritized tasks. Thus, a priority inversion occurs: The lower prioritized task frees the stack memory and blocks the higher prioritized task because the required stack memory is not available. Thus, we are investigating the maximum blocking time B_τ of task τ for the RM schedulability analysis [16]:

$$\frac{C_{\tau_0}}{T_{\tau_0}} + ... + \frac{C_{\tau_{n-1}}}{T_{\tau n-1}} + \max\left(\frac{B_{\tau_0}}{T_{\tau_0}}, ..., \frac{B_{\tau_{n-1}}}{T_{\tau_{n-1}}}\right) \le n(2^{\frac{1}{n}} - 1) \quad (5)$$

In CoStack, the blocking time compounds on some administrative work and the freeing of the stack memory. Thereby, the time for freeing the stack memory is not constant, because MultiStackMMU is based on pages, which must be released one after another. Therefore, we are defining the collaborate time $t^k_{\tau\prime}(t)$ of task $\tau\prime$ at time t (see Figure 2a), which defines the time required by the collaborative task $\tau\prime$ to perform all the operations to free $\tau\prime$'s collaborative stack memory. Consequently, the blocking time B_τ of task τ can be calculated as follows:

$$B_\tau := \sum_{\forall i: p_{\tau_i} < p_\tau} \max_{\forall t \in \mathbb{N}} \{t^k_{\tau_i}(t)\} \quad (6)$$

The blocking time B_τ is the sum of the longest collaborate time $t^k_{\tau_i}$ of all lower prioritized tasks τ_i. Thus, the blocking time highly depends on the requested stack memory, the collaborative stack memory of each collaborative task, the number of lower prioritized collaborative tasks, and the time when the stack memory is requested.

IV. Implementation Details

We implemented the collaborative stack sharing approach into our *mosart*MCU research platform, running with the *mosart*MCU-OS.

A. mosartMCU

The *mosart*MCU (i.e., Multi-Core Operating-System-Aware Real-Time MCU) project implements OS awareness into embedded multi-core systems. The open RISC-V [17] architecture, maintained by the University of California of Berkeley, is the specification of the softcore *mosart*MCU. The *mosart*MCU is based on the offered open source Verilog implementation vScale. vScale implements all the 32 Bit integers and the multiplication/division instructions [18] and executes the instruction in a three stage pipeline. The specification specifies 32 registers whereas the compiler does not use the register `tp`. That register indicates the TCB, which contains information about the task including its priority, of the currently running task. We extended the basic implementation with an automatic read operation, triggered by the hardware if the register `tp` is changed. Therefore, the hardware is always aware of the currently running task's priority. In parallel to the normal execution, a read operation is automatically performed by using an additional connection to the data memory through a dual-port memory. This dual-port memory is also used by some other extensions (e.g., [19]). Further, the CPU specification defines three different operating modes, whereas the *mosart*MCU supports only the non-privileged *user*-mode and the privileged *kernel*-mode. These operating modes define permissions for some instructions and for accessing CSRs, which are hardware registers used to configure and to get information from the Microcontroller Unit (MCU).

B. mosartMCU-OS

The *mosart*MCU-OS is a real-time OS supporting the OS-awareness extension of the *mosart*MCU. Besides other OS-awareness concepts, it only has to initialize some CSRs (i.e., stack area) and the references of the next free available pages at startup, for supporting MultiStackMMU. After that, the MultiStackMMU operates transparent to the OS. However, the OS must be extended to support our proposed collaborative stack sharing approach.

C. Collaborative Stack Management

In a CSR, the hardware provides the number of required stack pages after a collaborate exception. The exception handler stores the number of required pages into the TCB of the currently running task. After executing the exception handler, the OS does its management work and selects a task according to the RM scheduling policy. Before leaving the OS, by restoring all the task's registers, the OS checks if the scheduled task requires more pages than available. The number of available pages is provided by the hardware through another CSR register. If there are more pages available than required, the OS lefts and resumes with the scheduled task. Otherwise, the OS searches, starting by the lowest prioritized task, a collaborative task that is recognized by the information in the TCB. If a collaborative task is found, the scheduler selects that task for executing. Thereby, the scheduler changes the program counter to continue at the collaborate handler, also stored in the TCB. The collaborate code restores its callee saved registers, frees the collaborate stack memory, and yields itself to return to the OS.

V. Experimental Evaluation

We experimentally evaluated the collaborative stack sharing approach in the *mosart*MCU, running with 50 MHz in a Xilinx Artix-7 FPGA. First, we demonstrate the cooperative stack sharing, measured with an oscilloscope; and second, we show the synthesized results for the FPGA.

A. CoStack Evaluation

This evaluation illustrates CoStack on a collaborative 600 Byte stack memory provided by task $\tau\prime$, once the stack memory is not available for the higher prioritized task τ. Figure 3 depicts the execution flow with the signal *os* showing

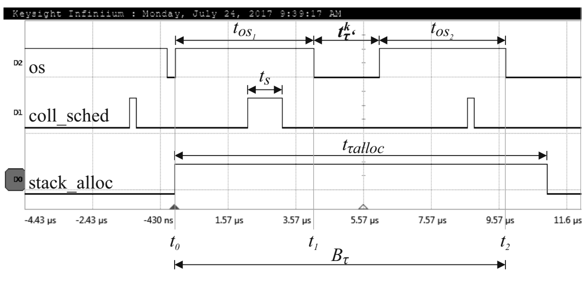


Figure 3. Execution flow example of CoStack.

the execution of the OS, the signal *coll_sched* demonstrating the part in the scheduler that is responsible for searching and scheduling a collaborative task, and the signal *stack_alloc* showing the time for allocating the required 600 Bytes stack memory to task τ. Table I lists all the measured times, and

TABLE I. Measured times for the example in Figure 3.

t_{os_1}	t_s	t_{os_2}	$t^k_{\tau\prime}$	B_τ	$t_{\tau alloc}$
3.74 µs	1.02 µs	4.08 µs	1.94 µs	9.76 µs	11.00 µs

following emphasizes some specific points in time of Figure 3:

- At time t_0, task τ requests 600 Byte stack memory. The memory is not available; therefore, the OS is called by a collaborate exception. There, the OS saves the context of task τ, does its management work including the work for scheduling the collaborative task $\tau\prime$ (i.e., t_s), and restores its context. All this in total consumes t_{os_1}.
- At time t_1, the OS returns, and task $\tau\prime$ is running. Instead of continuing with the previous preempted program counter it continues at the label `COLLABORATE`. There, its callee saved registers are restored, the collaborative stack memory $\kappa_{\tau\prime}$ is released, and the task yields to return to the OS, which again schedules task τ. These operations reflect the collaborate time $t^k_{\tau\prime}$.
- After leaving the OS (i.e., t_{OS_2}) on time t_2, there is enough stack memory available for task τ; therefore, the required stack memory is allocated to task τ.

The blocking time B_τ (i.e., $t^k_{\tau\prime}$ including the OS overheads t_{OS_1} and t_{OS_2}) and the stack allocation time $t_{\tau alloc}$ are not constant, because they depend on the number of collaborative tasks, the location of the task in the searching list, and the number of required and collaborate stack pages. However, the execution is still predictable because MultiStackMMU works predictable. Further, CoStack avoids an out-of-stack, because the collaborative task $\tau\prime$ voluntarily frees its collaborative stack memory $\kappa_{\tau\prime}$ for the higher prioritized task τ. Otherwise, task τ would not be able to execute, because no stack memory is

TABLE II. Resource comparison of the original and the extended *mosart*MCU.

	*mosart*MCU		
	Original	MultiStackMMU	CoStack
LUT slices	2799	4283	4298
FF slices	2078	2378	2378
max. frequency	73.992 MHz	63.339 MHz	63.391 MHz
Dynamic power	16 mW	21 mW	21 mW

available; consequently, this might result into an unpredictable blocking of task τ and to possibly violated real-time constraints.

B. Synthesized Results

The synthesized results, provided by the Xilinx Vivado 2017.3 development toolchain, for the Xilinx Artix-7 FPGA are listed in Table II. It compares Look Up Table (LUT) slices, Flip-Flop (FF) slices, the maximum achievable frequency, and the dynamic power of the original *mosart*MCU, and the extended versions MultiStackMMU and CoStack. If we compare the original *mosart*MCU with the extended versions, it is remarkable that the original version requires about 65 % and 87 % of LUTs and FFs, respectively. The increased resource utilization is caused by the VM address translation and some other registers that are required for handling the StackMMU approach. However, comparing the two extended *mosart*MCU versions, the resource utilization remains negligible the same.

For the maximal achievable frequency, and the dynamic power consumption the table represents a similar behavior. For the former, the large frequency reduction is caused by the implementation of the StackMMU in the already longest path of the original *mosart*MCU. For the latter, the dynamic power increases due to the additional required LUTs and FFs. However, for the two extended *mosart*MCU versions, the maximum achievable frequency remains almost the same and both require the same amount of power.

VI. Conclusion

Memory is a rare and expensive resource in embedded systems. This makes the economical usage of memory important. The stack memory has the potential to optimize the overall memory consumption because its size changes dynamically over the time. The paper proposes CoStack, an extension of the dynamically sharing stack memory concept StackMMU with collaborative stack memory. A task tags a code part with collaborative stack memory and if a higher prioritized task requires stack memory that is not available at that moment, the collaborative task deallocates the collaborative stack memory for enabling the higher prioritized task to continue. Our analysis shows that the whole stack memory requirement is virtually reduced. Further, we showed the impact of CoStack in the schedulability analysis for RM scheduling. The experimental evaluation shows that the synthesized results for the FPGA remain almost constant. Therefore, CoStack contributes to a reduction of the memory usage by introducing a collaborative stack sharing mechanism and remains predictable as aimed for real-time systems.

Acknowledgment

This research project was partially funded by AVL List GmbH and the Austrian Federal Ministry of Sciences, Research and Economy (bmwfw).

References

[1] "Micrium uC/OS-III," URL: https://www.micrium.com/rtos/ [accessed: 2018-02-27].

[2] "The FreeRTOS Kernel," URL: http://www.freertos.org/ [accessed: 2018-02-27].

[3] "Contiki: The Open Source OS for the Internet of Things," URL: http://www.contiki-os.org [accessed: 2018-02-27].

[4] T. P. Baker, "A stack-based resource allocation policy for realtime processes," in Proc. of the 11th IEEE Real-Time Systems Symposium (RTSS), Dec 1990, pp. 191–200.

[5] B. Middha, M. Simpson, and R. Barua, "MTSS: Multitask Stack Sharing for Embedded Systems," ACM Trans. on Embedded Computer Systems, vol. 7, 2008, pp. 41:1–46:37, ISSN: 0001-0782.

[6] J. Fotheringham, "Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store," Communications of the ACM, vol. 4, 1961, pp. 435–436, ISSN: 0001-0782.

[7] F. Mauroner and M. Baunach, "StackMMU: Dynamic Stack Sharing for Embedded Systems," in Proc. of the 22nd IEEE Int. Conference on Emerging Technologies and Factory Automation (ETFA), Sep 2017, pp. 1–9.

[8] Y. Wang and M. Saksena, "Scheduling Fixed-Priority Tasks with Preemption Threshold," in Proc. of the 6th Int. Conference on Real-Time Computing Systems and Applications (RTCSA), 1999, pp. 328–335.

[9] "ThreadX," 2018, URL: https://rtos.com/solutions/threadx/real-time-operating-system/ [accessed: 2018-02-27].

[10] R. Chu, L. Gu, Y. Liu, M. Li, and X. Lu, "SenSmart: Adaptive Stack Management for Multitasking Sensor Networks," IEEE Trans. on Computers, vol. 62, 2013, pp. 137–150, ISSN: 0018-9340.

[11] S. Yi, S. Lee, Y. Cho, and J. Hong, OTL: On-Demand Thread Stack Allocation Scheme for Real-Time Sensor Operating Systems. Springer Berlin Heidelberg, 2007, pp. 905–912, in Computational Science – ICCS 2007, ISBN: 978-3-540-72590-9.

[12] E. A. Hauck and B. A. Dent, "Burroughs' B6500/B7500 Stack Mechanism," in Proc. of the Spring Joint Computer Conference, ser. AFIPS '68 (Spring), 1968, pp. 245–251.

[13] D. Grunwald and R. Neves, "Whole-program Optimization for Time and Space Efficient Threads," in Proc. of the 7th Int. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), ser. ASPLOS VII, 1996, pp. 50–59.

[14] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable Threads for Internet Services," in Proc. of the 9th ACM Symposium on Operating Systems Principles (SOSP), ser. SOSP '03, 2003, pp. 268–281.

[15] M. Baunach, "CoMem: collaborative memory management for real-time operation within reactive sensor/actor networks," Trans. on Real-Time Systems, vol. 48, 2012, pp. 75–100, ISSN: 1573-1383.

[16] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," IEEE Trans. on Computers, vol. 39, Sep 1990, pp. 1175–1185, ISSN: 0018-9340.

[17] "RISC-V," URL: https://riscv.org/ [accessed: 2018-02-27].

[18] A. Waterman, Y. Lee, R. Avizienis, D. Patterson, and K. Asanovic, The RISC-V Instruction Set Manual, 2016, URL: https://riscv.org/specifications/ [accessed: 2018-02-27].

[19] F. Mauroner and M. Baunach, "EventIRQ: An Event based and Priority aware IRQ handling for Multi-Tasking Environments," in Proc. of the 20th Euromicro Conference on Digital System Design (DSD), Aug 2017, pp. 102–110.

G. EventQueue: An Event based and Priority aware Interprocess Communication for Embedded Systems

Publication Information

F. Mauroner and M. Baunach. EventQueue: An Event based and Priority aware Interprocess Communication for Embedded Systems. *In Proceedings of the 13th International Symposium on Industrial Embedded Systems (SIES)*. Graz, Austria. June 2018.

Contribution

Main author

EventQueue: An Event based and Priority aware Interprocess Communication for Embedded Systems

Fabian Mauroner and Marcel Baunach
Institute of Technical Informatics
Graz University of Technology, Graz, Austria
Email: {mauroner, baunach}@tugraz.at

***Abstract*—Modern embedded systems are targeting for isolated tasks and for an efficient Interprocess Communication (IPC). However, unifying both requirements is not a trivial challenge. In this paper, we propose EventQueue that unifies both requirements with the assistance of a hardware extension. With a message queue, the data is transferred from one task to another. Thereby, sending the data is performed in hardware, what eliminates the problem of an Operating System Priority Inversion (OS-PI) and concurrently enables the isolation of the tasks among each other. We implemented EventQueue into the *mosart*MCU, running in a Field Programmable Gate Array (FPGA). This is illustrated in performance evaluations. The evaluation shows a significant throughput improvement, what makes EventQueue well suitable for future real-time embedded systems.**

***Index Terms*—Embedded Systems; Interprocess Communication; OS-Awareness; FPGA implementation**

I. Introduction

In today's real-time embedded systems the real-time Operating System (OS) is still a fundamental layer of the whole system; and therefore, still part of today's research. In the OSs a finite set of software parts, called *task*s, is selected to be scheduled on the Central Processing Unit (CPU). Thereby, the OS selects a task according to the OS's scheduling policy. The scheduling policy of real-time OSs aims to fulfill all task's real-time constrains; thus, a task has to be started and/or finished within a lower and/or upper time bound, respectively. A violation of these timing bounds could result in dramatic situations, where machines are smashed or even human lives are affected. Therefore, it must be ensured that all real-time constraints in a real-time embedded system are satisfied.

The schedulability analysis (e.g., [1]) proves if a set of tasks does not violate real-time constrains. The analysis calculates the utilization of the whole system that requires, among other parameters, the Worst Case Execution Time (WCET) of each task. A longer WCET increases the utilization of the whole system; and thus, the utilization could exceed the threshold to ensure a feasible scheduling. To reduce the utilization, for instance, a faster CPU or a higher CPU frequency must be used, what would however increase the power consumption. However, embedded systems, as in the Internet of Things (IoT) are mostly power constrained. Therefore, a more efficient solution to improve the utilization is to reduce the WCET of the tasks.

978-1-5386-4155-2/18/$31.00 © 2018 IEEE

Modern real-time OSs support a multitude of features to assist the application developers in implementing their applications and to achieve functionalities that are not achievable at the application layer. Thus, real-time OSs support, for instance, task management, memory management, file management, and Interprocess Communication (IPC). The IPC represents the exchange of data between tasks locally (i.e., single-core) or globally (i.e., distribute systems as multi-cores). An OS may realize an IPC in different variations. Linux [2] supports IPCs via files, sockets, pipes, shared memory, and message queues. Often, these methods require synchronization primitives to avoid race-conditions (e.g., for shared memory). Thus, these IPC methods influence the performance of the whole system [3]. Especially for microkernels (e.g., [4], [5]), which are mainly found in real-time embedded systems, an IPC with a minimum of overhead is aimed, because IPCs are intensively used there.

The Memory Protection Unit (MPU) may enable the possibility to isolate confidential task data from each other. Thus, a task puts critical data into a protected memory region and non-critical data to the non-protected memory region. However, not all IPC approaches are suitable for transferring critical data from one task to another (e.g., shared memory). For protected IPC transfers the sender must be able to send confidential data to the protected memory region of the receiver task. This would be possible with a one directed IPC mechanism and suitable memory protection. Thus, message queue IPCs would be a suitable mechanism to fulfill a secure IPC transfer.

Newly received data is recognized by polling or by receiving an interrupt. The polling may consume a lot of CPU cycles only for checking, what badly influences the WCET of a task. Receiving an interrupt could introduce a rate monotonic priority inversion [6], where a high prioritized task is preempted by an Interrupt Request (IRQ) that is addressed to a lower prioritized task. In [7], we generalized the rate monotonic priority inversion definition through the priority inversion caused by the OS, as a syscall that performs operations for a lower prioritized task. We named this phenomenon Operating System Priority Inversion (OS-PI).

This paper presents EventQueue, a message queue based IPC for real-time embedded systems. EventQueue is based on an event approach that avoids the OS-PI problem, what leads to the fact that higher prioritized tasks are not unpredictably preempted while receiving new data from lower prioritized tasks. Furthermore, EventQueue reduces the number of send syscalls executing in the kernel-mode, what improves the tasks' WCETs. EventQueue does not limit the number of IPC channels, which are usually limited in other hardware solutions. Thus, it is already ready for future highly adaptive embedded real-time systems. As a side effect, the proposed approach enables the possibility of securely transferring data to tasks, isolated from each other, by using an MPU.

The rest of the paper is organized as follows: First, Section II shows similar message queue based IPC approaches in hardware. Second, Section III shows the fundamental architecture, where the EventQueue is applied, and Section IV shows the EventQueue in detail. While Section V shows performance evaluations and the synthesis result in a Field Programmable Gate Array (FPGA), Section VI discusses the performance, security, and general aspects of EventQueue. Finally, Section VII concludes this paper.

II. Related work

More than 25 years ago, the first message passing OSs has been developed, with a multitude number of IPC calls. At the beginning, the pure software IPCs were slow; therefore, already then started some research projects with the aim of supporting the IPCs by a message co-processor [3], [8]. The hardware solution showed a huge performance increase, but it have been designed for server applications in which space and power consumption are not that relevant as in embedded systems. Moreover, the real-time constraints have not been considered at all, what makes these IPCs not applicable for real-time systems.

The work in [9] presents a message queue IPC with hardware support that considers also real-time constrains by noting the priorities. The hardware supports a limited number of message queue channels, which can be owned by a task. Therefore, the number of IPC channels is limited by the hardware. This does not make this hardware suitable for today's complex embedded systems, which mostly require a multitude number of IPC channels.

In [10], the authors proposed an approach of a predictable IPC mechanism for embedded systems. Their solution is based on a two-level shared memory structure. A local memory level for each task and a global memory level to combine the local levels. The data is moved between the two memory levels, whereby the local memory level, in contrast to the global memory level, does not require a synchronization primitive. Thus, when synchronizing the global memory layer, the message passing overhead increases with the number of tasks. To counteract this issue, they adapted a Direct Memory Access (DMA) controller to perform the transfer of the data between the two layers and to reduce the delay to get the synchronization primitive of the global memory layer. Nevertheless, this approach requires the copying of the data from the local memory layer to the global memory layer and back to the local layer of the receiver task. This approach leads to a predictable IPC, but each transfer requires two memory transfers what results in a significant overhead on many IPC transfers. Further, potentially confidential data is put on the global memory and can be read by non-trustable tasks.

The message queue support in hardware is only rarely found in commercial computer architectures. ARM [11] specifies its IPC module and NXP [12] offers the Queue Manager, both implemented in hardware. ARM's IPC module is based on a specific number of mailboxes. In each mailbox an element can be sent to the receiver and the receiver is notified by an IRQ. Thus, the IRQ interrupts the current program flow, what may lead to an OS-PI that has to be avoided for real-time systems. The Queue Manager on NXP chips is a huge extension with the aim of transferring any packet to any accelerator or core in the System on Chip (SoC). The extension is built on frames that define the data location and size in the data memory, and descriptors, which handle the buffering or queuing of the frames. This is a powerful hardware extension, but not applicable to small power and resource constrained embedded systems.

Due to the rare hardware support in commercial computer architectures, the common way to implement IPCs in embedded real-time systems is to use pure software solutions. For the automotive context, in AUTOSAR [13], the IPC is realized with events. The event is a task synchronization primitive through which the tasks are able to exchange information between each other. Hereby, the OS overhead is increased due to many context switches; and furthermore, all tasks have full access to the shared memory where the information is exchanged. The avionic domain [14] offers the same IPC approach plus a queue approach. However, for sending data still a syscall has to be executed, what leads to a context switch and could lead to an OS-PI.

The proposed EventQueue approach neither restricts the number of queues nor leads to an OS-PI, what none of the previous works is able to handle in one solution.

III. Architecture

A. Terminology and Assumption

For EventQueue we assume a single-core CPU with a multi-tasking OS running a task $\tau_{run} \in T$ of the set of tasks T. Each task $\tau \in T$ possesses a static priority p_τ that is defined at compile time and follows the Rate Monotonic (RM) [1] scheduling rules (i.e., shorter deadline possesses higher priority). To synchronize the tasks with each other, the OS supports events. An event $e \in E$ is a single-directed synchronization primitive to notify a task τ, waiting for that specific event. If the event e is set, the highest prioritized task in the event queue $q_e \subset T$ is resumed.

The computation unit owns OS-awareness at hardware level. Thus, the computation unit is always aware of the currently running task's priority p_{run} and further task's information, as for instance the maximum stack size. With this knowledge, the computation unit assists the OS to achieve properties, which are not achievable at OS level, or only with a huge computation effort. To enable the OS-awareness, we assume a system architecture as depicted in Fig. 1. There, the computation unit possesses a instruction bus to the Read-Only Memory (ROM)

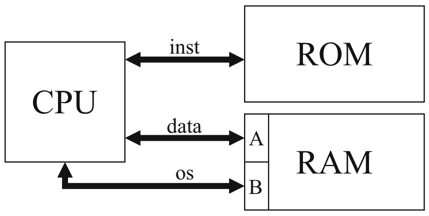


Fig. 1. The system architecture to support the EventQueue approach.

for reading the instruction and a data bus to the Random-Access Memory (RAM). Beside the data connection, a further connection to the RAM is implemented for supporting all the OS-awareness functionalities. To support such architecture the data memory must be a dual-port. A dual-port memory allows to access the memory with two independent address and data signals and possesses an arbitration logic to handle possible concurrent writes to the same memory address. Due to this architecture, the OS-awareness does not interfere the normal operation of the computation unit by operating simultaneously to it.

B. mosartMCU

To realize the EventQueue approach, we use the *mosart*MCU[1] project that implements OS awareness into embedded multi-core systems (e.g., [7], [15]). The base for the *mosart*MCU is the open source vScale[2], which is a RISC-V [16] architecture, specified by the University of California, Berkeley. vScale implements all 32 bit integers and the multiplication/division instructions from the Instruction Set Architecture (ISA) specification [17] and executes the instructions in a three stage pipeline. The specification defines 32 registers, whereas the compiler does not use the register `tp`. This register indicates the Task Control Block (TCB) of the currently running task τ_{run}, which contains information about the task including its priority p_{run}. We extended the basic implementation with an automatic read operation, triggered by the hardware if the register `tp` is changed. Therefore, the hardware is always aware of the currently running task's priority. Concurrently to the normal execution, a read operation is automatically performed by using the additional connection to the data memory through the dual-port memory. This dual-port memory is also used by some other OS-awareness extensions. Further, the RISC-V specification defines three different operating modes, whereas the *mosart*MCU supports only the non-privileged *user*-mode and the privileged *kernel*-mode. These operating modes define permissions for some instructions and for accessing Control Status Registers (CSRs), which are hardware registers used to configure and to get information from the CPU.

In the *mosart*MCU runs the hardware/software co-designed full-preemptive multi-tasking *mosart*MCU-OS. The *mosart*MCU-OS is an embedded real-time OS that supports all the OS-awareness extensions of the *mosart*MCU. The kernel is designed as a microkernel; therefore, IPC is an essential required feature to communicate among tasks.

[1]Multi-Core Operating-System-aware Real-Time MCU
[2]https://github.com/ucb-bar/vscale

C. EventIRQ

EventIRQ [7] is a hardware extension of the basic *mosart*MCU, on which EventQueue is based on. EventIRQ is an IRQ handling approach to avoid the unpredictable interruption of IRQs. To achieve that, all the interrupts are mapped to OS events, which a task τ is waiting for. Therefore, the handling of an IRQ is moved from the Interrupt Service Routine (ISR) to a task. On an IRQ, the hardware extension accesses the TCB of the triggered task by knowing the internal OS data structures. All these operations are performed simultaneously to the normal execution flow, by using the additional connection to the data memory. At the end of the TCB access, EventIRQ is aware of the currently running task's priority p_{run} and the priority of the triggered task. Thus, the currently running task is only interrupted iff the priority of the triggered task is higher than the one of the currently running task. Otherwise, the task is appended to a list, which will be caught up by the OS later on. With this postponing of the IRQ handling, the response time of the IRQ may increase; however, the unpredictable interruption of a high prioritized task is avoided and no OS-PI occurs or is at least timely bounded.

For all IRQs, except the system timer, the tasks waiting for the event are sorted by priority. Thus, on a *set event*, the highest prioritized task consumes the event. However, for the system timer all the tasks are sorted by its timeout. To handle the timeout queue properly and to avoid the OS-PI issue, EventIRQ additionally sets the system timer with the new timeout of the next waiting task in the queue.

To avoid the OS-PI caused by setting a software event, which is directed to a lower prioritized task, EventIRQ extends the base ISA with the set event `sev` instruction. This instruction performs the same operations as the mentioned process for an IRQ, but instead of operating on an IRQ event it operates on the software event.

IV. EventQueue

EventQueue enables the communication between tasks, based on the message queue mechanism and the mentioned EventIRQ approach. For sending data to a task, a queue $\rho \in Q$, of the queue set Q, is required. The queue ρ is represented as a tuple

$$\rho := (e_{\rho_s}, e_{\rho_r}, s_\rho, read_\rho, write_\rho, r_\rho, b_\rho). \tag{1}$$

The data in the queue ρ is stored in the buffer b_ρ. The size of the buffer b_ρ is defined with the buffer size s_ρ. The indices $read_\rho$ and $write_\rho$ are used to read and write the contents in the buffer b_ρ, respectively. The buffer length l_ρ represents the number of valid elements stored in the buffer b_ρ. The buffer length l_ρ is calculated with the indices $write_\rho$ and $read_\rho$ as follows:

$$l_\rho := (write_\rho - read_\rho) \bmod s_\rho \tag{2}$$

The buffer length l_ρ will be 0 for an empty buffer or s if the buffer b_ρ would support to store s_ρ elements. To distinguish between an empty and a full buffer, the buffer b_ρ is able to store up to $s_\rho - 1$ elements. Therefore, for an empty buffer the condition

$$empty_\rho := \begin{cases} true & \text{if } read_\rho = write_\rho \\ false & \text{else} \end{cases} \tag{3}$$

and for a full buffer the condition

$$full_\rho := \begin{cases} true & \text{if } read_\rho = (write_\rho + 1) \bmod s_\rho \\ false & \text{else} \end{cases} \tag{4}$$

reflects their states. For receiving data over the queue ρ, the requested size $r_\rho := [1, s_\rho - 1]$ is defined; hence, the receiver is not notified by the event e_{ρ_r} as long as the buffer length l_ρ does not reach the number of the requested size r_ρ. The event e_{ρ_s} notifies the sender about the condition, that the buffer is not full anymore. Both, the sender and receiver are suspended as long as the buffer is full or the requested size r_ρ is not reached. Thus, the triggering of the events will resume the suspended tasks if the conditions become true. The next sections will present the EventQueue realization in the *mosart*MCU and in the *mosart*MCU-OS.

A. Hardware Assistance

EventQueue extends the *mosart*MCU with an additional instruction `qwr dst, src1, src2`, which must also be supported by the compiler. The instruction triggers the operation, which transfers data to the queue ρ's buffer, in hardware. For that, the instruction uses the two source registers for referencing the queue ρ and for transferring a data content. The referenced queue is described in the Queue Control Block (QCB), which is a data structure stored in the RAM, with all the information of the queue ρ. The destination register stores the return value of the instruction that gives information about the transfer's success. After calling the `qwr` instruction the following steps are performed (also depicted in Fig. 2) with the support of the OS-awareness in the *mosart*MCU:

① The size s_ρ of the queue ρ is read from the QCB.
② Read of index write $write_\rho$, of queue ρ, in the QCB.
③ Read of index read $read_\rho$, of queue ρ, in the QCB.
④ Read of requested size r_ρ, of queue ρ, in the QCB.
⑤ The index $write_\rho$ is incremented and stored in the QCB. The destination register is filled with a value indicating success and the EventQueue continues with the next step if the buffer condition $full_\rho$ is not true. Otherwise, the index write $write_\rho$ is not updated, the destination register is filled with an error code, and the instruction is finished. To avoid race conditions, the computation unit is not allowed to continue, within the five memory operations. Thus, the pipeline stalls the *mosart*MCU for 4 cycles.

```
① read $s_\rho$
② read $write_\rho$
③ read $read_\rho$                                   CPU stalled
④ read $r_\rho$
⑤ **if** $full_\rho$ **then**
      dst := ERROR
      **abort**
  **else**
      dst := SUCCESS
      $write_\rho := (write_\rho + 1) \bmod s_\rho$
  **end if**
⑥ $b_\rho[write_\rho]$ := src2
⑦ **if** $r_\rho = l_\rho$ **then**
      set event $e_{\rho_r}$ {further handled by EventIRQ }
  **end if**
```

Fig. 2. Pseudo-code triggered by a `qwr dst, src1, src2` instruction.

⑥ If the queue ρ is not full, the content in the second source register will be stored to the buffer b_ρ at the index $write_\rho$.
⑦ If the buffer length l_ρ reaches the requested size r_ρ, EventQueue triggers the receiver event e_{ρ_r} that is located in the QCB of the queue ρ. After that, the *set event* approach of EventIRQ is executed.

All these operations, including EventIRQ, are performed on the OS connection to the RAM. Therefore, after stalling the pipeline for four instructions (i.e., the instruction consumes exactly 4 cycles) the remaining queue handling and event handling steps are performed simultaneously to the regular execution flow. The instruction replaces the syscall (i.e., call of an OS functionality, running in the kernel-mode), thus it avoids context switches and improves the execution time. Further, EventQueue ensures, due to EventIRQ, that the currently running task is only interrupted iff the priority of the waiting task exceeds its priority. Therefore, the EventQueue approach avoids also the OS-PI issues for transferring data from one task to another.

B. Software Responsibility

The OS is responsible for implementing the message queue handled in software and must consider the usage of the introduced instruction `qwr`, mentioned before. The instruction `qwr` allows sending one word per instruction call without a timeout. For better programming convenience, a function is required that calls the instruction and allows to send a specific number of words with an upper timeout. Thereby, that function is not running in the kernel-mode, but in the user-mode. Thus, it avoids a syscall, what consequently reduces the number of context switches. If the queue ρ is full (i.e., $full_\rho = true$), the syscall `wait_event_until(`e_{ρ_s}`,`t`)` is called, which leads the sending task to wait for the event e_{ρ_s} for a maximum time of t. Thus, the task is suspended until the receiver throws the event e_{ρ_s} due to the receiver that read some data in the buffer b_ρ or the waiting times out.

The receiver is completely implemented in software as a syscall and is executed in the kernel-mode. The syscall checks if the number of valid data is at least the requested size r_ρ, otherwise it calls the syscall `wait_event_until(`e_{ρ_r}`,t)` and sets the requested size r_ρ in the QCB. There, the task is suspended either until EventQueue sets the receiver event e_{ρ_r} if the queue length l_ρ reaches the requested size r_ρ, or until the timeout times out.

It is remarkable, that the sender code does not access queue ρ's QCB in user-mode. All the memory accesses for the QCB are performed either in hardware or in the syscall, executed in kernel-mode. Thus, to protect the QCBs among the tasks, the receiver task must put the queue's QCBs within an MPU protected region. With that approach, the tasks are isolated from each other and are able to communicate securely via an IPC with a high performance and OS-PI avoidance.

V. Evaluation

We implemented EventQueue into our research platform *mosart*MCU in which the *mosart*MCU-OS runs. The *mosart*MCU is implemented into a Xilinx Artix-7 FPGA, which is assembled on the Nexys 4 DDR board from Digilent. We compared the pure software queue implementation with the EventQueue approach regarding the maximal throughput and execution time for sending data. Further, we investigated the resource utilization in the FPGA and the memory consumption for the OS.

A. Performance Evaluations

The first performance evaluation investigates the maximal throughput in the *mosart*MCU. Three tasks $T := \{\tau_s, \tau_r, \tau_m\}$ are instantiated; whereby, the task τ_s sends data to the task τ_r over the queue ρ. The queue ρ's size is $s_\rho = 5$, which means that the buffer b_ρ stores up to 4 elements. Task τ_m sets a countdown and waits for it. If it expires, the task τ_m prints the number of transmitted bits.

We investigated the evaluation with different configurations: For the receiver we set the request size r_ρ to 1, 2, 3, and 4. The sender calls the function `send_queue_until(`ρ`, `d`, `s`, `t`)`, which sends the transmitting words d of sending size s to the queue ρ. If the buffer is full, the function waits until it is not full anymore or the timeout t expires. We evaluated the performance with the sending size s of 1 and 4. All combinations were tested with the pure software IPC and EventQueue approach and the results are depicted in Fig. 3. With increasing requested size r_ρ and sending size s the throughput increases for both approaches. EventQueue possesses almost the double maximal achievable throughput compared to the pure software solution. This is caused by the reduced syscalls and the shorter execution times for sending a word over the queue ρ.

The second performance evaluation investigates the execution time of the `send_queue_until` function in the pure software and in the EventQueue solution. Fig. 4 depicts the execution times of the `send_queue_until` function call with different sending sizes s of transmitting words. For EventQueue, the function requires $0.4\,\mu s$ for each additional word; for the pure software solution $0.9\,\mu s$. EventQueue does not invoke a syscall; thus it avoids a move into the kernel-mode. Thus, EventQueue improves the overall IPC transfers

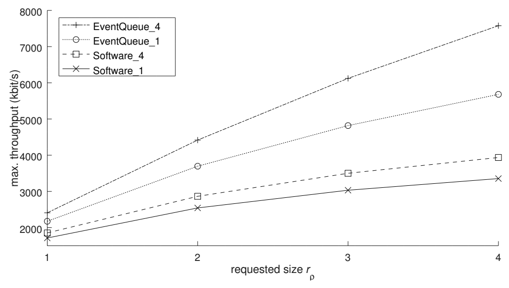


Fig. 3. Maximal throughput comparison of the pure software implementation and EventQueue with different configurations.

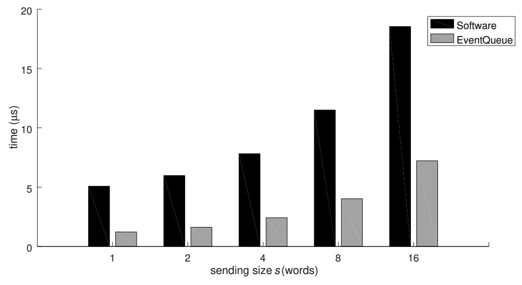


Fig. 4. Execution times of `send_queue_until(`ρ`, `d`, `s`, `t`)` with different number of sending size s.

performance; moreover, it avoids the OS-PI in IPC transfers. Especially microkernels, which are intensively using IPCs as their base concept, will benefit of EventQueue.

B. Synthesize and OS Results

The second part of the evaluation investigates the synthesis results of the FPGA and the OS memory consumption. Thereby, we compare the original *mosart*MCU, the EventIRQ extended, and the EventQueue extended version. Table I lists the synthesis results, which are reported by Xilinx Vivado 2017.3.

The Look Up Table (LUT) and Flip-Flop (FF) slices increase with the extension of EventIRQ and once again with the extension of EventQueue. The original implementation does not contain OS-awareness; and therefore, it does not require an additional connection to the RAM. Thus, the EventIRQ and the EventQueue approach require additional slices. EventQueue requires additional slices to handle the instruction `qwr` and

TABLE I
Synthesis results of the original and the extended *mosart*MCU versions.

	*mosart*MCU		
	Original	EventIRQ	EventQueue
LUT slices	2799	3543	3982
FF slices	2078	2413	2632
max. frequency	73.992 MHz	72.124 MHz	73.373 MHz
Dynamic power	16 mW	18 mW	19 mW

TABLE II
MEMORY COMPARISON OF THE ORIGINAL AND THE EXTENDED *mosart*MCU-OS VERSIONS.

	*mosart*MCU-OS		
	Original	EventIRQ	EventQueue
ROM	3952 B	4316 B	4216 B
RAM	12 B	136 B	136 B

the steps to access QCB's data over the RAM connection used by the *mosart*MCU's OS-awareness. The maximal achievable frequency remains almost the same for all three implementations. The power consumption depends on the required slices; thus, the EventQueue's power consumption is the highest of all.

We also investigated the static memory consumption of the OS, which is reported by the `gcc` compiler with the first optimization level (i.e., `-O1`) enabled. Table II lists the memory utilization of the original version and the extended versions. The original version consumes less ROM and RAM, because EventIRQ requires additional code for catching up the triggered tasks and because the interrupt vector table is moved from the ROM into the RAM. The EventQueue approach replaces the software implementation for sending an element by using the `qwr` instruction; therefore, the ROM requirement is reduced, compared to EventIRQ.

VI. DISCUSSION

The steadily growing complexity of today's real-time embedded systems requires even more security features to prevent the access to confidential information stored in a task. Thus, the protection of memory finds one's way into today's real-time embedded systems. However, the realization of IPC is then restricted. Shared memory is a feasible solution, but it requires additional synchronization primitives and leads to other issues, as for example the priority inversion problem for resource management [18]. EventQueue enables the sending of data between tasks, which are isolated from each other, because only the receiver, hardware, and OS have access to the queue's QCB. The sender uses only the addresses of the QCB to trigger the hardware extension to forward its data. Thereby, the sender cannot read data in the buffer b_ρ; and thus, EventQueue is suitable to forward confidential information.

Beside the potential to isolate the tasks from each other, EventQueue improves the throughput for sending information from one task to another. The performance of the IPC is a crucial property, especially for microkernels, which are intensively using IPCs. Embedded system's kernel designs are mostly based on microkernels; thus, the EventQueue approach is well suitable for todays and future real-time embedded systems, as in the automotive or IoT domain.

EventQueue is also applicable in other computer architecture, for protecting the QCB's data and for improving the performance. However, only by using EventQueue together with EventIRQ, the full potential of avoiding the OS-PI issues is reached, what is only possible with the OS-awareness approach in the *mosart*MCU. Furthermore, as mentioned in Section II, research and commercial IPC solutions implemented in hardware, limit the number of IPC channels. Whereby, EventQueue does limit neither the number of IPC channels, the number of tasks, nor the number of events. This is possible through the direct memory access to the RAM by the additional OS connection. In the *mosart*MCU, the OS-awareness functionalities are directly accessing the OS data structures in the RAM and no hardware registers are used for IPC channels, events, or tasks as in the mentioned past works.

VII. CONCLUSION AND OUTLOOK

In this paper, we demonstrated EventQueue for IPC among tasks in a core. EventQueue is based on EventIRQ, which avoids the OS-PI issue through OS-awareness in the *mosart*MCU. We extended EventIRQ with an additional instruction that performs the insertion of an element into the queue instead of a pure software solution. Besides the ability to isolate the tasks' data from each other, EventQueue admits a communication between tasks. We implemented the approach into our *mosart*MCU and investigated the performance, synthesis results, and OS requirements. The throughput of EventQueue for transferring data from one task to another is almost doubled, compared to the pure software solution. The EventQueue's synthesis results for the FPGA and the memory consumption in the MCU remain almost the same, compared to the EventIRQ, except the LUT and FF slices that are required by the additional hardware logic.

At hardware level, the next step is to extend our OS-awareness concepts to multi-core systems. Therefore, we need to support EventIRQ and EventQueue for multi-core embedded systems, to enable the queue IPC not only among tasks in the same core, but also among tasks across different cores, while still avoiding or at least bounding the OS-PI issue.

ACKNOWLEDGMENT

This research project was partially funded by AVL List GmbH and the Austrian Federal Ministry of Sciences, Research and Economy (bmwfw).

REFERENCES

[1] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, jan 1973.
[2] "The Linux Foundation," https://www.linuxfoundation.org/. Last access: 01.12.2017.
[3] U. Ramachandran, M. Solomon, and M. Vernon, "Hardware Support for Interprocess Communication," in *Proc. of the 14th Annual Int. Symposium on Computer Architecture (ISCA)*, ser. ISCA '87. ACM, 1987, pp. 178–188.
[4] "QNX Operating Systems," http://blackberry.qnx.com/en/products/neutrino-rtos/index. Last access: 01.12.2017.
[5] "ThreadX," https://rtos.com/solutions/threadx/.Lastaccess:01.12.2017.
[6] L. E. Leyva-del Foyo, P. Mejia-Alvarez, and D. de Niz, "Integrated Task and Interrupt Management for Real-Time Systems," *ACM Trans. on Embedded Computing Systems*, vol. 11, no. 2, pp. 32:1–32:31, jul 2012.
[7] F. Mauroner and M. Baunach, "EventIRQ: An Event based and Priority aware IRQ handling for Multi-Tasking Environments," in *Proc. of the 20th Euromicro Conference on Digital System Design (DSD)*, Aug 2017, pp. 102–110.

[8] J.-M. Hsu and P. Banerjee, “A Message Passing Coprocessor for Distributed Memory Multicomputers,” in *Proc. of the 1990 ACM/IEEE Conference on Supercomputing (SC)*, ser. Supercomputing ’90. IEEE Computer Society Press, 1990, pp. 720–729.

[9] J. Furunäs, J. Adomat, L. Lindh, J. Starner, and P. Voros, “A Prototype for Interprocess Communication Support, in Hardware,” in *Proc. of the 9th Euromicro Workshop on Real Time Systems*. IEEE, Jun 1997, pp. 18–24.

[10] S. Srinivasan and D. B. Stewart, “High Speed Hardware-assisted Real-time Interprocess Communication for Embedded Microcontrollers,” in *Proc. of the 21st IEEE Conference on Real-time Systems Symposium (RTSS)*, ser. RTSS’10. IEEE Computer Society, 2000, pp. 269–279.

[11] *PrimeCell Inter-Processor Communications Module (PL320)*, ARM Limited, 2004.

[12] K. Johnson, “QorIQ Platform: Architecture Advantages,” Presentation, Oct. 2013.

[13] “AUTOSAR,” https://www.autosar.org. Last access: 20.11.2017.

[14] “ARINC-653,” https://www.arinc.com. Last access: 20.06.2017.

[15] F. Mauroner and M. Baunach, “StackMMU: Dynamic Stack Sharing for Embedded Systems,” in *Proc. of the 22nd IEEE Int. Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, Sept 2017, pp. 1–9.

[16] RISC-V Foundation, “RISC-V,” https://riscv.org/. Last access: 01.12.2017.

[17] A. Waterman, Y. Lee, R. Avizienis, D. Patterson, and K. Asanovic, *The RISC-V Instruction Set Manual*, 2016.

[18] L. Sha, R. Rajkumar, and J. P. Lehoczky, “Priority Inheritance Protocols: An Approach to Real-Time Synchronization,” *IEEE Trans. on Computers*, vol. 39, no. 9, pp. 1175–1185, Sep 1990.

H. *mosart*MCU: Multi-Core Operating-System-Aware Real-Time Microcontroller

Publication Information

F. Mauroner and M. Baunach. *mosart*MCU: Multi-Core Operating-System-Aware Real-Time Microcontroller. *In Proceedings of the 7th Mediterranean Conference on Embedded Computing (MECO)*. Budva, Montenegro. June 2018.

Contribution

Main author

2018 7th MEDITERRANEAN CONFERENCE ON EMBEDDED COMPUTING (MECO), 11-14 JUNE 2018, BUDVA, MONTENEGRO

*mosart*MCU: Multi-Core Operating-System-Aware Real-Time Microcontroller

Fabian Mauroner and Marcel Baunach
Institute of Technical Informatics
Graz University of Technology
Graz, Austria
{mauroner, baunach}@tugraz.at

Abstract— **In this paper, we present the *mosart*MCU, which is a hardware/software co-designed Microcontroller Unit (MCU) with Operating System (OS)-awareness. Here, we present how the OS-awareness is implemented in our *mosart*MCU, running on a Field Programmable Gate Array (FPGA). Further, we show some concepts based on the OS-awareness, solving issues that are inefficient or even not applicable to be handled in pure software solutions. The synthesize results present the impact of the OS-awareness in the hardware. Hereby, the resource utilization remains constant although the number of OS instances, as tasks or events, changes.**

Keywords-Real-Time Embedded Systems; Operating System awareness; FPGA implementation; HW/SW Co-Design

I. Introduction

Embedded real-time systems are ubiquitous in our daily life. The systems are sensing the environment whereupon the actuators are performing some actions. Thereby, the systems often have to perform their operations within a specific time. To fulfill those time requirements, a real-time system is required. In a real-time system, a time window specifies when an operation is allowed to be finished. Thus, a lower time bound and an upper time bound are specified. If those time boundaries are sometimes slightly violated, the calculated results may be ignored or the result has temporally a lower quality. A system with such weak time boundaries is called soft real-time system. Whereas, if the time bound violations end up in a smashed machine or, even worse, affect human lives, those systems are called hard real-time systems [1]. Here, the Operating System (OS) must ensure that no time boundaries are violated.

The real-time OS schedules the tasks, which are parts of an application, with another scheduling policy as desktop or server systems. In desktop systems, the scheduler tries to schedule the tasks in a way that the user has the feeling of a highly reactive system, whereas for server systems high throughput is aimed. For real-time OSs, the scheduling policy is designed to observe strictly all the time boundaries. Thus, there exist many different scheduling approaches; for instance, Rate Monotonic (RM) or Earliest Deadline First (EDF). Those scheduling policies are based on priorities. This means, a higher prioritized task will be scheduled instead of a lower prioritized task. However, through resource conflicts a priority inversion may occur [2]. This means, a lower prioritized task runs instead of a higher prioritized one, which would run without that resource conflict. A priority inversion may occur on Interrupt Request (IRQ) handling, too. In traditional embedded systems, the IRQs are prioritized beyond all task priorities. Thus, if an IRQ is mentioned for a low prioritized task, the Interrupt Service Routine (ISR) is unpredictably interrupting a higher prioritized task. This unpredictable interruption makes it impossible to check all the real-time requirements of a real-time embedded system.

In addition, embedded systems are often memory constraint. This means, the memory must be used in a very efficient way. Especially the size of stack is a constantly changing memory. The stack memory, temporally stores the registers of the Central Processing Unit (CPU) on a function call or IRQ, and temporally stores allocated variables of the function. If the function is finished, the temporal variables are trashed, by changing the stack pointer in the stack frame. In real-time systems, usually for each task an own individual stack frame is assigned, to allow a high schedulability. However, most of the time the individual stack frame is not fully utilized, what ends up in a waste of memory. A Memory Management Unit (MMU) however is rarely found in embedded systems, because it behaves non-deterministically and consumes, compared to the rest of the embedded system, an enormous power [3].

Todays real-time embedded system designs are still divided into software and hardware layers. In this paper, we present a new computer architecture for embedded real-time systems that supports OS-awareness in hardware by merging the hardware and software layer together. The OS-awareness enables the CPU to be aware of the internal data structure of the OS, leading to solutions that are not reachable with a pure software OS. Thus, we show how the priority inversion on IRQs is solved and how the memory consumption is reduced, by having OS-awareness in the Microcontroller Unit (MCU).

978-1-5386-5683-9/18/$31.00 ©2018 IEEE

The rest of the paper is organized as follows: Section II introduces related works regarding OS implementations in hardware. Thereby, we demonstrate the lacks of those solutions, which can be avoided with the proposed *mosart*MCU presented in Section III. Section IV shows the resource utilization of the *mosart*MCU, and Section V concludes this paper.

II. Related Work

OS-awareness is little or not found in embedded computer systems. The only support are the two permission levels, namely the kernel-mode and the user-mode. The user-mode has only a restricted access to addresses and peripherals as well as to some instructions; therefore, this mode is usually used for the tasks. Whereas, the kernel-mode has full access and is mentioned to be used by the OS. However, academic projects already investigated how to implement an OS fully or partially into hardware.

FASTCHART [4] and FASTHARD [5] are the first fully implemented OSs in hardware. There, a co-processor implements OS functionalities that are executed simultaneously to the normal CPU execution. The co-processor contains the Task Control Blocks (TCBs), a scheduler, and queues for handling the ready tasks and waiting tasks. Thus, IRQs are handled by moving a task, waiting for an IRQ, from the waiting queue to the ready queue. This leads to a priority unification. However, the number of tasks and other resources are configured statically at development time. Therefore, it is not adaptable at run-time. Further projects, as Silicon OS [6], SEOS [7], or µC-OS-III HW-RTOS [8] are implementing the OS fully into hardware, with the same restrictions. The Real-Time Task Manager [9], which implements only parts of the OS into hardware, has the same restrictions. Here, a hardware/software co-designed OS is used.

The fully hardware implemented OS or the hardware/software co-designed OS approaches have not been accepted in the industry, because it is still thought, that hardware and software are two independent layers. However, future CPU architecture development needs to bring OS development directly in-house to use all the benefits of the hardware [10]. Thus, our proposed OS-aware *mosart*MCU is hardware/software co-designed, what does not limit the number of tasks; and at the same time, it avoids unpredictable IRQs and reduces the stack memory consumption.

III. The *mosart*MCU

The aim of the *mosart*MCU[1] project is the implementation of OS-awareness into embedded multi-core real-time systems. The MCU is based on the vScale[2] soft-core from the University of California, Berkeley. The vScale implementation uses the royalty-free RISC-V [11] Instruction Set Architecture (ISA), executed in a three stage pipeline. The RISC-V specifies 32 CPU registers. One of the CPU registers is the Task Pointer (TP). In the *mosart*MCU, the TP is used to point to the TCB of the currently running task, as depicted in Fig. 1. To achieve OS-awareness, the MCU is aware of the internal OS structure. Therefore, the *mosart*MCU knows the relative offsets in the

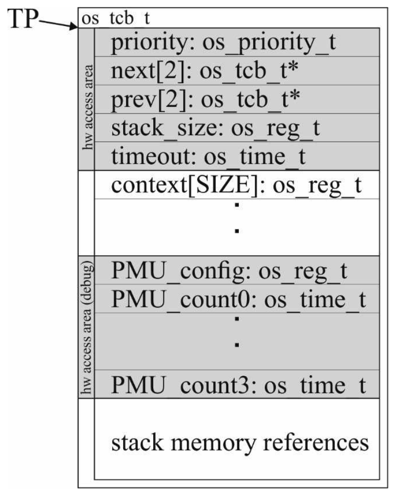


Figure 1. TCB structure accessed by the register TP.

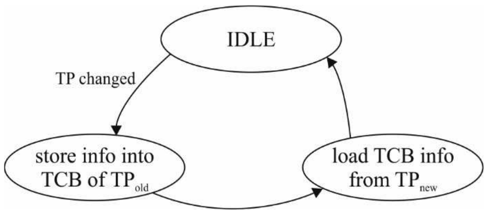

IDLE
TP changed
store info into TCB of TP$_{old}$
load TCB info from TP$_{new}$

Figure 2. *mosart*MCU state machine on a TP change.

TCB for reading or writing information of the currently running task. If the OS scheduler selects a new task, the TP will be changed.

The hardware recognizes the TP change and starts to store the currently running task's information into the TCB. In the next step, the information of the new scheduled task is read into the hardware. Fig. 2 shows the state machine. For instance, the Performance Monitor Unit (PMU) configurations and counters are automatically stored and restored [12]. Thus, if a system supports a number of performance measurements (i.e., run-time, IRQ counter, etc.), each task is able to individually use that number of performance measurements for its purpose. To be aware of the currently running task's priority in the read state, the priority and further information of the task are read. To improve the performance of the read and write operations on a TP change and for supporting other OS-aware features, we propose to use an architecture as depicted in Fig. 3. Additionally to the common instruction and data connection, the *mosart*MCU possesses a second connection to the data memory, in which the TCBs are stored. To support that, the

[1]Multi-Core Operating-System-Aware Real-Time MCU
[2]https://github.com/ucb-bar/vscale

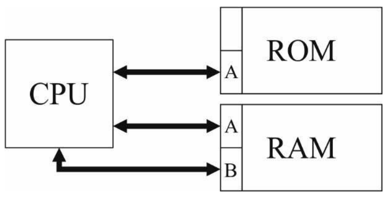


Figure 3. The system architecture to support the OS-awareness.

precondition is that the data memory is implemented as a dual-port memory. A dual-port memory contains two independent memory access interfaces, found in Field Programmable Gate Arrays (FPGAs) as the Xilinx Artix-7 [13]. With this architecture, the MCU has the knowledge of task information and is able to handle IRQs simultaneously to the regular execution flow.

A. EventIRQ

The EventIRQ [14] avoids the unpredictable interruption caused by an IRQ. EventIRQ is based on the idea, that the whole IRQ handling software is mapped to tasks. The tasks are waiting for an OS event, which is a one-bit event for signaling the occurrence of the event. If an IRQ is triggered, EventIRQ is responsible for handling that occurrence simultaneously to the regular execution flow. The internal hardware is aware of the OS data structure; thus, it removes the task, which is waiting for the triggered IRQ, from the waiting queue and appends it to an internal handled list. During those operations, the EventIRQ reads the priority of the task that waited for the event and compares it with the currently running task's priority. Iff the triggered task's priority exceeds the priority of the currently running task, an interrupt occurs and the OS schedules the new task. Otherwise, no interruption occurs and the OS catches up the triggered task through the internal handled list later on.

The EventIRQ approach is not limited to standard IRQs, but it is also responsible for handling the OS timer queue. The OS timer queue contains tasks, ordered according to their timeouts. If an OS timer queue is triggered, EventIRQ is responsible for setting the new timeout value of the next task waiting in the OS timer queue. Furthermore, a software set event syscall would end up in an Operating System Priority Inversion (OS-PI), as if for instance a higher prioritized task sets the event for a lower prioritized task that is waiting for it. Thus, an unnecessary jump into OS is done; for it we implemented a new instruction performing the same operation as a triggered IRQ, but EventIRQ performs its operations on the software event instead on the IRQ events.

B. EventQueue

The EventQueue [15] approach bases on the EventIRQ and implements Inter-Process Communication (IPC) in the *mosart*MCU. The EventIRQ approach makes it possible, that the receiver task will not produce an OS-PI through the notification of new incoming data. Furthermore, the OS-awareness is aware of the OS data structure of the queue, what means that all the operations needed to transfer the data is done by hardware. The sender task is not accessing the receiver's queue data structure; therefore, it enables a secure transfer where no other task is able to read confidential data if the data structure is protected by a Memory Protection Unit (MPU). Compared to other hardware supported IPC modules (e.g., [16], [17]), EventQueue does not limit the number of IPC instances. Furthermore, we extended the EventIRQ and EventQueue approach to multi-core environments with the Remote Instruction Call (RIC) [18] approach.

C. StackMMU

The StackMMU [19] approach handles the stack memory growth, shrinkage, and access in a deterministic time and at the same time; it shares the stack memory among all tasks. The shared stack memory is divided into pages. Further, the stack memory is accessed with Virtual Memory (VM) addresses, which are translated by the StackMMU to Physical Memory (PM) addresses. The translation table is allocated in the TCB of the respective task. If a task needs more stack space, the hardware automatically allocates a new page from a pool of pages, and the page's PM address is stored into the TCB. If the stack is not needed anymore, the StackMMU automatically returns the page to the pool of pages. For a stack access, StackMMU looks up the page address stored in the TCB of the currently running task, and in the next cycle, the stack memory access is performed. Thus, all stack operations require an additional cycle, but they are still working predictable, what is aimed for real-time embedded systems

D. CoStack

The CoStack [20] approach supports the scheduling of higher prioritized tasks, if the system runs out-of-stack-memory. Thus, a task declares a part of its code as collaborative, similar to a try-catch block in languages as Java or C++. If the stack memory runs out, the scheduler schedules a lower prioritized task that declares its stack as collaborative, and it will deallocate its collaborative stack memory. Thus, the higher prioritized is blocked, whereby its blocking time is bounded. Here, the OS-awareness is aware of an out-of-stack-memory condition and handles it properly in hardware.

IV. Resource Utilization

In this section, we show the resource utilization of the hardware (i.e., *mosart*MCU). The synthesize results are reported by the Xilinx Vivado 2017.3 for the Xilinx Artix-7 FPGA. All the results are listed in Table I.

The *mosart*MCU consumes more Look-Up-Tables (LUTs) and Flip-Flops (FFs) if more features are implemented. This is caused by the additional internal states that must be saved and the additional logic to handle all those features. The maximal frequency reduces with more enabled resource. This is on one hand the increasing logic resulting in longer paths, and on the other hand, it is the StackMMU implemented in the already longest path of the CPU. To counteract this frequency reduction, an additional pipeline stage would help. The dynamic power consumption is slightly increased through the additional LUTs and FFs. However, if we compare the increases of the StackMMU with other address virtualization

TABLE I. SYNTHESIZE RESULTS OF THE ORIGINAL AND THE EXTENDED *mosart*MCU VERSIONS.

Features	None	EventIRQ & EventQueue	StackMMU & CoStack	All
LUT slices	5214	6529	6749	7649
FF slices	3802	4433	4153	4717
Max. frequency	66.32 MHz	58.12 MHz	41.49	39.31
Dynamic power	29 mW	30 mW	32mW	33 mW

approaches as MMUs (e.g., [3]), the additional power consumption is low. Compared to other solutions that are implementing the OS fully or partly in hardware, our solution does not depend on the number of OS instances (e.g., tasks, events, etc.). This is reached by the knowledge of the internal OS data structure by the hardware.

V. CONCLUSION

In this paper, we presented the OS-awareness concept in our *mosart*MCU. The *mosart*MCU is aware of the internal OS data structures; thus, it can operate on them simultaneously to the normal execution flow. With this approach, we avoid or at least bound the OS-PI problem for IRQs and IPCs. For real-time systems, this is essential for proving the satisfaction of all real-time requirements. Furthermore, the *mosart*MCU allows the sharing of stack memory and still operates deterministically, aimed for real-time systems. Thus, we claim, that the era of developing the OS and the MCU independently is over. Future embedded real-time systems have to be hardware/software co-designed to fulfill all their challenging requirements. This paper presented solutions for handling only a small subset of problems in embedded real-time system. Therefore, we suggest that with more OS-awareness in future MCUs the remaining problems will be reduced.

ACKNOWLEDGMENT

This research project was partially funded by AVL List GmbH and the Austrian Federal Ministry of Sciences, Research and Economy (bmwfw).

REFERENCES

[1] A. S. Tanenbaum and H. Bos, Modern Operating Systems: Global Edition, 4th ed. Prentice Hall, 2014.

[2] B. W. Lampson and D. D. Redell, "Experience with Processes and Monitors in Mesa," Communications of the ACM, vol. 23, no. 2, pp. 105–117, Feb. 1980.

[3] H.-C. Ng, Y.-M. Choi, and H. K.-H. So, "Direct virtual memory access from FPGA for high-productivity heterogeneous computing," in Proc. of the Int. Conference on Field-Programmable Technology (FPT). IEEE, Dec 2013, pp. 458–461.

[4] L. Lindh and F. Stanischewski, "FASTCHART - A Fast Time Deterministic CPU and Hardware Based Real-Time-Kernel," in Proc. of the Euromicro '91 Workshop on Real-Time Systems. IEEE, 1991, pp. 36–40.

[5] L. Lindh, "FASTHARD - A Fast Time Deterministic HARDware Based Real-time Kernel," in Proc. of the 4th Euromicro Workshop on Real-Time Systems. IEEE, June 1992, pp. 21–25.

[6] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, and M. Imai, "Hardware Implementation of a Real-time Operating System," in Proc. of the 12th TRON Project International Symposium. IEEE, Nov 1995, pp. 34–42.

[7] S. E. Ong, S. C. Lee, N. B. Z. Ali, and F. A. B. Hussin, "SEOS: Hardware Implementation of Real-Time Operating System for Adaptability," in Proc. of the 1st Int. Symposium on Computing and Networking (CANDAR). IEEE, 2013, pp. 612–616.

[8] "uC/OS-III HW-RTOS," https://www.micrium.com/rtos/ucosiiihwrtos /features/. Last access: 12.10.2016.

[9] P. Kohout, B. Ganesh, and B. Jacob, "Hardware support for real-time operating systems," in Proc. of the 1st IEEE/ACM/IFIP Int. Conference on Hardware/ Software Codesign and Systems Synthesis (CODES+ISSS). ACM, Oct 2003, pp. 45–51.

[10] C. Schlager, "Keynote: The Impact of Operating Systems on Modern ¨CPU Designs (and Vice Versa)," in Proc. of the 21st Int. Conference Architecture of Computing Systems (ARCS), Dresden, Germany, Feb. 2008.

[11] A. Waterman, Y. Lee, R. Avizienis, D. Patterson, and K. Asanovic, The RISC-V Instruction Set Manual, 2016.

[12] T. Scheipel, F. Mauroner, and M. Baunach, "System-Aware Performance Monitoring Unit for RISC-V Architectures," in Proc. of the 20th Euromicro Conference on Digital System Design (DSD), Aug 2017, pp. 86–93.

[13] Xilinx Inc., "Artix7," https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html.

[14] F. Mauroner and M. Baunach, "EventIRQ: An Event based and Priority aware IRQ handling for Multi-Tasking Environments," in Proc. of the 20th Euromicro Conference on Digital System Design (DSD), Aug 2017, pp. 102–110.

[15] F. Mauroner and M. Baunach, "EventQueue: An Event based and Priority aware Interprocess communication for Embedded Systems," in Proc. of the 13th Int. Symposium on Industrial Embedded Systems (in press), IEEE, June 2018.

[16] J. Furunas, J. Adomat, L. Lindh, J. Starner, and P. Voros, "A Prototype ¨ for Interprocess Communication Support, in Hardware," in Proc. of the 9th Euromicro Workshop on Real Time Systems. IEEE, Jun 1997, pp. 18–24.

[17] PrimeCell Inter-Processor Communications Module (PL320), ARM Limited, 2004.

[18] F. Mauroner and M. Baunach, "Remote Instruction Call: An RPC approach on Instructions for Embedded Multi-Core Systems," in Proc. of the 19th Int. Conference on Industrial Technology (ICIT), IEEE, Feb. 2018, pp. 1442–1446.

[19] F. Mauroner and M. Baunach, "StackMMU: Dynamic Stack Sharing for Embedded Systems," in Proc. of the 22nd IEEE Int. Conference on Emerging Technologies and Factory Automation (ETFA), IEEE, Sept 2017, pp. 1–9.

[20] F. Mauroner and M.Baunach, "CoStack: Collaborative Stack Sharing for Embedded Real-Time Systems," in Proc. of the 13th Int. Conference on Systems (ICONS) (in press), June 2018.

Appendix

A. "O" Standard Extension for OS Awareness, Version 1.0

This section describes the standard OS awareness extension, named "O". It introduces new CSRs and instructions. The instructions trigger an OS event and an IPC transfer, both performed by the OS awareness in the MCU. To use the OS awareness extensions, the internal OS structure has to follow the definition shown in Figure A.2 and Figure A.3.

```
struct os_tcb{
  os_priority_t       priority;
  struct os_tcb       *next[2];
  struct os_tcb       *prev[2];
  os_reg_t            stack_size;
  os_time_t           timeout;
  // further variables, not accessed by the hardware
};
```

Figure A.2.: TCB definition in *mosart*MCU-OS.

```
struct os_qcb {
  os_event_t     tasks_wait_to_receive;
  os_event_t     **tasks_wait_to_send;
  os_reg_t       tasks_waiting_to_send_num;
  os_reg_t       size;
  os_reg_t       wait_for_size;
  os_reg_t       *read;
  os_reg_t       *write;
  os_reg_t       data[];
};
```

Figure A.3.: QCB definition in *mosart*MCU-OS.

Additional instructions

To support the O extension the additional instructions listed in Table A.1 are necessary.

Table A.1.: Instructions for the O extension.

Name	Arguments	Description
`sev`	src1	Set Event
`qwr`	dst, src1, src2	Write Queue

Set Event Instruction (`sev`)

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
000000000000	src1	SEV	00000	OS

The set event instruction `sev` performs the EventIRQ approach at the OS instance address stored in source register `src1`.

Queue Write Instruction (`qwr`)

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:0]	rs2	rs1	funct3	rd	opcode
12	5	5	3	5	7
000000000000	src2	src1	QWR	dst	OS

The queue write instruction `qwr` performs the EventQueue approach at the OS instance, which address is stored in source register `src1`. The source register `src2` contains the data content to be transferred. The destination register `dst` contains the success of the instruction; in case of success a 0, otherwise a 1 (i.e., caused by a full queue buffer).

Additional CSRs

The CSRs, which support the O extension, are listed in Table A.2.

Table A.2.: CSRs for the O extension.

Number	Name	Description
0x780	`pth`	Pending Tasklist Head
0x781	`ptt`	Pending Tasklist Tail
0x782	`mic`	Machine Interrupt Control
0x783	`mevec`	Machine Event Vector
0x784	`sstart`	Stack Start
0x785	`send`	Stack End
0x786	`sfree`	Stack Free
0x787	`spage_size`	Stack Page Size
0x788	`tss`	Task Stack Size
0x789	`spf`	Stack Pages Free
0x800	`ptp`	Priority Task Pointer
0x801	`spr`	Stack Pages Required
0xFC0	`core_id`	Core Identifier

Pending Tasklist Head Register (`pth`)

The `pth` is a read and write register indicating the first task in the pending task list ϕ. This register must be implemented if EventIRQ is implemented. The PTP_ADDR contains the address

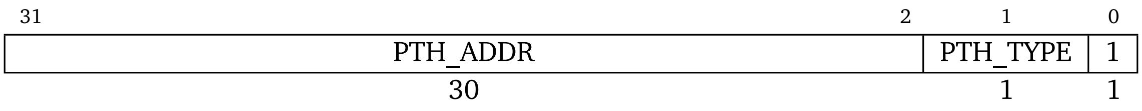


Figure A.4.: Pending Tasklist Head Register (`pth`)

of the first task in the pending task list ϕ. The PTH_TYPE indicates if the first task was triggered by a regular event (i.e., 0) or by a timeout event (i.e., 1). The Least Significant Bit (LSB) is always set to 1. The `pth` register is used by the OS to easily traverse from the top to the tail in the pending task list ϕ.

Pending Tasklist Tail Register (`ptt`)

The `ptt` is a read and write register indicating the last task in the pending task list ϕ. This register must be implemented if EventIRQ is implemented. The PTT_ADDR contains the address

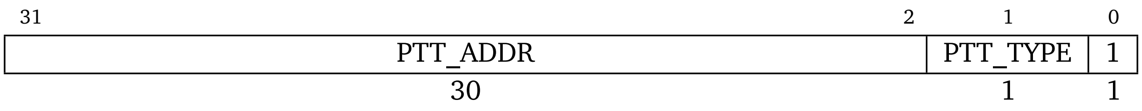


Figure A.5.: Pending Tasklist Tail Register (`ptt`)

of the last task in the pending task list ϕ. The PTH_TYPE indicates if the last task was triggered by a regular event (i.e., 0) or by a timeout event (i.e., 1). The LSB is always set to 1. The `ptt` register is used by EventIRQ to append triggered tasks to the pending task list ϕ.

Machine Interrupt Control Register (`mic`)

The `mic` is a read-only register indicating the type of each IRQ handled by EventIRQ. This register must be implemented if EventIRQ is implemented. Each bit corresponds to an IRQ. If

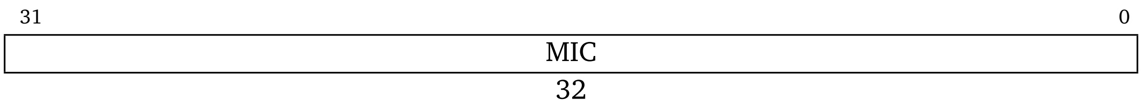


Figure A.6.: Machine Interrupt Control Register (`mic`)

an IRQ is configured with 0, EventIRQ will handle it such as a regular IRQ. Otherwise, EventIRQ will handle it such as a timer event and sets the OS timer with the timeout value of the next task in the timeout waiting queue.

Machine Event Vector Register (`mevec`)

The `mevec` is a register containing a word-aligned address indicating the start of the event vector table. This register must be implemented if EventIRQ is implemented. Starting from

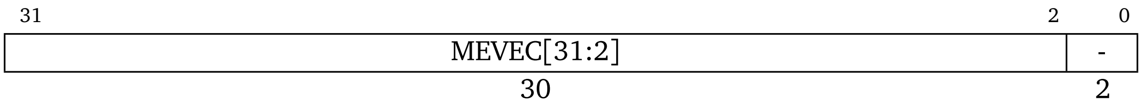


Figure A.7.: Machine Event Vector Register (`mevec`)

the address stored in `mevec`, an event must be instantiated for each IRQ by the OS. Thus, all the IRQs are mapped to the corresponding events in the event vector table, and EventIRQ is responsible for handling the corresponding event on a triggered IRQ.

Stack Start Register (`sstart`)

The `sstart` CSR is a register containing the word-aligned start address of the stack area for the StackMMU approach. This register must be implemented if StackMMU is implemented.

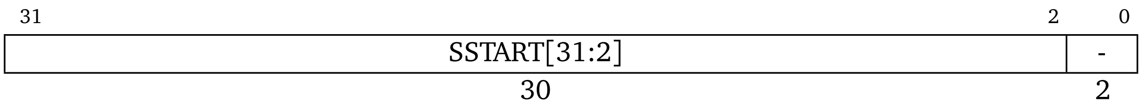


Figure A.8.: Stack Start Register (`sstart`)

Stack End Register (`send`)

The `send` CSR is a register containing the word-aligned end address of the stack area for the StackMMU approach. This register must be implemented if StackMMU is implemented.

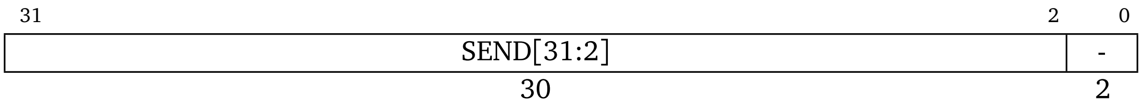


Figure A.9.: Stack End Register (`send`)

Stack Free Register (`sfree`)

The `sfree` CSR is a register containing the word-aligned address of a free stack page in the stack area for the StackMMU approach. This register must be implemented if StackMMU is implemented. In the OS initialization phase, this register must be initialized to the first free stack page to work properly. If the OS initialized all the pages properly, then, while StackMMU runs, the `sfree` register is modified by the StackMMU approach.

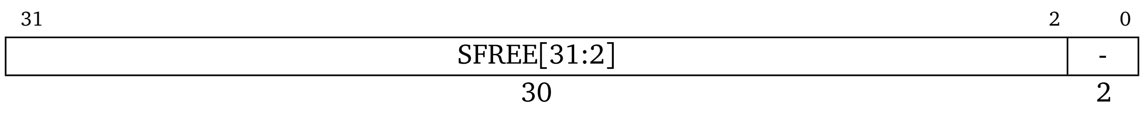


Figure A.10.: Stack Free Register (`sfree`)

Stack Page Size Register (`spage_size`)

The `spage_size` is a partly read-only and partly read-and-write register configuring and representing the page size in the StackMMU approach. This register must be implemented if StackMMU is implemented. The SPAGE_SIZE is read-only and returns the stack page size

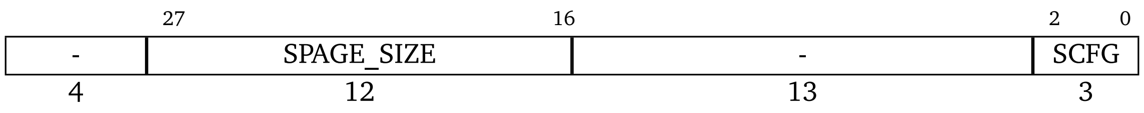


Figure A.11.: Stack Page Size Register (`spage_size`)

configured with SCFG. Table A.3 lists the corresponding stack page size for the configured SCFG.

Table A.3.: SPAGE_CONFIG configuration.

SCFG	SPAGE_SIZE
000	16 Bytes
001	32 Bytes
010	64 Bytes
011	128 Bytes
100	256 Bytes
101	512 Bytes
110	1024 Bytes
111	2048 Bytes

Task Stack Size Register (`tss`)

The `tss` register is a read-only register containing the stack memory size of the currently running task. This register must be readable in implementations with enabled StackMMU. On

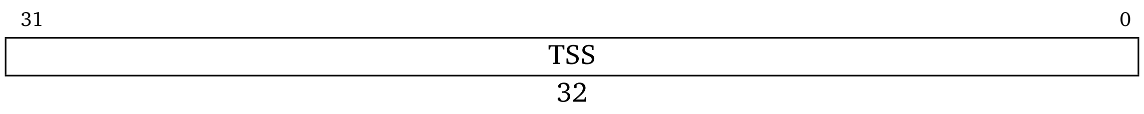


Figure A.12.: Task Stack Size Register (`tss`)

a `tp` change, the hardware automatically reads the task's stack size from the TCB and stores it into `tss`. This value is primarily required by StackMMU to allocate correctly the pages in the PP-LUT.

Stack Pages Free Register (`spf`)

The `spf` CSR is a 12-Bit read-only register containing the number of free stack pages in the StackMMU approach. This register must be readable in implementations with enabled CoStack. The `sstart`, `send`, and `space_size` define the available number of stack pages for

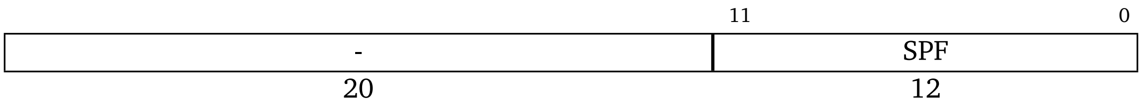


Figure A.13.: Stack Pages Free Register (`spf`)

the StackMMU. With the available number of stack pages and the internally recorded number of allocated stack pages to tasks, the field `SPF` is calculated.

Priority Task Pointer Register (`ptp`)

The `ptp` register is a read-only register that indicates the priority of the currently running task. This register must be readable in implementations with enabled OS awareness. On a `tp` change,

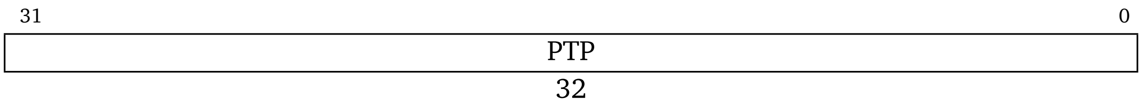


Figure A.14.: Priority Task Pointer Register (`ptp`)

the hardware automatically reads the task's priority from the TCB and stores it into `ptp`.

Stack Pages Required Register (`spr`)

The `spr` register is a 12-Bit read-only register representing the required number of stack pages on a collaboration exception. This register must be readable in implementations with enabled CoStack. On a collaboration exception, the OS has to read the number of required stack pages

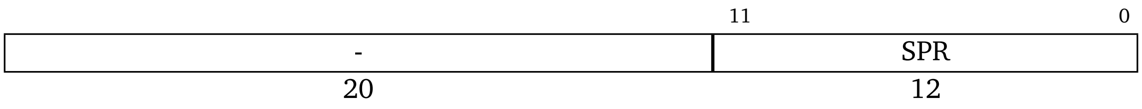


Figure A.15.: Stack Pages Required Register (`spr`)

and to store it into the task's TCB that caused the collaboration exception. By comparing this stored information in the task's TCB with the `spf` register, the OS is able to decide if enough stack pages are available to schedule the task. Otherwise, the OS has to search a collaborative task that offers collaborative stack memory to free it.

Core Identifier Register (`core_id`)

The `core_id` register is a read-only register containing the identifier of the core. The register must be readable in all implementations. For a single-core system, the register returns 0.

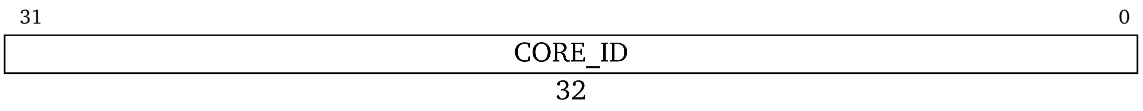


Figure A.16.: Core Identifier register (`core_id`)

B. *mosart*MCU-OS References

B.1. *mosart*MCU-OS Data Type Reference

```
typedef unsigned long            os_reg_t;
typedef unsigned int             os_priority_t;
typedef uint64_t                 os_time_t;
#define DEADLINE_INFINITE        UINT64_MAX
typedef uint64_t                 os_delay_t;
typedef struct os_tcb            os_tcb_t;
typedef union os_event           os_event_t;
typedef struct os_queue          os_queue_t;
typedef struct os_remote_queue   os_remote_queue_t;
typedef struct os_resource       os_resource_t;
```

```
struct os_tcb{                                      // task $\tau$
    //--- start hardware access area
    os_priority_t       priority;                   // task priority $p_\tau$
    struct os_tcb       *next[2];                   // next pointers $next_\tau$ and $tnext_\tau$
    struct os_tcb       *prev[2];                   // previous pointers $prev_\tau$ and $tprev_\tau$
    os_reg_t            stack_size;                 // maximum required stack memory $\varsigma_\tau$
    os_time_t           timeout;                    // timeout $\hat{t}_\tau$
    //--- end hardware access area
    os_reg_t            context[CONTEXT_SIZE];      // task $\tau$'s saved context
    os_priority_t       base_priority;              // task base priority
    struct os_tcb       **member_list;              // reference of membered priority list
    os_reg_t            stack_req;                  // required stack pages (read from CSR spr)
    os_reg_t            *stack_collaborate_fp;      // CoStack's stored frame pointer
    os_reg_t            *collaborate_handler;       // CoStack's stored collaborate handler address
    struct os_tcb       *ordered_next;              // pointer to the next task
    os_resource_t       *wait_resource;             // reference to the waiting resource
    // stack page areas start here
};
```

```
union os_event {                                                // event $e$
    os_tcb_t  *queue;
    struct {
        unsigned long value   : 1;                              // 0=not set, 1=set
        unsigned long type    : 1;                              // 0=regular, 1=timer
        unsigned long address : sizeof(unsigned long)-2;        // reference to task
    } bits;
};
```

```
struct os_queue {                                   // queue $\rho$
    os_event_t  tasks_wait_to_receive;              // receiver event $e_{\rho_r}$
    os_event_t  **tasks_wait_to_send;               // sender event $e_{\rho_s}$
    os_reg_t    tasks_waiting_to_send_num;          // counting tasks that are blocked due to $full_\rho = true$
    os_reg_t    size;                               // size $s_\rho$
    os_reg_t    wait_for_size;                      // request size $r_\rho$
    os_reg_t    *read;                              // read index $read_\rho$
    os_reg_t    *write;                             // write index $write_\rho$
    os_reg_t    data[];                             // buffer $b_\rho$
};
```

```
struct os_remote_queue {
    os_queue_t  *q;                                 // reference to the remote queue $\rho$
    os_event_t  e;                                  // event for waiting if $full_\rho = true$
};
```

```
struct os_resource {                        // resource
    os_tcb_t        *owner;                 // owner of the resource
    os_event_t      *e;                     // event queue
    int             value;                  // counts how often the task called get_resource_until
#if defined(PCP) || defined(HLP)
    os_priority_t priority;                 // ceiling priority resource management protocol
#endif
};
```

B.2. *mosart*MCU-OS API Reference

```
void os_init(void);
void os_run(void);
int os_register_task(task_entry_t t, os_reg_t stack_size, os_priority_t priority);
```

```
os_time_t get_current_time(void);
void task_exit(void);
int yield (void);
```

```
int set_event (os_event_t *e);
int clear_event (os_event_t *e);
int wait_event_until (os_event_t *e, os_time_t t);
int wait_event_until_new (os_event_t *e, os_time_t t);
int sleep_until (os_time_t t);
int sleep (os_delay_t d);
int wait_event (os_event_t *e);
int wait_event_new (os_event_t *e);
int wait_event_for (os_event_t *e, os_delay_t t);
int wait_event_for_new (os_event_t *e, os_delay_t t);
```

```
int resource_register_priority(os_resource_t *t, os_priority_t p);
int get_resource_until (os_resource_t *r, os_time_t t);
int get_resource_for (os_resource_t *r, os_delay_t d);
int get_resource (os_resource_t *r);
int release_resource  (os_resource_t *r);
```

```
#define CREATE_REMOTE_QUEUE(name, queue)
#define CREATE_QUEUE_GLOBAL(name, size, events)
#define CREATE_QUEUE_LOCAL(name, size)

int send_queue (os_queue_t *q, os_reg_t d);
int send_queue_until (os_queue_t *q, os_reg_t* data, os_reg_t size, os_time_t t);
int send_remote_queue_until (os_remote_queue_t *q, os_reg_t* data, os_reg_t size, os_time_t t);
int wait_queue_until (os_queue_t *q, os_reg_t *data, os_reg_t size, os_time_t t);
```

```
#define COLLABORATIVE_STACK
#define COLLABORATE_STACK
#define COLLABORATE_STACK_END
```

Bibliography

[1] Mark Weiser. The Computer for the 21st Century. *ACM SIGMOBILE Mobile Computing and Communications Review*, 3(3):3–11, July 1999.

[2] Crossbow Technology Inc., San Jose (USA). *MICA2 Wireless Measurement System*, 2005.

[3] Jan Beutel, Oliver Kasten, Friedemann Mattern, Kay Römer, Frank Siegemund, and Lothar Thiele. *Prototyping Wireless Sensor Network Applications with BTnodes*, pages 323–338. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[4] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: enabling ultra-low power wireless research. In *Proc. of the 4th Int. Symposium on Information Processing in Sensor Networks (IPSN)*, pages 364–369. IEEE, April 2005.

[5] Marcel Baunach, Reiner Kolla, and Clemens Mühlberger. *SNoW5 : A Versatile Ultra Low Power Modular Node for Wireless Ad Hoc Sensor Networking*, pages 55–59. Springer Berlin Heidelberg, 2006.

[6] Norbert Sailer, Fabian Mauroner, and Marcel Baunach. meto1 - A Versatile and Modular 32 bit low power Sensor Node Protoyping Platform for the IoT. In *Proc. of the Int. Conference on Embedded Wireless Systems and Networks (EWSN)*, pages 290–293. Junction Publishing, February 2017.

[7] Internet of things - strategic research roadmap. `http://www.internet-of-things-research.eu/pdf/IoT_Cluster_Strategic_Research_Agenda_2009.pdf.Lastaccess:16.04.2018`, September 2009.

[8] TinyOS Documentation Wiki. `http://tinyos.stanford.edu/tinyos-wiki/index.php/TinyOS_Documentation_Wiki`. Last access: 28.11.2017.

[9] Contiki: The Open Source OS for the Internet of Things. `http://www.contiki-os.org`. Last access: 01.12.2017.

[10] The friendly Operating System for the Internet of Things. `https://riot-os.org/`. Last access: 28.11.2017.

[11] The FreeRTOS Kernel. `http://www.freertos.org/`. Last access: 01.12.2017.

[12] Marcel Baunach. *Advances in Distributed Real-Time Sensor/Actuator Systems Operation*. Phd thesis, University of Würzburg, 2012.

[13] TÜV SÜD. Functional safety. `https://www.tuv-sud.com/activity/focus-topics/functional-safety`. Last access: 28.11.2017, 2017.

[14] IEC SC 65A. Functional safety of electrical/electronic/programmable electronic safety-related systems. Technical Report IEC 61508, The International Electrotechnical Commission, 3, rue de Varembé, Case postale 131, CH-1211 Genève 20, Switzerland, 1998.

[15] DO-178B. Software considerations in airborne systems and equipment certification. Technical Report DO-178B, Radio Technical Commission for Aeronautics, 2011.

[16] ISO/TC 22. Road vehicles – Functional safety. Technical Report ISO 26262, International Organization for Standardization, 2011.

[17] ARINC-653. `https://www.arinc.com`. Last access: 20.06.2017.

[18] AUTOSAR. `https://www.autosar.org`. Last access: 20.11.2017.

[19] Andrew S. Tanenbaum and Todd Austin. *Structured Computer Organization*. Prentice Hall, 6 edition, 2009.

[20] Anton Hattendorf, Andreas Raabe, and Alois Knoll. Shared Memory Protection for Spatial Separation in Multicore Architectures. In *Proc. of the 7th IEEE Int. Symposium on Industrial Embedded Systems (SIES)*, pages 299–302. IEEE, June 2012.

[21] Farid Shamani, Vida Fakour Sevom, Jari Nurmi, and Tapani Ahonen. Design, Implementation and Analysis of a Run-Time Configurable Memory Management Unit on FPGA. In *Proc. of the Nordic Circuits and Systems Conference (NORCAS): NORCHIP & International Symposium on System-on-Chip (SoC)*, pages 1–8. IEEE, Oct 2015.

[22] Micrium. uC/OS-III. `https://www.micrium.com/rtos/`. Last access: 01.12.2017.

[23] Adam Dunkels and Oliver Schmidt. Protothreads - lightweight, stackless threads in c. techreport, Swedish Institute of Computer Science, March 2005.

[24] Theodore P. Baker. A stack-based resource allocation policy for realtime processes. In *Proc. ot the 11th Real-Time Systems Symposium (RTSS)*, volume 11, pages 191–200. IEEE, dec 1990.

[25] Rony Ghattas and Alexander G. Dean. Preemption Threshold Scheduling: Stack Optimality, Enhancements and Analysis. In *Proc. of the 13th Real Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, April 2007.

[26] Chao Wang, Zonghua Gu, and Haibo Zeng. Global Fixed Priority Scheduling with Preemption Threshold: Schedulability Analysis and Stack Size Minimization. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 27(11):3242–3255, Nov 2016.

[27] Butler W. Lampson and David D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[28] Ragunathan Rajkumar, Liu Sha, and John P. Lehoczky. Real-time Synchronization Protocols for Multiprocessors. In *Proc. of the 9th IEEE Real-Time Systems Symposium (RTSS)*, pages 259–269. IEEE, Dec 1988.

[29] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems: Global Edition*. Prentice Hall, 4 edition, 2014.

[30] Liu Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. on Computers*, 39(9):1175–1185, Sep 1990.

[31] Marcel Baunach. Dynamic hinting: Collaborative real-time resource management for reactive embedded systems. *Journal of Systems Architecture*, 57(9):799–814, Oct 2011.

[32] Chia-Mei Chen and Satish K. Tripathi. Multiprocessor Priority Ceiling Based Protocols. Technical report, University of Maryland, 1994.

[33] Paolo Gai, Marco Di Natale, Giuseppe Lipari, Alberto Ferrari, Claudio Gabellini, and Paolo Marceca. A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. In *Proc. of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 189–198. IEEE, May 2003.

[34] Aaron Block, Hennadiy Leontyev, Björn B. Brandenburg, and James H. Anderson. A Flexible Real-Time Locking Protocol for Multiprocessors. In *Proc. of the 13th IEEE Int. Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 47–56. IEEE, Aug 2007.

[35] Motorola Inc. *M6800 Microprocessor Application Manual*, 1975.

[36] Frank M. Gruppuso. *Computer Strucures: Principles and Examples*, chapter PIC1650: Chip Architecture and Operation, pages 602–609. Hill Book, 1982.

[37] John Wharton. *An Introduction to the Intel MCS-51 Single-Chip Microcomputer Family, Application Note AP-69*. Intel Corporation, 1980.

[38] Texas Instruments. *MSP430x5xx and MPS430x6xx Family User's Guide*, 2014.

[39] Johann Wiesböck. Erfolgreicher 16-Bit-Controller aus Freising. `https://www.meilensteine-der-elektronik.de/erfolgreicher-16-bit-controller-aus-freising-a-521929/`. Last access: 24.11.2017, February 2016.

[40] Renesas Electronics K.K. SuperH RISC engine Family. `https://www.renesas.com/en-eu/products/microcontrollers-microprocessors/superh.html`. Last access: 28.11.2017. Accessed on 28.11.2017.

[41] ARM Limited. *ARM7TDMI Technical Reference Manual*, 2001.

[42] Atmel Corporation. *AVR Enhanced RISC microcontroller*, May 1996.

[43] Infineon Technologies AG. 32-bit TriCore Microcontroller. `https://www.infineon.com/cms/de/product/microcontroller/32-bit-tricore-microcontroller/`. Last access: 28.11.2017. Accessed on 28.11.2017.

[44] Xilinx Inc. *MicroBlaze Processor Reference Guide - UG981*, 2009.

[45] Altera Corporation. *Nios II Classic Processor Reference Guide*, 2016.

[46] Atmel Corporation. *AVR32UC Technical Reference Manual*, 2010.

[47] Chris Schläger. Keynote: The Impact of Operating Systems on Modern CPU Designs (and Vice Versa). In *Proc. of the 21st Int. Conference Architecture of Computing Systems (ARCS)*, Dresden, Germany, February 2008.

[48] APA. Google-hardware-chef: Keine rücksicht auf android-partner mehr. `https://derstandard.at/2000075104621/Google-Hardware-Chef-Keine-Ruecksicht-auf-Android-Partner-mehr.Lastaccess:21.04.2018`, February 2018.

[49] James R. Bulpin. Operating system support for simltaneous multithreaded processors. techreport, University of Cambridge, 2005.

[50] Lennart Lindh and Frank Stanischewski. FASTCHART - A Fast Time Deterministic CPU and Hardware Based Real-Time-Kernel. In *Proc. of the Euromicro '91 Workshop on Real-Time Systems*, pages 36–40. IEEE, 1991.

[51] Lennart Lindh. FASTHARD - A Fast Time Deterministic HARDware Based Real-time Kernel. In *Proc. of the 4th Euromicro Workshop on Real-Time Systems*, pages 21–25. IEEE, June 1992.

[52] Takumi Nakano, Andy Utama, Mitsuyoshi Itabashi, Akichika Shiomi, and Masaharu Imai. Hardware Implementation of a Real-time Operating System. In *Proc. of the 12th TRON Project International Symposium*, pages 34–42. IEEE, Nov 1995.

[53] Paul Kohout, Brinda Ganesh, and Bruce Jacob. Hardware support for real-time operating systems. In *Proc. of the 1st IEEE/ACM/IFIP Int. Conference on Hardware/ Software Codesign and Systems Synthesis (CODES+ISSS)*, pages 45–51. ACM, Oct 2003.

[54] Soon Ee Ong, Siaw Chen Lee, Noohul Basheer Zain Ali, and Fawnizu Azmadi B. Hussin. SEOS: Hardware Implementation of Real-Time Operating System for Adaptability. In *Proc. of the 1st Int. Symposium on Computing and Networking (CANDAR)*, pages 612–616. IEEE, 2013.

[55] Pramote Kuacharoen, Mohamed A. Shalan, and Vincent J. Mooney III. A Configurable Hardware Scheduler for Real-Time Systems. In *Proc. of the Int. Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 96–101. CSREA Press, 2003.

[56] Xilinx Inc. Zynq-7000. `https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`.

[57] Carl Stenquist. HW-RTOS Improved RTOS Performance by Implementation in Silicon. White Paper, May 2014.

[58] Renesas Electronics K.K. *R - IN32M3 Series User's Manual*, 2016.

[59] XMOS xCore Architecture. `http://www.xmos.com`. Last access: 01.12.2017.

[60] Theo Ungerer, Borut Robič, and Jurij Šilc. A Survey of Processors with Explicit Multithreading. *ACM Trans. on Computing Surveys (CSUR)*, 35(1):29–63, March 2003.

[61] David Koufaty and Deborah T. Marr. Hyperthreading Technology in the Netburst Microarchitecture. *IEEE Trans. on Micro*, 23(2):56–65, March 2003.

[62] Florian Kluge, Joerg Mische, Sascha Uhrig, Theo Ungerer, and Rafael Zalman. Use of Helper Threads for OS Support in the Multithreaded Embedded TriCore 2 Processor. In *Proc. Work-In-Progress-Session of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTSA)*, pages 1–3, 2007.

[63] John Fotheringham. Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store. *Communications of the ACM*, 4(10):435–436, October 1961.

[64] Brad Cohen and Ralph McGarity. The Design And Implementation of the MC68851 Paged Memory Management Unit. *IEEE Trans. on Micro*, 6(2):13–28, April 1986.

[65] ARM Limited. *Cortex-A5 MPCore Technical Reference Manual*, 2009.

[66] Ho-Cheung Ng, Yuk-Ming Choi, and Hayden Kwok-Hay So. Direct virtual memory access from FPGA for high-productivity heterogeneous computing. In *Proc. of the Int. Conference on Field-Programmable Technology (FPT)*, pages 458–461. IEEE, Dec 2013.

[67] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Stack Size Minimization for Embedded Real-Time Systems-on-a-Chip. *Trans. on Design Automation for Embedded Systems*, 7(1), 2002.

[68] Yun Wang and Manas Saksena. Scheduling Fixed-Priority Tasks with Preemption Threshold. In *Proc. of the 6th Int. Conference on Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 1999.

[69] ThreadX. `https://rtos.com/solutions/threadx/.Lastaccess:01.12.2017`.

[70] Erwin A. Hauck and Benjamin A. Dent. Burroughs' B6500/B7500 Stack Mechanism. In *Proc. of the Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 245–251. ACM, 1968.

[71] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable Threads for Internet Services. In *Proc. of the 9th ACM Symposium on Operating Systems Principles (SOSP)*, SOSP '03, pages 268–281. ACM, 2003.

[72] Dirk Grunwald and Richard Neves. Whole-program Optimization for Time and Space Efficient Threads. In *Proc. of the 7th Int. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS VII, pages 50–59. ACM, 1996.

[73] Mohamed Shalan and Vincent J. Mooney, III. Hardware support for real-time embedded multiprocessor system-on-a-chip memory management. In *Proc. of the 10th Int. Symposium on Hardware/Software Codesign (CODES + ISSS)*, CODES '02, pages 79–84. ACM, 2002.

[74] Rui Chu, Lin Gu, Yunhao Liu, Mo Li, and Xicheng Lu. SenSmart: Adaptive Stack Management for Multitasking Sensor Networks. *IEEE Trans. on Computers*, 62(1):137–150, Jan 2013.

[75] Sangho Yi, Seungwoo Lee, Yookun Cho, and Jiman Hong. *OTL: On-Demand Thread Stack Allocation Scheme for Real-Time Sensor Operating Systems*, volume 4490 of *Lecture Notes in Computer Science*, pages 905–912. Springer Berlin Heidelberg, 2007.

[76] Bhuvan Middha, Matthew Simpson, and Rajeev Barua. MTSS: Multitask Stack Sharing for Embedded Systems. *ACM Trans. on Embedded Computer Systems*, 7(4):41:1–46:37, August 2008.

[77] Marcel Baunach. CoMem: collaborative memory management for real-time operation within reactive sensor/actor networks. *Trans. on Real-Time Systems*, 48(1):75–100, 2012.

[78] David B. Stewart. Twenty-five-most commons mistakes with real-time software development. In *Proc. of the 1999 Embedded Systems Conference (ESC)*, September 1999.

[79] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: the mars approach. *IEEE Trans. on Micro*, 9(1):25–40, Feb 1989.

[80] Steve Kleiman and Joe Eykholt. Interrupts As Threads. *SIGOPS Trans. on Operating Systems Review*, 29(2):21–26, April 1995.

[81] Luis E. Leyva-del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable Interrupt Management for Real Time Kernels over conventional PC Hardware. In *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 14–23. IEEE, April 2006.

[82] Luis E. Leyva-del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable interrupt scheduling with low overhead for real-time kernels. In *Proc. of the 12th IEEE Int. Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 385–394, 2006.

[83] Luis E. Leyva-del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Integrated Task and Interrupt Management for Real-Time Systems. *ACM Trans. on Embedded Computing Systems*, 11(2):32:1–32:31, jul 2012.

[84] Daniel Stodolsky, J. Bradley Chen, and Brian N. Bershad. Fast interrupt priority management in operating system kernels. In *Symp. on the USENIX Microkernels and Other Kernel Architectures*, volume 4 of *moas'93*, pages 9–9, Berkeley, CA, USA, 1993. USENIX Association.

[85] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. SLOTH: Threads as Interrupts. In *Proc. of the 30th IEEE Real-Time Systems Symposium (RTSS)*, pages 204–213. IEEE, Dec 2009.

[86] OSEK/VDX - Operating System Version 2.2.3, February 2005.

[87] ARM Limited. *ARMv8-M Architecture Reference Manual*, 2017.

[88] Infineon Technologies AG. *Aurix TC29x B-step*, Mar 2014.

[89] Tiago Gomes, Paulo Garcia, Filipe Salgado, João Monteiro, Mongkol Ekpanyapong, and Adriano Tavares. Task-Aware Interrupt Controller: Priority Space Unification in Real-Time Systems. *IEEE Embedded Systems Letters*, 7(1):27–30, March 2015.

[90] Freescale Semiconductor Inc. *MC9S12XEP100 Reference Manual Covers MC9S12XE Family*, 2013.

[91] Fabian Scheler, Wanja Hofer, Benjamin Oechslein, Rudi Pfister, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Parallel, Hardware-supported Interrupt Handling in an Event-triggered Real-time Operating System. In *Proc. of the Int. Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, CASES '09, pages 167–174, New York, NY, USA, 2009. ACM.

[92] Infineon Technologies AG. *TC1797 32-Bit Single-Chip Microcontroller*, August 2014.

[93] Umakishore Ramachandran, Marvin Solomon, and Mary Vernon. Hardware Support for Interprocess Communication. In *Proc. of the 14th Annual Int. Symposium on Computer Architecture (ISCA)*, ISCA '87, pages 178–188. ACM, 1987.

[94] Jiun-Ming Hsu and Prithviraj Banerjee. A Message Passing Coprocessor for Distributed Memory Multicomputers. In *Proc. of the 1990 ACM/IEEE Conference on Supercomputing (SC)*, Supercomputing '90, pages 720–729. IEEE Computer Society Press, 1990.

[95] Johan Furunäs, Joakim Adomat, Lennart Lindh, Johan Starner, and Peter Voros. A Prototype for Interprocess Communication Support, in Hardware. In *Proc. of the 9th Euromicro Workshop on Real Time Systems*, pages 18–24. IEEE, Jun 1997.

[96] ARM Limited. *PrimeCell Inter-Processor Communications Module (PL320)*, 2004.

[97] Kelly Johnson. QorIQ Platform: Architecture Advantages. Presentation, October 2013.

[98] Sujaya Srinivasan and David B. Stewart. High Speed Hardware-assisted Real-time Interprocess Communication for Embedded Microcontrollers. In *Proc. of the 21st IEEE Conference on Real-time Systems Symposium (RTSS)*, RTSS'10, pages 269–279. IEEE Computer Society, 2000.

[99] ARM Limited. *ARM AMBA 5 AHB Protocol Specification*, 2015.

[100] Altera Corporation. *Avalon Interface Specifications*, 2015.

[101] OpenCores. *Wishbone B4 - WISHBONE System-on-Chip (SoC) Interconnection Architecturefor Portable IP Cores*, 2010.

[102] Xilinx Inc. *Processor Local Bus (PLB) Arbiter Design Specification*, 2002.

[103] ARM Limited. *AMBA Specification*, 1999.

[104] Anurag Shrivastava and Sudhir Kumar Sharma. Various Arbitration Algorithm for On-Chip(AMBA) Shared Bus Multi-Processor SoC. In *IEEE Students' Conf. on Electrical, Electronics and Computer Science*, pages 1–7. IEEE, March 2016.

[105] Ruchi Chandraker Bhawna Tiwari and Nidhi Goel. Comparative Analysis of Different Lottery Bus Arbitration Techniques for SoC Communication. In *Proc. of the Int. Conference on Computational Techniques in Information and Communication Technologies (ICCTICT)*. IEEE, March 2016.

[106] Hardik Shah, Andreas Raabe, and Alois Knoll. Priority division: A high-speed shared-memory bus arbitration with bounded latency. In *Proc. of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, March 2011.

[107] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy Barua. A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms. *Handbook on Scheduling Algorithms, Mehtods, and Models*, 2004.

[108] Chang L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM (JACM)*, 20(1):46–61, jan 1973.

[109] VLOGSyn Verilog Register Transfer Level Synthesis Working Group. IEEE Standard for Verilog Hardware Description Language. Technical Report ANSI/IEEE 1364-2005, Institute of Electrical and Electronics Engineers, New York, NY, USA, 2006.

[110] RISC-V Foundation. RISC-V. `https://riscv.org/`. Last access: 01.12.2017.

[111] MIPS. `https://www.mips.com/` . Last access: 27.03.2018. Accessed on 27.03.2018.

[112] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David Patterson, and Krste Asanovic. *The RISC-V Instruction Set Manual*, 2016.

[113] Fabian Mauroner and Marcel Baunach. mosartMCU: Multi-Core Operating-System-Aware Real-Time Microcontroller. In *Proc. of the 7th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–4, June 2018.

[114] Tobias Scheipel, Fabian Mauroner, and Marcel Baunach. System-Aware Performance Monitoring Unit for RISC-V Architectures. In *Proc. of the 20th Euromicro Conference on Digital System Design (DSD)*, pages 86–93, Aug 2017.

[115] Fabian Mauroner and Marcel Baunach. StackMMU: Dynamic Stack Sharing for Embedded Systems. In *Proc. of the 22nd IEEE Int. Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–9. IEEE, Sept 2017.

[116] Infineon Technologies AG. *TriCore V1.6*, May 2012.

[117] Fabian Mauroner and Marcel Baunach. CoStack: Collaborative Stack Sharing for Embedded Real-Time Systems. In *Proc. of the 13th Int. Conference on Systems (ICONS)*, pages 32–37, March 2018.

[118] Fabian Mauroner and Marcel Baunach. EventIRQ: An Event based and Priority aware IRQ handling for Multi-Tasking Environments. In *Proc. of the 20th Euromicro Conference on Digital System Design (DSD)*, pages 102–110, Aug 2017.

[119] Fabian Mauroner and Marcel Baunach. EventQueue: An Event based and Priority aware Interprocess communication for Embedded Systems. In *Proc. of the 13th Int. Symposium on Industrial Embedded Systems (SIES)*, pages 1–7. IEEE, March 2018.

[120] Fabian Mauroner and Marcel Baunach. Task Priority aware SoC-Bus for Embedded Systems. In *Proc. of the 19th Int. Conference on Industrial Technology (ICIT)*, pages 1453–1458. IEEE, February 2018.

[121] Fabian Mauroner and Marcel Baunach. Remote Instruction Call: An RPC approach on Instructions for Embedded Multi-Core Systems. In *Proc. of the 19th Int. Conference on Industrial Technology (ICIT)*, pages 1442–1446. IEEE, February 2018.

[122] Xilinx Inc. Artix7. `https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html`.

List of Figures

List of Tables

List of Abbreviation

ABI	Application Binary Interface
ACET	Average Case Execution Time
AHB	Advanced High-performance Bus
ALU	Arithmetic Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
AUTOSAR	AUTomotive Open System ARchitecture
CISC	Complex Instruction Set Computer
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CSA	Context Save Area
CSR	Control Status Register
DMA	Direct Memory Access
DoS	Denial of Service
DPRAM	Dual-Port-RAM
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing
EAS	Embedded Automotive Systems
ECB	Event Control Block
EDF	Earliest Deadline First
FF	Flip Flop
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
HAL	Hardware Abstraction Layer
iff	if and only if
IoT	Internet of Things
IP	Intellectual Property
IPC	Inter-Process Communication
IRQ	Interrupt Request
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
LIFO	Last-In First-Out
LSB	Least Significant Bit
LUT	Lookup Table
MCU	Microcontroller Unit
MMIO	Memory Mapped I/O
MMU	Memory Management Unit
MPU	Memory Protection Unit
NoC	Network on Chip
OS	Operating System
OS-PI	Operating System Priority Inversion
OSEK	Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
PC	Personal Computer
PLB	Processor Local Bus
PM	Physical Memory

PMU Performance Monitoring Unit

PP-LUT Page Pointer Look Up Table

QCB Queue Control Block

RAM Random Access Memory

RIC Remote Instruction Call

RISC Reduced Instruction Set Computer

RM Rate Monotonic

RPC Remote Procedure Call

RTOS Real-Time Operating System

SoC System-on-a-Chip

SRP Stack Resource Protocol

TCB Task Control Block

TDMA Time Division Multiple Access

TLB Translation Lookaside Buffer

VM Virtual Memory

WCET Worst Case Execution Time

WSAN Wireless Sensor/Actuator Network

WSN Wireless Sensor Network

Errata

"A person who never made a mistake never tried anything new."
—Albert Einstein

page 50: The equation 3.19 and the preceding lines are to be replaced by:

The set A_s contains all masters that want to access the slave s. With the help of the set $X := \left\{m \in A_s \mid \pi_m = \max_{\forall a \in A_s} \{\pi_a\}\right\}$, which contains all masters with the same highest priority in A_s that want to access the slave s, the accepted master α_s is selected by

$$\alpha_s := \begin{cases} \emptyset, & \text{if } X = \emptyset \\ x \in X \mid master\_id(x) = \min_{\forall \tilde{x} \in X} \{master\_id(\tilde{x})\}, & \text{else} \end{cases} \tag{3.19}$$