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

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)

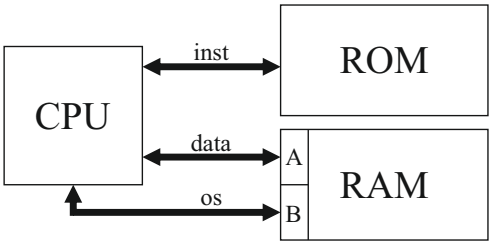

Fig. 1. The system architecture to support the EventQueue approach.

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 $32\,\mathrm{bit}$ 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.

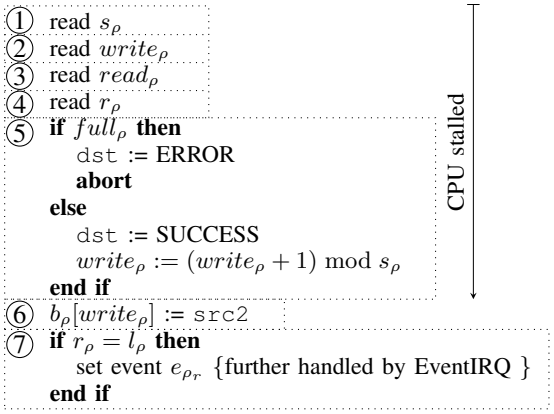

Fig. 2. Pseudo-code triggered by a `qwr dst, src1, src2` instruction.

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

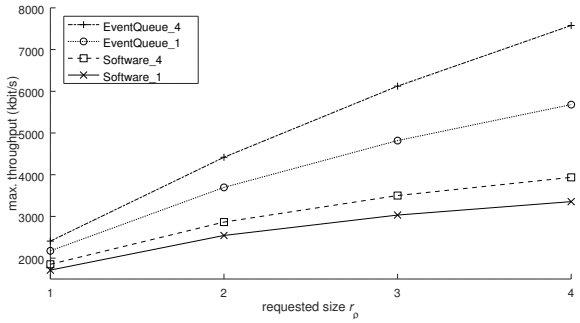

Fig. 3. Maximal throughput comparison of the pure software implementation and EventQueue with different configurations.

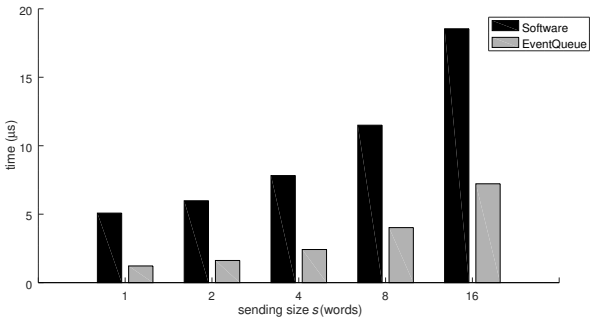

Fig. 4. Execution times of `send_queue_until(`ρ`,` d`,` s`,` t`)` with different number of sending size s.

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.