Virtualization of Reconfigurable Mixed-Criticality Systems

Cornelia Wulf [1], Najdet Charaf [1], Diana Göhringer [1,2]
[1] *Chair of Adaptive Dynamic Systems, Technische Universität Dresden, Germany*
[2] *Centre for Tactile Internet with Human-in-the-Loop (CeTI), Technische Universität Dresden, Germany*
{Cornelia.Wulf, Najdet.Charaf, Diana.Goehringer}@tu-dresden.de

***Abstract*— The increasing complexity of reconfigurable embedded systems often requires the integration of multiple applications with potentially different levels of criticality on the same hardware platform. As the deployment scales, there is a need for resource management, isolation, and performance that makes FPGA virtualization techniques a key consideration. FPGA virtualization enables multiple guest operating systems to run with different requirements, such as real-time, safety, or security. Most state-of-the-art systems incorporate mechanisms to strictly isolate subsystems in spatial respect at the expense of lower resource utilization. In this work, we present L4ReC, a microkernel-based virtualization layer that enables the sharing of reconfigurable resources among multiple virtual machines. The mapping and scheduling strategy for hardware threads considers not only deadlines, but also the real-time levels of guest operating systems. A POSIX thread-based interface facilitates the access to hardware accelerators. Compared with an existing scheduler for hardware threads, the average utilization factor - indicating the FPGA resource usage – is 1,9 times higher when threads are mapped and scheduled with L4ReC. Deadline misses are reduced by 3%.**

Keywords*— *FPGA virtualization, hypervisor, mixed criticality, real-time operating system

I. Introduction

Mixed Criticality Systems (MCS) incorporate components with different criticality levels. The criticality level specifies the level of assurance against failure. Hypervisors isolate these components in separated Virtual Machines (VM) and so prevent them from interfering with each other. They enable the joint execution of deterministic real-time operating systems (RTOS) for safety- or security-critical control systems with non-critical General-Purpose Operating Systems (GPOS), e.g., for the processing of large data amounts [1]. Examples can be found in the automotive domain, in avionics, or in robot-based I4.0 industrial plants [2]. Criticality-aware mapping to multicore processors is well researched ([3], [4]), whereas this is a rather new concept in the area of FPGAs. While the spatial separation of subsystems can lead to an under-utilization of hardware resources, the shared usage of FPGA area bears the risk to jeopardize isolation: Performance isolation ensures that applications in different VMs do not affect each other. Data isolation protects from malicious applications and untrusted users, improves maintainability and enables independent validation [5].

This paper presents L4ReC, which adds an FPGA virtualization layer to the micro-hypervisor L4Re [6] to enable the shared usage of hardware accelerators by multiple VMs. L4ReC provides a dynamic mapping and scheduling strategy that respects real-time levels (hard, soft, and no real-time) of hardware tasks. For hard real-time tasks, the missing of deadlines can lead to system failure, while for soft real-time tasks the usefulness of a result degrades after its deadline. The proposed strategy was inspired by the Robust Earliest Deadline (RED) algorithm [7] that was designed for a single core system to cope with overload conditions. Tasks with a lower real-time value are moved from the ready queue to the reject queue, if they deter the execution of a task with a higher real-time value. The RED-algorithm was implemented in hardware as an ASIC design [8]. None of these works considers the scheduling of hardware tasks.

The main contributions of this paper comprise:

- A mechanism that isolates hardware accelerators and protects them from malicious access.
- A dynamic mapping and scheduling strategy for hardware tasks considering different real-time requirements. In order not to jeopardize deadlines by Dynamic Partial Reconfiguration (DPR / DFX) latencies, hardware accelerators can be reused, prefetched, and reserved.
- A POSIX thread-based unified hardware-software execution model, which abstracts from the underlying hardware and offers a high-level programming interface.

This work is structured as follows: Related work is described in Section II. Section III gives an overview over L4ReC, while the mapping and scheduling strategy is depicted in Section IV. Its benefit as well as its overhead is evaluated in Section V. A conclusion complements this work in Section VI.

II. Related Work

A. Hypervisors for embedded reconfigurable computing

Hypervisors for embedded reconfigurable computing target the virtualization of FPGA resources and the acceleration of hypervisor functionality [9]. Janßen proposed a hypervisor that is executed as part of the hardware in order to improve performance and determinism [10]. In BlueVisor [11], the hypervisor is also implemented in hardware while guest operating systems run on softcore processors in a Network-on-Chip (NoC).

Hypervisors that are executed on a processor in an MPSoC often build upon a microkernel, as microkernels increase security by minimizing the Trusted Computing Base (TCB). Ker-ONE [12] is an extension of the microkernel Mini-NOVA. Hardware accelerators are mapped via interface components to the VM address space. In contrast to Ker-ONE, µRTZVisor [13]

offers full virtualization and offloads an Inter-Partition Communication (IPC) mechanism to hardware. µRTZVisor exploits the ARM TrustZone-hardware extension. Jain et al. [14] modified the microkernel CODEZERO. An intermediate fabric is overlaid on top of the FPGA fabric in order to mitigate the expense through DPR. SPHERE [15] distributes application execution between several FPGA SoC connected via a Time-Sensitive Network (TSN). The focus lies on time-predictable I/O virtualization, hardware task scheduling is not provided. SPHERE distinguishes different real-time levels for traffic flow.

All mentioned hypervisors for embedded reconfigurable systems offer mechanisms to control access to FPGA resources, but only Ker-ONE considers different real-time levels of hardware tasks. However, Ker-ONE supports only one guest RTOS and multiple GPOS. The hypervisor presented in this work is the only one that sustains several operating systems with multiple real-time levels.

B. *Scheduling techniques for hardware tasks*

Scheduling techniques for hardware tasks address the problem of efficient resource management on FPGAs. Several approaches are based on the Earliest Deadline First (EDF) scheduling policy (e.g., [16], [17], [18]). Saha et al. [19] present a static scheduling algorithm for safety critical periodic tasks and a dynamic algorithm for less critical aperiodic tasks, minimizing the reject rate of aperiodic tasks. Sehhatbakhsh et al. [20] propose a static scheduling strategy for mixed-criticality tasks that increases schedulability by mapping tasks to the Partially Reconfigurable Region (PRR) with the smallest normalized number of context switches. Guha et al. [21] propose a reliability driven hybrid scheduling approach.

As reconfiguration latencies impair real-time capacities, many scheduling policies aim to reduce DPR latencies. The authors of [22] investigate configuration prefetching and reuse for preemptive hardware multitasking. RACOS [23] adds an out-of-order scheduling that brings forward those hardware tasks that require an accelerator already available on the FPGA. The Run-Time System Manager (RTSM) [24] avoids DPR latencies by reusing hardware accelerators, reserving PRRs and prefetching bitstreams. The Reconfiguration-Dependency Non-Consistent Algorithm (RDNC-SA) [25] merges accelerators to the same PRR to improve area usage and to reconfigure several accelerators simultaneously.

None of the described related works considers the dynamic scheduling of hardware tasks with different real-time levels and the objective to not only meet all timing requirements but also to reach a high resource utilization.

III. Overview of L4ReC

A. *Architecture*

We propose a virtualization layer called L4ReC running on top of L4Re in order to enable virtualization of reconfigurable resources. The L4Re Runtime Environment is an operating system framework written in C++ for building systems with real-time, security, safety, and virtualization requirements [26]. L4Re builds upon the capability-based microkernel Fiasco.OC. It provides virtual CPUs (vCPUs), that are mapped to physical CPUs [27], and enables the creation and management of VMs.

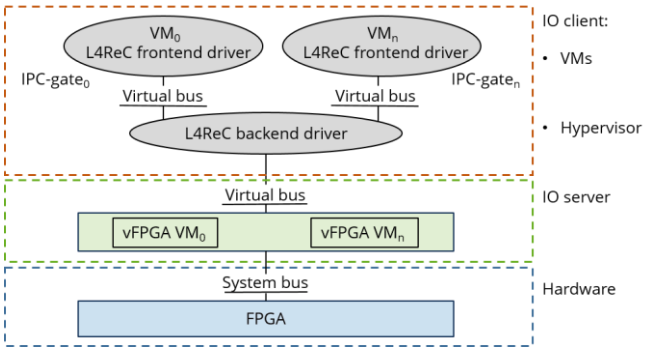

Fig. 1 Isolation mechanism

L4Re does not consider FPGAs. Our FPGA virtualization layer L4ReC offers virtual FPGAs (vFPGAs) as abstraction from physical FPGAs (Fig. 1). Virtual FPGAs represent the VMs' view of the FPGA, which can contain an arbitrary number of hardware accelerators and allows the use of accelerators independent from their actual location. L4ReC comprises a frontend and a backend driver that communicate via Inter-Process Communication (IPC). The frontend driver offers a POSIX thread API to the guest operating systems for accessing hardware accelerators. The backend driver enables communication as well as data buffering and contains the hardware thread scheduling and mapping mechanism. Both drivers are executed in a non-privileged processor mode (user mode).

B. *Isolation of FPGA resources*

For safety- and security critical applications, interference of threads has to be prevented when accessing shared hardware components. In L4ReC, software threads in VMs cannot directly access hardware accelerators. We use the I/O server that L4Re provides as a service to handle all platform devices and resources such as memory mapped I/O (MMIO) regions, interrupts, and bus numbers (Fig. 1). Each I/O client is attached to a virtual bus that aggregates a set of resources accessible to the client. The L4ReC backend driver obtains the right to access the complete MMIO resources and interrupts of the FPGA, while direct access to the FPGA is prohibited for VMs. Any application in a VM that tries to access the FPGA directly receives a page fault exception. The L4ReC frontend and backend drivers communicate via virtual busses, using IPC-gate capabilities for sending and receiving messages. IPC-gates allow to identify the sender of a message. To prevent unauthorized access to accelerators allocated to a different virtual machine, the hardware scheduler assigns an ID to each hardware thread that is a concatenation of the local thread ID and the respective IPC-gate ID.

C. *Programming model*

L4ReC extends the multi-threading programming model to hardware threads. Software and hardware threads are handled with a uniform POSIX thread-based API (Table 1). The standardized interface facilitates the deployment also for inexperienced users. L4ReC is not the first system that builds upon the multi-threading programming model: ReconOS [28] offers unified operating system services for functions executing in software and hardware. HybridThreads (hthreads) [29] realizes operating system tasks like scheduling and synchronization as finite state machines in hardware and so achieves a low-jitter solution with deterministic behavior. In contrast to both works,

TABLE 1 PTHREAD-BASED API

Functions from pthread.h: int pthread_create (hw_thread *, const pthread_attr *, void *(*)(void *), void *) int pthread_join (hw_thread, void **)
Additional functions: int pthread_reserve (hw_thread) int pthread_delete (hw_thread)

L4ReC supports periodic threads and adds further functionality, e.g., to mitigate DPR latency.

The functions *pthread_create()* and *pthread_join()* for creating as well as joining threads and reading their results have the same signature as the functions in the pthreads library. They only differ in the type *hw_thread* that stores all information necessary for mapping and scheduling the hardware thread, like the required hardware function, worst case execution time (WCET), deadline, period, and real-time level. Real-time levels denote hard, soft, and no real-time, but any finer subdivision is possible. The second parameter in *pthread_create()* contains attributes, the third a software function with the same functionality that is called in the case that the hardware thread cannot be scheduled to the FPGA. The last parameter serves for input respectively output parameters. Two additional functions are introduced: In order to mitigate DPR latency, hardware accelerators can be reserved in advance using the function *pthread_reserve()*. Next to aperiodic threads, periodic threads are supported as they represent a typical use case in FPGA-based systems. Periodic threads can be created and deleted at arbitrary points of time. All instances of a periodic thread use the same *hw_thread* variable. For periodic threads, a PRR is reserved for the execution of the next instance in order to minimize DPR latencies. The function *pthread_join()* frees memory only for aperiodic threads. We added the function *pthread_delete()*, that deletes all reservations of a periodic thread and frees memory dynamically allocated for input and output parameters.

IV. MAPPING AND SCHEDULING STRATEGY

A. Thread and resource models

Given a set $\Gamma = \{\tau_0, \ldots, \tau_n\}$ of hardware threads, each thread $\tau_i = (f_i, c_i, p_i, d_i, r_i)$ is characterized by its function f_i, running time c_i, period p_i, deadline d_i, and real-time level r_i. The function f_i is associated with a hardware accelerator and optionally with a software function that is called if the mapping to a hardware accelerator fails. The running time c_i comprises the WCET as well as communication time of the associated hardware accelerator. Execution times of hardware threads can depend on input parameters. For aperiodic threads, p_i is set to 0, and for no real-time threads, d_i takes the value INT_MAX.

Fig. 2 gives an overview of the data structures used in the proposed strategy. A *hw_thread* can be an instance of a hardware thread τ, disposing of a dynamically allocated *data_space* for input and output parameters. A *hw_thread* can also be a reservation τ^{res} for an instance of a hardware thread. A thread instance can take the states running, waiting or finished, while a reservation can only be waiting. Queues exist for ready, rescheduled, rejected, and finished threads. The finished queue serves as a buffer: When an application has read the results of its associated hardware thread, the thread instance is deleted from the finished queue.

Accelerators can be loaded to different PRRs. *Number_in* and *number_out* denote the number of input and output parameters to give a generic interface for all hardware functions. To perform DPR, the reconfigurable modules require a uniform interface. Therefore, we provide a wrapper for the hardware accelerators that specifies the necessary parameters (start signal, done signal, number of input respectively output parameters, input and output parameters).

B. Mapping and scheduling strategy

The outline of the L4ReC mapping and scheduling strategy was introduced in [30]. Hardware threads are assigned to PRRs. Either their execution can be started immediately or they are sorted into the ready queue of the respective PRR (Fig. 3). To avoid deadline misses because of high reconfiguration latencies, the number of reconfigurations is reduced by reusing previously downloaded partial bitstreams, by prefetching partial bitstreams prior to their usage, and by reserving accelerators for future use. Threads are primarily scheduled to a PRR that already contains the required accelerator. If the number of accelerators exceeds the number of PRRs, reconfiguration times are added to the respective running times c_i. Reservations provide information about accelerators needed in future.

In order to add robustness, a hardware thread is only scheduled to the ready queue, if the meeting of its real-time requirements can be guaranteed. A hard real-time thread must be able to finish before its deadline. For soft real-time threads, a coefficient D denotes the deadline tolerance, i.e., the maximum time that thread τ_i may execute after its deadline and still produce a valid result. If a soft real-time thread cannot be scheduled to a

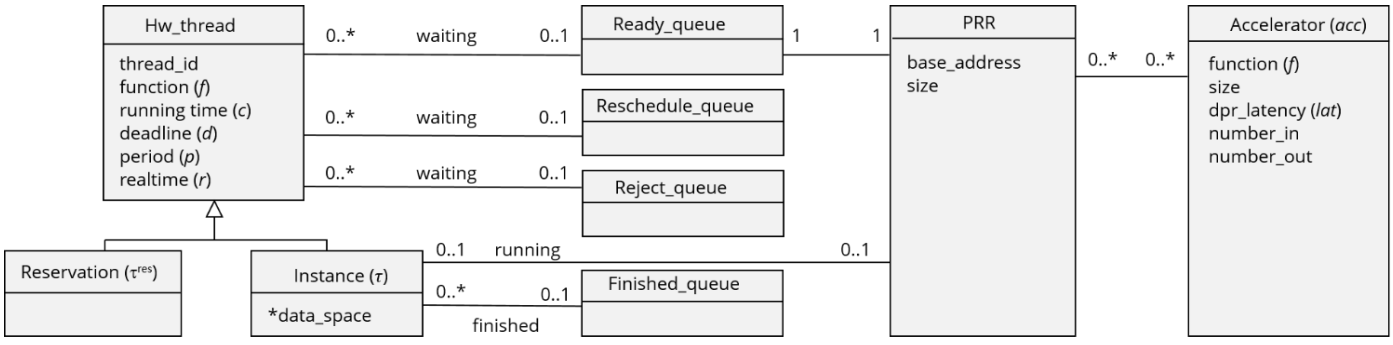

Fig. 2. Structure diagram

PRR that enables finishing before its deadline d_i, it is assigned to a ready queue that allows finishing before $D * d_i$ with a percentage D of d_i, $D > 1$.

While real-time threads and their reservations are ordered by their deadlines within each ready queue, non real-time threads are added into idle times. Idle times result from periodic threads that cannot start earlier than a multiple of their period value. If a thread cannot be scheduled to any ready queue without violating its real-time requirements, the L4ReC scheduler identifies suitable threads with a lower real-time value that can be replaced with the current thread. The replaced threads are removed to the reschedule queue, and the L4ReC scheduler tries to assign them to another ready queue. In systems with a high resource usage, this might not be possible, and the threads are passed on to the reject queue. Primarily, this should be non real-time threads. If real-time threads are pushed to the reject queue, the system is overloaded, which might lead to failure and should be avoided. The application in the respective VM receives an error message. If a software function was supplied in the *pthread_create()* call, the thread is removed from the reject queue and executed in software. Depending on the processor workload, the software function can still meet its deadline.

The function *next_thread()* (Fig. 3) is called whenever an accelerator has finished execution. The results are stored in the data space of the related thread in the finished queue. The function checks whether the thread from the head of the ready queue can be started. If this is a reservation and the associated thread has not arrived yet, either because its current period has not started or because previous software threads required more time than expected, unexpected idle times might occur, in which a suitable thread from the reject queue can be executed.

Algorithm 1 demonstrates the L4ReC scheduling algorithm. It determines the PRR in which to execute the current hardware thread. Either the thread is started immediately, or it is sorted into a waiting queue. Five possible scenarios are distinguished:

1. *A reservation for the thread exists (line 2):* The thread takes the place of its reservation. The execution is started, if the PRR is idle and the reservation was the head element. In case of a periodic thread, a further reservation is added. Testing the schedulability of a periodic task system is a (weakly) co-NP-hard problem [31]. For this reason, the schedulability is tested with the function *deadline_missed()* (line 9) only for a time interval which leaves enough time for a possible reconfiguration. In case of a failure, the reservation is pushed to the reschedule queue.

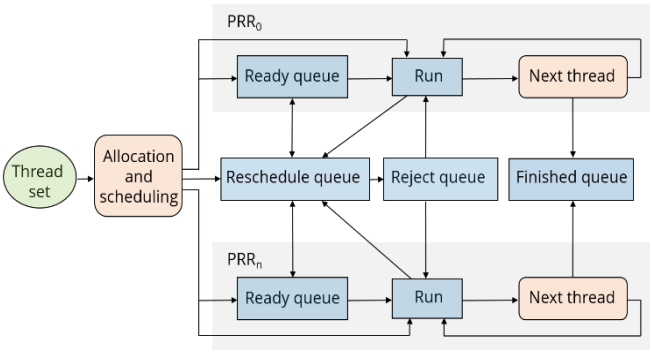

Fig. 3 Software architecture

ALGORITHM 1: L4REC SCHEDULING ALGORITHM

Input: hardware thread τ
Output: success

```
1  success = true
2  // 1. scenario: a reservation exists
3  if ∃ i: τ^res ∈ PRR_i.ready
4      PRR_i.ready.τ^res = τ
5      if PRR_i.running == NULL ∧ PRR_i.ready_0 == τ
6          start(τ, PRR_i)
7      if τ.p > 0
8          add τ^res to PRR_i.ready with τ^res.d += τ^res.p
9      if deadline_missed(PRR_i.ready)
10         reschedule_queue.push(τ^res)
11 // 2. scenario: the accelerator is loaded, enough time
12 else if ∃ i, j: PRR_i.acc_j.f == τ.f  ∧
13     ∃ idle time t in PRR_i.ready: t + τ.c ≤ τ.d
14     run_or_wait(τ, PRR_i)
15 // 3. scenario: an empty PRR exists
16 else if ∃ i: PRR_i.acc == NULL
17     acc = Accelerator with acc.f == τ.f
18     if (acc.lat + τ.c) > τ.d ∧ ∃ j:
19         τ_j.f == τ.f ∧ τ_j.r < τ.r ∧ τ_j.d ≤ τ.d
20             exchange position of τ_j in queue with τ
21     load acc via DPR
22     start(τ, PRR_i)
23     if τ.p > 0
24         create_reservation(τ)
25 // 4. scenario: overload situation without DPR
26 else if ∃ i, j: PRR_i.Acc_j.f == τ.f ∧ ∀ τ_k with τ_k.r < τ.r
27     ∧ τ_j.d*D ≤ τ.d: ∑ τ_k.c ≥ τ.c
28     while not (∃ enough idle time for τ in PRR_i.ready)
29         reschedule_queue.push(τ_k, τ_k^res)
30     run_or_wait(τ, PRR_i)
31 // 5. scenario: overload situation with DPR
32 else if ∃ PRR_i, k:
33     ∑ τ_k.c ≥ (acc.lat + τ.c) ∧ τ_k.r < τ.r ∧ τ_j.d*D ≤ τ.d
34     while not (∃ enough idle time for acc.lat + τ.c
35     in PRR_i.ready)
36         reschedule_queue.push(τ_k, τ_k^res)
37     run_or_wait(τ, PRR_i)
38 else
39     reject_queue.push(τ)
40     success = false
41 return success
```

2. *A suitable accelerator exists with enough idle time (line 11):* The function *run_or_wait()* is called that checks whether a running thread exists with a lower real-time value that has to be interrupted in order to guarantee deadline compliance of the current thread. In this case, the interrupted thread is pushed to the reschedule queue. The current thread is either started or inserted into the ready queue. If τ is a real-time thread, it is sorted by its deadline. Otherwise, τ is inserted at the first idle time long

enough to execute the thread. In case of a periodic thread, a reservation is created.

3. *An empty PRR exists (line 15):* The accelerator needed by the current thread has to be loaded via DPR. If DPR latency is so high that the deadline d_i is missed, the current thread is exchanged with a thread on another PRR with the same functionality, a lower real-time value and a finish time that matches d_i.

4. *Overload situation without DPR (line 25):* An overload situation is given, if those PRRs whose accelerator list contains the accelerator needed by τ, have not enough idle time to run τ. If suitable threads with lower real-time value exist, they are pushed to the reschedule queue to make space for τ.

5. *Overload situation with DPR (line 31):* The overload situation described in scenario 4 is mitigated by adding the accelerator needed by τ to the accelerator list of a suitable PRR. Now the reconfiguration time has to be added when searching for a suitable space to add τ.

If τ is a non real-time thread, the if-statements using τ.*d* have to be omitted in Algorithm 1. If no PRR can be found that provides enough idle time to execute τ, even when replacing all threads with a lower real-time value, none of the scenarios applies and τ is pushed to the reject queue.

V. Evaluation

The proposed FPGA virtualization was evaluated on a Xilinx Zynq Ultrascale+ MPSoC ZCU104 that is equipped with an ARM Cortex-A53 processor. The FPGA was clocked with a frequency of 100 MHz, the processor with 1.2 GHz. L4Re supports paravirtualization as well as full virtualization. Paravirtualized operating systems are aware of the underlying virtualization layer, resulting in an improved performance. For this reason, three paravirtualized FreeRTOS instances were chosen as guest operating systems. In order to simulate a mixed criticality system, each FreeRTOS instance was assigned a different real-time mode (hard, soft, and no real-time). On the FPGA, four PRRs were provided to host three different accelerators (matrix multiplication, encryption, and Fibonacci numbers). The Fibonacci core allows to choose a variable execution time: As the n^{th} Fibonacci number is determined in n+1 clock cycles, the order passed as a parameter specifies the number of clock cycles needed for the execution of the hardware thread.

A. Isolation and dynamic allocation

Ker-ONE is the hypervisor closest to our work. In Ker-ONE, hardware accelerators are mapped via interface components to the VM address space. The page table of each VM is manipulated in order to statically map an interface component to its address space. The interface components are dynamically connected to PRRs. When an interface component is not connected to a PRR, a writing access causes a page-fault exception. This mechanism guarantees the unique use of accelerators. While the Ker-ONE hypervisor module requires the execution in a privileged processor level, L4ReC runs in an unprivileged processor level, complying with the microkernel principle to execute everything in the lowest privilege level possible. Table 2 provides a comparison between important features of Ker-ONE and L4ReC.

TABLE 2 COMPARISON BETWEEN KER-ONE AND L4REC

	API	**Processor privilege level**	**Hardware task context switch**	**Mitigation of DPR latency**
Ker-ONE	Custom	Privileged level	Preemptive at consistency points	Reservation (but no sharing during idle times), reuse only for GPOS
L4-ReC	Pthread based	Unprivileged level	Cooperative	Reservation with sharing during idle times, reuse, preload

B. Mapping and scheduling strategy

Ker-ONE is the only hypervisor of those described in Section II, that considers the mapping and scheduling of hardware tasks, but next to several GPOS only one RTOS is allowed. The RTOS hardware tasks have the highest priority. GPOS hardware tasks have no deadlines and are therefore scheduled according to a priority-based round-robin strategy during idle times of the RTOS. As the execution of several RTOS is not supported, an accurate comparison with L4ReC is inappropriate. EDF is a commonly used scheduling strategy for real-time systems, so we compare our strategy with an EDF algorithm that does not differentiate between real-time levels, but uses the same thread reservation mechanism as L4ReC.

The FPGA utilization factor $U(\Gamma, \Delta t)$ serves as metric for the occupancy rate of the FPGA of a given thread set Γ during time interval Δt. All threads $\tau_{ij} \in \Gamma$ are considered that have finish times f_{ij} within Δt.

$$U(\Gamma, \Delta t) = \frac{\sum_{i \in PRR} \sum_{\tau_{ij} \in \Gamma} c_{ij}^{exe} + c_{ij}^{com} + b_{ij} * c_{ij}^{rec}}{\Delta t} \quad \forall \tau_{ij}: f_{ij} \in \Delta t \tag{1}$$

The running time of the j^{th} thread instance executed on the i^{th} PRR equals the sum of its execution time c_{ij}^{exe}, its communication time c_{ij}^{com}, and a binary variable b_{ij} times the reconfiguration time c_{ij}^{rec} of the accelerator requested by τ_{ij}. The binary variable b_{ij} is set to 0 if the previous thread running on PRR_i requires the same accelerator as the current thread:

$$b_{ij} = \begin{cases} 0, & \tau_{ij}.acc == \tau_{i\,j-1}.acc \\ 1, & \text{else} \end{cases} \tag{2}$$

The finish time f_{ij} is less than or equal to the deadline d_{ij} for hard real-time threads and $D * d_{ij}$ for soft real-time threads for a deadline tolerance $D > 1$. For non real-time threads, f_{ij} is the time a thread finishes when the scheduler placed it into an idle phase within the time interval Δt. In our test environment, the reconfiguration latency of the biggest accelerator amounts to 2,46 ms. In order to cover a wide range of possible hardware threads, we used synthetically generated thread sets with random values for c_i, p_i, and d_i. The period p_i was selected from the interval [30 ms, 200 ms], which allows a wide variability of execution times. Increasing the interval could lead to long threads blocking an accelerator resulting in an overload situation. We forced overload situations by increasing the number of threads. Deadlines d_i and execution times c_i were randomly chosen to lie within the given period. Execution times c_i were restricted to a minimum length of 10 ms. The threads inherit the real-time level of their VM, each VM is assigned a different real-time level.

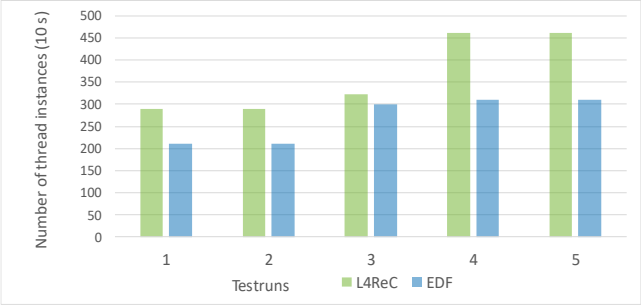

Fig. 4 Number of thread instances per 10 s

The number of thread instances that can be executed within a time interval depends on their period, deadline, and number of reconfigurations. Fig. 4 presents the number of thread instances that were executed during a time interval of 10 s. In this measurement, 50% of thread instances were aperiodic threads, 50% were instances of periodic threads. On average, L4ReC manages to execute 365 thread instances per 10 s, while EDF only reaches 268, which corresponds to an increase of 136%. L4ReC achieves a higher number of thread executions than EDF because it allows non real-time thread instances to be inserted into slack times $p_i - d_i$, as they are not bound to the order of deadlines in the ready queue. Fig. 5 presents the utilization factor $U(\Gamma, 1s)$ for thread sets with 6 to 21 periodic threads, evenly distributed between three VMs and leading to 80 - 139 thread instances for L4ReC respectively 39 - 64 thread instances for EDF. The average values of the utilization factor are marked with vertical lines. L4ReC reaches a utilization factor that is 1,93 times better than that of EDF. Significant differences between utilization factors of L4ReC and EDF result from the fact that EDF provides better values for thread sets with longer execution times while L4ReC exploits slack times.

The occurrence of overload conditions is presented in Fig. 6. The number of real-time thread instances in the reject queue corresponds to deadlines not met by the execution in hardware. These deadlines can still be met via an execution as a software function. Nevertheless, we did not investigate this as the execution time in software depends on the workload of the processor and the main focus of this work lies on the evaluation of the L4ReC algorithm. In Fig. 6, the real-time thread instances in the reject queue are subdivided into hard and soft real-time instances. The soft real-time instances were granted an extension of their deadline by D = 1,2. Thread sets with 6 threads meet all deadlines. On average, L4ReC reaches 97,3% of all deadlines in hardware, while the EDF algorithm accomplishes only 94,2%. In contrast to EDF, in L4ReC only 6,2% of all real-time thread instances in the reject queue are hard real-time threads.

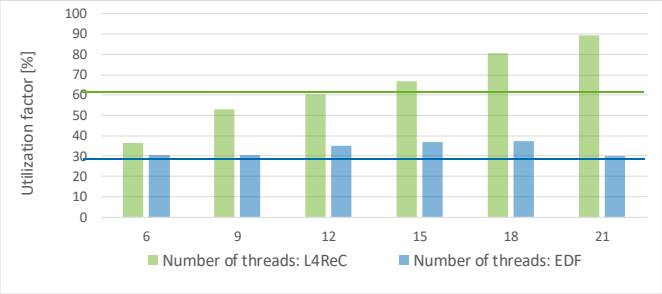

Fig. 5 FPGA utilization for differently sized thread sets

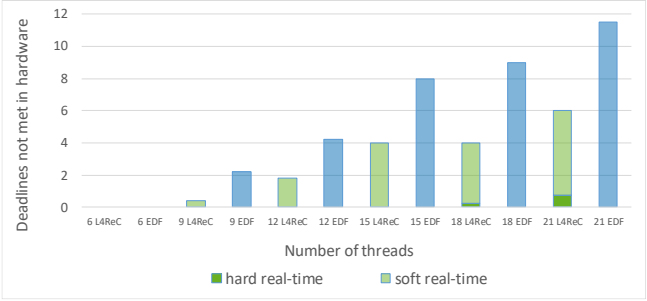

Fig. 6 Deadlines not met during hardware execution per second

C. Overhead analysis

To show the timing overhead induced by the L4ReC strategy, Fig. 7 gives an overview over the average clock cycles needed by the scenarios described in Section 4 and the function *create_reservation()*. The most frequent scenario, the search for a reservation, takes less than 50 clock cycles. The most cost-intensive operations are the insertion into a ready queue (scenario 2) and the creation of reservation thread instances.

VI. Conclusion

In this work, we presented the virtualization layer L4ReC. Its mapping and scheduling strategy aims at meeting all timing constraints of hardware threads and at the same time reaching a high resource utilization. The evaluation shows a considerable higher utilization factor and deadline compliancy compared to an EDF strategy that neglects real-time levels.

Several improvements are left for future work: In the current system, threads are executed in software if they cannot be scheduled in hardware. In a future version an execution in software can be favored if this leads to a higher throughput. Second, the cooperative scheduling scheme could be changed to a preemptive scheme so that threads that are interrupted need not be rescheduled in their entire length. Third, the system can be adapted to PRRs with different sizes. In this case, the smallest suitable PRR could be favored. Nevertheless, such a system limits the interchangeability of accelerators. Finally, as the Processor Configuration Access Port (PCAP) allows only sequential execution, the scheduling could be adjusting to the PCAP availability.

Acknowledgment

This work was funded by the German Research Foundation (DFG, Deutsche Forschungsgemeinschaft) as part of Germany's Excellence Strategy – EXC 2050/1 – Project ID 390696704 – Cluster of Excellence "Centre for Tactile Internet with Human-in-the-Loop" (CeTI) of Technische Universität Dresden.

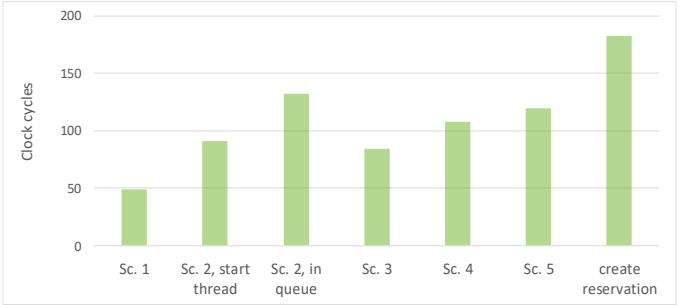

Fig. 7 Overhead of L4ReC scheduling algorithm

References

[1] H. Baek and J. Lee, "Incorporating security constraints into mixed-criticality real-time scheduling," *IEICE Trans. Inf. Syst.*, vol. 100, no. 9, pp. 2068-2080, 2017.

[2] J. Simó, P. Balbastre, J. F. Blanes, J.-L. Poza-Luján, and A. Guasque, "The role of mixed criticality technology in industry 4.0," *Electronics*, vol. 10, no. 3, p. 226, 2021.

[3] S. Groesbrink and L. Almeida, "A criticality-aware mapping of real-time virtual machines to multi-core processors," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, 2014: IEEE, pp. 1-9.

[4] V. Muttillo, G. Valente, and L. Pomante, *Criticality-aware Design Space Exploration for Mixed-Criticality Embedded Systems*. 2018, pp. 45-46.

[5] G. Heiser, "The role of virtualization in embedded systems," in *Open Kernel Labs and NICTA*, 2008: ACM Press, doi: 10.1145/1435458.1435461.

[6] Kernkonzept. "L4Re Technology." https://www.kernkonzept.com/l4re.html (accessed 23.06.2022).

[7] G. C. Buttazzo and J. A. Stankovic, "Adding robustness in dynamic preemptive scheduling," in *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems*: Springer, 1995, pp. 67-88.

[8] L. Kohútka, L. Nagy, and V. Stopjaková, "RED-based Scheduler on Chip for Mixed-Criticality Real-Time Systems," in *9th Mediterranean Conference on Embedded Computing (MECO)*, 2020: IEEE, pp. 1-4.

[9] C. Wulf, M. Willig, and D. Göhringer, "A Survey on Hypervisor-based Virtualization of Embedded Reconfigurable Systems," in *31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021: IEEE, pp. 249-256.

[10] B. Janßen, F. Korkmaz, H. Derya, M. Hübner, M. L. Ferreira, and J. C. Ferreira, "Towards a type 0 hypervisor for dynamic reconfigurable systems," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2017, pp. 1-7, doi: 10.1109/RECONFIG.2017.8279825.

[11] Z. Jiang, N. C. Audsley, and P. Dong, "BlueVisor: A Scalable Real-Time Hardware Hypervisor for Many-core Embedded Systems," in *24th IEEE Real-Time and Embedded Technology and Applications Symposium*, R. Pellizzoni Ed. New York: IEEE, 2018, pp. 75-84.

[12] T. Xia, Y. Tian, J.-C. Prévotet, and F. Nouvel, "Ker-ONE: A new hypervisor managing FPGA reconfigurable accelerators," *Journal of Systems Architecture*, vol. 98, pp. 453-467, 2019/09/01/ 2019, doi: https://doi.org/10.1016/j.sysarc.2019.05.003.

[13] J. P. A. Ribeiro, "A TrustZone-assisted hypervisor supporting dynamic partial reconfiguration," Diss., Universidade do Minho, 2018. [Online]. Available: http://hdl.handle.net/1822/64306

[14] A. K. Jain, K. D. Pham, J. Cui, S. A. Fahmy, and D. L. Maskell, "Virtualized Execution and Management of Hardware Tasks on a Hybrid ARM-FPGA Platform," *Journal of Signal Processing Systems*, Article vol. 77, no. 1-2, pp. 61-76, 2014, doi: 10.1007/s11265-014-0884-1.

[15] A. Biondi *et al.*, "SPHERE: A Multi-SoC Architecture for Next-Generation Cyber-Physical Systems Based on Heterogeneous Platforms," *IEEE Access*, vol. 9, pp. 75446-75459, 2021, doi: 10.1109/access.2021.3080842.

[16] H. Walder and M. Platzner, "Online Scheduling for Block-Partitioned Reconfigurable Devices," in *Proceedings of the conference on Design, Automation and Test in Europe* - Volume 1, 2003.

[17] K. Danne and M. Platzner, "An EDF schedulability test for periodic tasks on reconfigurable hardware devices," *ACM Sigplan Not.*, vol. 41, no. 7, pp. 93-102, 2006, doi: Doi 10.1145/1159974.1134665.

[18] C. Hong, K. Benkrid, X. Iturbe, A. Ebrahim, and T. Arslan, "Efficient On-Chip Task Scheduler and Allocator for Reconfigurable Operating Systems," *IEEE Embed. Syst. Lett.*, vol. 3, no. 3, pp. 85-88, 2011, doi: 10.1109/Les.2011.2167737.

[19] S. Saha, A. Sarkar, A. Chakrabarti, and R. Ghosh, "Co-Scheduling Persistent Periodic and Dynamic Aperiodic Real-Time Tasks on Reconfigurable Platforms," *IEEE Trans. Multi-Scale Comput. Syst.*, Article vol. 4, no. 1, pp. 41-54, 2018, doi: 10.1109/tmscs.2017.2691701.

[20] S. Sehhatbakhsh and Y. Sedaghat, "Scheduling Mixed-criticality Systems on Reconfigurable Platforms," in *9th International Conference on Computer and Knowledge Engineering (ICCKE)*, 2019: IEEE, pp. 431-436.

[21] K. Guha, A. Majumder, D. Saha, and A. Chakrabarti, "Reliability driven mixed critical tasks processing on FPGAs against hardware Trojan attacks," in *21st Euromicro Conference on Digital System Design (DSD)*, 2018: IEEE, pp. 537-544.

[22] A. Morales-Villanueva, R. Kumar, and A. Gordon-Ross, "Configuration Prefetching and Reuse for Preemptive Hardware Multitasking on Partially Reconfigurable FPGAs," in *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition*, New York: IEEE, 2016, pp. 1505-1508.

[23] C. Vatsolakis and D. Pnevmatikatos, "RACOS: Transparent access and virtualization of reconfigurable hardware accelerators," *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2017.

[24] G. Charitopoulos, I. Koidis, K. Papadimitriou, and D. Pnevmatikatos, "Hardware Task Scheduling for Partially Reconfigurable FPGAs," Cham, 2015: Springer International Publishing, in Applied Reconfigurable Computing, pp. 487-498.

[25] Z. Wang, Q. Tang, B. Guo, J.-B. Wei, and L. Wang, "Resource Partitioning and Application Scheduling with Module Merging on Dynamically and Partially Reconfigurable FPGAs," *Electronics*, vol. 9, no. 9, p. 1461, 2020.

[26] "L4Re Runtime Environment." https://l4re.org/ (accessed 23.06.2022).

[27] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig, "Flattening hierarchical scheduling," in *Proceedings of the tenth ACM international conference on Embedded software*, 2012, pp. 93-102.

[28] A. Agne, M. Platzner, C. Plessl, M. Happe, and E. Lübbers, "ReconOS," in *FPGAs for Software Programmers*, 2016, ch. Chapter 13, pp. 227-244.

[29] D. Andrews and M. Platzner, "Programming models for reconfigurable manycore systems," in *11th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2016: IEEE, pp. 1-8.

[30] C. Wulf, N. Charaf, and D. Goehringer, "Scheduling of Hardware Tasks in Reconfigurable Mixed-Criticality Systems," in *IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 15-18 May 2022, pp. 1-1, doi: 10.1109/FCCM53951.2022.9786181.

[31] F. Eisenbrand and T. Rothvoß, "EDF-schedulability of synchronous periodic task systems is coNP-hard," in *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, 2010: SIAM, pp. 1029-1034.