Appears in: Proceedings of the 24th International Symposium on Real-Time Distributed Computing (ISORC'21), **Authors' Version**

Taming Non-Deterministic Low-Level I/O: Predictable Multi-Core Real-Time Systems by SoC Co-Design

Steffen Vaas*, Peter Ulbrich†, Christian Eichler*, Peter Wägemann*, Marc Reichenbach* and Dietmar Fey*

* Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
firstname.lastname@fau.de

† Technische Universität Dortmund
peter.ulbrich@tu-dortmund.de

***Abstract*—Predictable and analyzable I/O is one of the considerable challenges in the design of multi-core real-time systems. A common approach to tackle this issue is to partition and schedule I/O transactions such that interference between tasks is minimized. While this works for packet-oriented interfaces with deterministic blocking times, such as ethernet, these techniques are inapplicable to a whole range of I/O devices with non-deterministic behavior that is commonly found in embedded applications. Interfaces, such as SPI, do not allow for fine-grained scheduling and thus exhibit uncontrolled blocking times. Even worse, their configuration and use must be considered as independent transactions requiring costly synchronization between tasks. The resulting detrimental effects are, in particular, pronounced in settings with mixed task requirements on predictability and determinism. All this makes the temporal analysis of such systems cumbersome and overly pessimistic.**

To solve these issues, we present $LOW_{I/O}$, an approach to eliminate the interference of low-level non-deterministic I/O interfaces for real-time tasks with high predictability demands (i.e., critical task) while preserving flexibility for tasks with lower requirements (i.e., uncritical tasks). Therefore, we leverage knowledge about the application-specific I/O usage patterns, obtained by static analysis, to derive a tailored hardware architecture. Its key feature is the anticipatory reservation of individual time slots for critical tasks and to mimic preemptivity of I/O units for the remaining system. We have implemented our approach as a toolchain for OSEK-based real-time systems that automatically generates an application-specific SoC design along with a hardware and timing model for subsequent WCET analysis. Our experimental results prove predictable timing for critical tasks with limited impact on uncritical tasks.

I. Introduction

One of the primary challenges in real-time systems' design is determining the worst-case execution time (WCET) and latencies of tasks. In this context, the execution platform's complexity and predictability are decisive as they ultimately impact analysis accuracy and resulting overapproximation. Consequently, this is a vivid research area in which much progress has been made to leverage the potential of modern multi-core architectures in hard real-time settings [1]–[3]. For example, memory and caches can be tamed by smart scheduling or partitioning. The situation is very similar to another fundamental shared resource: communication and interaction with the physical world by means of I/O interfaces. In many respects, the same concepts of partitioning and scheduling can be applied here [4], [5]. This partitioning is intuitively plausible for packet-switched communication (e.g., ethernet), where larger transactions are composed of individual packets [6]. If memory is predictable, then the (memory mapped) access to a network adapter is predictable as well [7]–[9]. Building on the packet-orientation, appropriate scheduling can be employed to facilitate real-time communication even for concurrent tasks. We will elaborate on all this related work in Section II.

In this paper, we focus on a class of low-level communication and periphery that is often neglected but ubiquitous in a wide range of embedded real-time systems: *low-level, transaction-based I/O with variable transaction length*. By this, we mainly refer to low-level serial communication interfaces (i.e., I^2C [10], SPI [11], RS232/485 [12]) for interfacing with sensors, actuators, or tertiary storage as well as general-purpose I/O used for signal analysis and generation. As for any periphery, the available I/O units and pins constitute the limiting factor while the availability of embedded multi-core platforms fosters the colocation of application and thus sharing I/O between tasks. For example, embedded platforms rarely offer more than two SPI units, leaving multiple sensors and actuators attached to a single bus. Likewise, a limited number of signal analysis and generation (e.g., ADC/DAC) units is multiplexed to a larger number of I/O pins. We argue that low-level I/O will continue its importance in interfacing peripherals while increasing application complexity implies their sharing.

A. Problem Statement

The fundamental problem with low-level peripherals is their poor predictability once they become a shared resource. Figure 1 illustrates this effect by the example of a SPI bus master with three devices. The previously mentioned concepts for (re)obtaining the predictability fail of this shared low-level periphery for the following reasons: (1) Transactions are non-preemptable, that is, an allocation once granted to one of the tasks cannot be revoked. However, unlike packet-switched communication, the transaction size is variable, application-dependent (e.g., message size), and very slow compared to computation. Specifically, on our target platform used for the

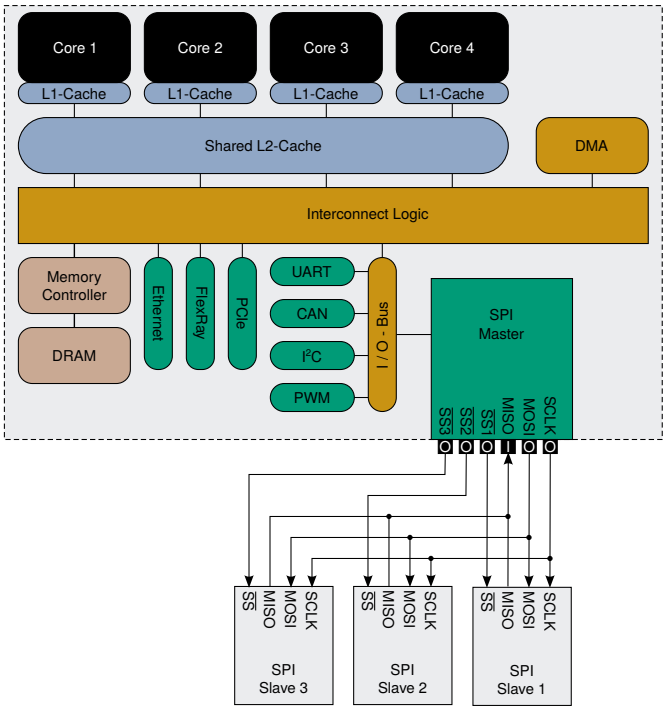


Figure 1: Peripheral devices are often connected to a shared bus. The specific characteristics of buses (e. g., two-stage and transaction-based communication) demand for hardware support to guarantee timeliness of critical tasks.

evaluation (see Section V), transmitting a single byte requires 1,600 processor cycles on the SPI bus. (2) The configuration (e.g., multiplexing and addressing) is decoupled from the actual data, that is, a transaction involves a two-step process: First, the respective peripheral unit must be configured for the correct multiplexer setting or device ID. Second, the actual transmission is initiated by reading or writing data. Consequently, the assumption that communication can be abstracted by a sequence of self-contained and freely schedulable entities (i.e., memory accesses) is inapplicable to most low-level peripherals; thus, deriving sound WCET bounds on blocking times quickly becomes infeasible.

In many cases, application complexity is accompanied by varying predictability demands of individual tasks and I/O accesses. Consequently, a recognized approach to minimize unnecessary overestimates is to subdivide the workload into tasks of low and high criticality (i. e., safety-critical or convenience). In such mixed-criticality settings [3], [13], the analysis pessimism can usually be hidden by adaptive scheduling: in the rare event that a task exceeds its average budget, the uncritical workload is preempted and deferred to guarantee sound estimates of critical tasks. Again, the indivisibility and irrevocability of the transactions disallow the operating system to offer meaningful abstractions to implement such an approach for low-level I/O: if a task puts a too large message on the SPI bus, a deadline violation of a critical message may be unavoidable.

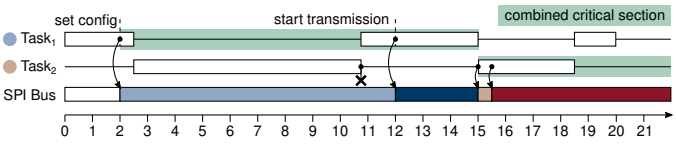


Figure 2: Communication over the low-level SPI bus is a two-stage transaction. First, the configuration is transmitted and, subsequently, the data.

B. Contribution

Starting from the idea of varying task requirements on the predictability (e.g., analysability, latencies, jitter) of low-level I/O, we address the problem in a slightly different way. Since the considered I/O transactions are indivisible, we propose a tailored partition into a predictable part with static allocation and one with dynamic schedulability. Therefore, we leverage the availability of FPGAs in contemporary embedded SoCs [14] to provide a tailored hardware architecture that facilitates high predictability for critical tasks while maintaining full flexibility to share the peripherals with uncritical tasks.

This approach, named $LOW_{I/O}$, is based on and extends R2-D2 [15], [16], an existing technique for co-design of heterogeneous real-time SoC architectures. Here, critical tasks are allocated to dedicated, deterministic computing cores while uncritical tasks are executed on a powerful but indeterministic multi-core processor. To extend this selective predictability to low-level I/O, we identified the following challenges:

Challenge #1: Predictable Low-Latency I/O. Considering I/O transactions as critical sections, a sound upper bound is required for all transactions (high and low) to derive the resulting blocking times. Still, the latter is at least as long as the largest of all sections causing jitter and jeopardize deadline adherence. Reconsider, for instance, the previous example of critical sensors with short but frequent communication and an uncritical tertiary memory with few long transactions. In the absence of preemptivity, time-division multiple access (TDMA) is an adequate solution that, however, fails due to priority-driven scheduling of uncritical tasks and their unpredictability communication behavior.

Our Approach: To reduce access latency to zero for critical transactions, we propose an anticipatory reservation of I/O. In other words, sporadic low-criticality transactions may only be allowed to occupy I/O if they do not conflict with critical transactions during their execution. For this purpose, we introduce *IO-Guards*, dedicated hardware units that coordinate the access to individual I/O units among the different cores. These IO-Guards utilize the critical tasks' access patterns determined by static analysis to isolate critical and uncritical accesses, without the need for more in-depth knowledge or sound upper bounds on low-criticality tasks and transactions.

Challenge #2: Two-stage Transactions (i.e., configuration). Another problem arises from the two-stage nature of the I/O transactions, which is illustrated in Figure 2. The timing-related problem here is that configuration and the actual data transfer can either be combined to a single critical section,

which further aggravates the analysis problem or is seen as two separate actions. However, the latter requires that periphery configuration, analogous to the processor context, must be accounted to the currently running task. We are unaware of existing operating-system support and preemption mechanisms to support such an extended context switch. Moreover, it would be extremely inefficient to implement in software due to the often high number of configuration registers. Even more severe: On real-world hardware platforms, read-only configuration registers exist. Such read-only registers can never be restored by software, but require dedicated hardware support.

Our Approach: Our systematic isolation of the individual peripheral units by IO-Guards at their interface with the I/O bus allows for efficient preemption of their configuration. We achieve this by employing *scan chains* and *shadow registers* for context saving.

Challenge #3: Platform Co-Design and System Integration. Besides the isolation and coordination mechanisms provided by the IO-Guard concept, the actual co-design of a given real-time application and architecture is a major challenge. Manually partitioning the available resources on an SoC-based platform is a labor-intensive task and, therefore, requires a high degree of automation.

Our Approach: In the $LOW_{I/O}$ approach, we use an integration of a code-analysis framework along with a framework for tailored hardware generation. For the software-related aspects, the static analysis framework reveals all communication patterns. This tooling support allows us to partition hardware resources in a way that guarantees interference-free communication of critical tasks, while uncritical tasks are scheduled with a best-effort strategy.

II. Related Work

The topic of tailored SoC platforms for guaranteed timeliness of I/O operations is well-explored in the real-time–systems community. However, to the best of our knowledge, no existing hardware-tailoring approach targets low-level communication buses (such as I^2C or SPI) that are shared between highly critical tasks and tasks of lower criticality, while the communication transactions are non-preemptable. Regarding the research on communication and shared resources in SoC-based systems, we refer to the review on mixed-criticality real-time systems [3].

Pellizzoni et al. [17]–[20] addressed numerous hardware-related aspects in multi- and many-core (FPGA-based) real-time systems, such as I/O transactions, resource allocation/arbitration, and interferences with peripheral components. However, none of these works addresses the unique properties in low-level communication buses, such as the two-stage communication procedure consisting of a configuration and an atomic transmission phase.

Due to the assumption of non-preemptive I/O transactions with variable length, our presented work does not benefit from existing work on (mixed-criticality) memory controllers [7]–[9], [21], [22]. Communication with memory-mapped operations allows preemptions during the use of the memory bus.

In the context of I/O for mixed-criticality systems, existing works are subdivided into TDMA-based [6], [23]–[27], virtualization-based [4], [23], and TrustZone-based communication systems [28]. $LOW_{I/O}$ chooses the approach of TDMA-based communication to guarantee the timely completion of highly critical I/O transactions. Similar to $LOW_{I/O}$'s approach, Cilku et al. presented a TDMA-based method with a two-layer arbitration scheme [23]. In their work, the first layer targets critical tasks, whereas uncritical I/O communications are executed on the second layer based on a round-robin scheme. However, this approach only targets the memory bus in contrast to $LOW_{I/O}$, with its focus on low-level communication interfaces. A further drawback of a round-robin scheme is that such an approach potentially leads to scenarios where time slots for low-criticality tasks remain unused. To mitigate this issue of low bus utilization, $LOW_{I/O}$ makes use of these slots by possibly reordering the workloads. Additionally, $LOW_{I/O}$ has the possibility to inform the system developer about response times of messages based on the assumption that the low-criticality estimates (i. e., C^{LO}) do not overshoot the given budget during runtime. In any case, $LOW_{I/O}$ guarantees the completion of high-criticality tasks by its hardware design.

Chip manufacturers have also recognized the problem and provide an individual, lightweight MPU for each peripheral [29]. Although this prevents unauthorized access by uncritical tasks, this mechanism does not allow anticipatory reservation of non-preemptive I/O transactions.

Several researchers address the aspect of exploiting statically determined knowledge for tailoring hardware platforms [30], [31]. $LOW_{I/O}$ similarly utilizes application-specific properties for the goal of predictability on the level of embedded communication buses. Our previous work in the area of exploiting application-specific knowledge introduced the concept of deterministic execution units (DEUs) [15], [16] in systems with different criticality levels. These units are the platform of highly critical tasks where (due to the design with reduced complexity) static timing analyses are possible with a comparably small degree of pessimism. DEUs are timing-anomaly–free processors with the property of timing compositionality. At the same time, $LOW_{I/O}$ does not restrict the performance cores for executing non-critical tasks in any way. Our previous work uses the concept of direct communication links between critical tasks in the system. Both the deterministic execution units and the direct links are achieved with the SoC-based design approach. In contrast to these existing techniques, $LOW_{I/O}$ is the first approach that tackles low-level communication interfaces with a similar strategy of exploiting SoC platforms and static analyses.

Despite the huge body of related work, no existing hardware-tailoring approach targets low-level communication buses (i. e., I^2C, SPI) that are shared between highly critical tasks and tasks of lower criticality. A major differentiation is that $LOW_{I/O}$ targets non-preemptable I/O operations with separated configuration and data-transmission phases.

III. System Model & Background

The following Section III-A first outlines our system model, including the assumptions on the targeted applications, and the subsequent Sections III-B/III-C provide necessary background information on the infrastructure used for whole-system analyses and hardware generation.

A. System Model

Our approach demands three fundamental properties from the real-time system: (1) For the tasks with high criticality, all allocation and scheduling-relevant system objects (threads, ISRs, resources, etc.) and their configuration are known ahead of time; either provided by some configuration file or statically extractable from the source code. (2) Precedence constraints and dependencies are explicitly implemented or modeled as dedicated tasks and resources. (3) A deterministic scheduling policy, such as fixed-priority preemptive scheduling.

Without loss of generality, we based our approach on the system model mandated by the AUTOSAR/OSEK-OS standard [32]. This standard defines a widely used class of fixed-priority real-time systems and has been the dominant industry standard for automotive applications for the last two decades. For a specific application, the developer declares all system objects and their parameters in a domain-specific configuration file, written in OSEK's interface language (OIL).

Furthermore, we make the following assumptions about the applications and the resulting taskset: (1) The latter consists of critical and uncritical tasks with high and low demands on predictability and timing-analysis accuracy, respectively. For the critical tasks, safe (and possibly pessimistic) worst-case execution-time estimates are available (i. e., C^{HI}), while uncritical are scheduled based on an optimistic timing estimate (i. e., C^{LO}). (2) For a given FPGA, the combined computational load cannot be met by deterministic hardware cores alone. (3) The predominant pattern of synchronization between critical tasks is either a producer/consumer relationship or mutual exclusion. In this relationship, all communication is issued explicitly via system calls, and thus the communication patterns can be extracted from static analysis. (4) Critical tasks do not depend on uncritical tasks. Regarding our terminology in view of mixed-criticality systems (MCS), $LOW_{I/O}$ expects settings with varying predictability demands and droppable workload. Our approach guarantees high-critical transactions and aborts uncritical transactions if C^{LO} is exceeded. (i. e., does not prevent overloading the available non-critical bandwidth). $LOW_{I/O}$ supports multiple levels by its runtime reconfigurability; however, only the highest-level guarantees predictability.

Motivating Example Applications: Examples of applications that are suitable targets for the $LOW_{I/O}$ approach are robotic arms (to support surgeries) or prosthetic hands [33]. Due to the goal of reducing cables in such cyber-physical systems to mitigate the risk of cable break, the sensors in these scenarios are ideally connected via one shared SPI bus. Further reasons for a shared SPI bus are that only a single SPI bus is available in the target platform or that SPI cables need to be reduced due to space limitations. In the example applications, I/O messages for position control are of higher criticality, and their timely completion needs to be guaranteed. In contrast, messages for the systems' condition monitoring [34] are of minor criticality, since degradation processes due to aging in such systems undergo a slower process. Thus, missing a few of such messages is comparably uncritical, and scheduling their related tasks with best effort is a suitable strategy. A dual-criticality approach is sufficient for the design of such systems where usually only few highly critical tasks exist. All mentioned design constraints are supported by the $LOW_{I/O}$ approach with regard to low-level communication.

B. Whole-System Analysis with ABB Graphs

In this Section, we outline existing work to derive predictable hardware architectures that are tailored to the applications' specific requirements from existing implementations. For an utmost generic approach, it is necessary to infer I/O usage and message length independently of the actual software development process. This necessitates a comprehensive toolchain spanning from software analysis to hardware tailoring.

In the first step, we employ the Real-Time Systems Compiler (RTSC) [35] to decompose the system's source code by static analysis into *Atomic Basic Blocks* (ABBs) [36], [37], which gives us a platform-agnostic intermediate representation of the system. Internally, the RTSC exploits the LLVM compiler infrastructure with its intermediate representation (LLVM IR) [38]. ABBs have the property of being single-entry single-exit regions of code. A system call in the code constitutes a terminator of an ABB. Regarding the I/O transactions of tasks, this means that communication patterns are explicitly expressed in the source code and, as a consequence thereof, are available to the static analysis for generating the hardware support for I/O on the SoC platform.

Based on the system's representation as an ABB graph, $LOW_{I/O}$ generates both a tailored timing model (see Figure 3) as well as an FPGA bitstream describing the hardware. We show the process of synthesizing application-specific multi-core hardware platforms in Section III-C.

Finally, the machine model obtained from this synthesizing process is fed back to the compiler and the WCET analyzer. Thereof, the RTSC generates the target binaries, whose WCETs are derived from the platform's specific properties.

C. Automated Hardware Generation

As described in the R2-D2 approach [15], [16], [39], existing system definition files, as OSEK OIL files, include all necessary information to derive application-specific hardware architectures that are highly improved towards predictability. For each critical task defined in the system description, a dedicated microcontroller subsystem, a deterministic execution unit (DEU), gets instantiated. All DEUs contain a highly predictable processor, such as an ARM Cortex-M0 or Cortex-M3, for which static WCET-analysis support is available [40], [41]. Furthermore, DEUs own an internal SRAM, an exclusive

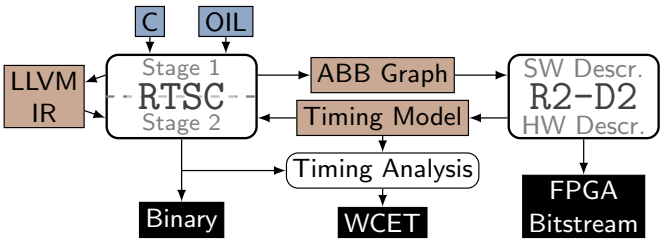


Figure 3: The integration of a static code-analysis framework (RTSC) and a hardware generation framework (R2-D2) allows for a tailored target platform that guarantees timeliness of communication over shared low-level buses.

AXI4-Lite bus system, and standard peripheral units, such as timers. The exclusive execution of a single task in combination with their isolated hardware structure makes the execution of the critical tasks highly predictable. Moreover, DEUs are customized for the corresponding critical tasks. For example, the memory internal to the DEUs is tailored to a reduced but sufficient size. The flexibility of an application-specific hardware design also provides the possibility to add further accelerator cores or dedicated peripheral controllers, if a critical task owns static, exclusive access. The hardware structure made up of several isolated DEUs prevents interferences between tasks running in parallel by hardware design. For communication between critical tasks on the DEUs, dedicated communication channels, which are implemented as physical direct links between the DEUs, are instantiated for each pair of DEUs.

$LOW_{I/O}$ extends this strategy by taking information about shared access to peripheral devices into account. As shown in Figure 4, additional hardware elements get instantiated to maintain the isolated structure of the computing units while providing controlled access to the shared peripherals.

IV. Approach

This Section gives details on the $LOW_{I/O}$ approach: Section IV-A introduces an example of a hardware platform. Section IV-B highlights how $LOW_{I/O}$ ensures selective guarantees by means of IO-Guards, which enforce anticipatory I/O reservations. Splitting the configurations and transactions is a major concern of $LOW_{I/O}$ and discussed in IV-C. $LOW_{I/O}$ demands for minimal code changes in existing applications (see Section IV-D). The final outcome of $LOW_{I/O}$ is a timing model on instruction level, which eventually yields precise timing bounds by the use of WCET-analysis tools (see Section IV-E).

A. Running Example

Figure 4 shows our running example of a system with two deterministic single-core processors (i. e., DEUs running critical tasks in a time-triggered manner), an ARM Cortex-A9 dual-core (COTS) processor (i. e., running uncritical tasks with priority-based scheduling), and a shared SPI controller. Except for the hardwired A9, the entire SoC architecture was synthesized with $LOW_{I/O}$ based on a static code analysis

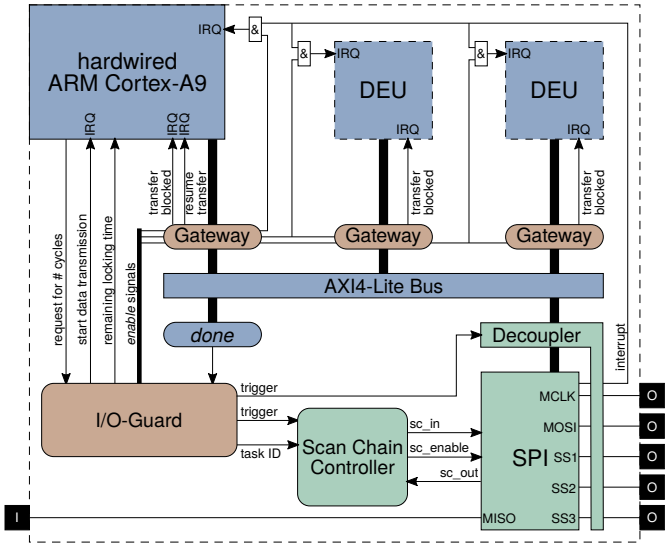


Figure 4: Running example of an application-specific system with two DEUs and one shared SPI, generated by $LOW_{I/O}$.

of the critical tasks. In this example, all processors require access to the SPI and are thus linked by a dedicated *AXI4-Lite*. To achieve the SPI's intended predictability, $LOW_{I/O}$ has generated an IO-Guard along with its associated infrastructure to control bus configuration and access. In this example, the SPI was run at 2 MHz and the DEUs operated at 100 MHz. Furthermore, we assume upper bounds on the message length and the execution time of the critical tasks. The transaction sizes generated by the non-critical tasks are assumed to be schedulable and do not lead to a bus overload on average.

B. Selective Predictability by Anticipatory I/O Reservations

At the heart of our approach is the *IO-Guard*, whose primary purpose is the arbitration of physical access times to a shared low-level peripheral according to varying predictability demands. The underlying mechanism is the anticipatory reservation of communication windows for critical tasks executed in a time-triggered manner on the DEUs.

At design time, $LOW_{I/O}$ leverages the functional (e. g., access patterns) and temporal information (e. g., WCET, maximum transaction size) inferred by our static analysis to tailor and synthesize one IO-Guard for each shared peripheral unit. This process results in the structure shown in Figure 5: (1) a *schedule table* to hold the next access window of each critical task, (2) a *gateway control* that manages access to the I/O unit and implements the unit's configuration switching. (3) The IO-Guard features a dedicated *interface for uncritical tasks* (i. e., indeterministic cores) that intercepts transaction requests, checks their (estimated) transaction length against the next critical transaction window, and, in case of a conflict, defers the allocation.

At configuration time, task parameters (i. e., offset, period, WCET) are set in the IO-Guard's schedule table. Reconfiguration at runtime is possible, for example, to implement

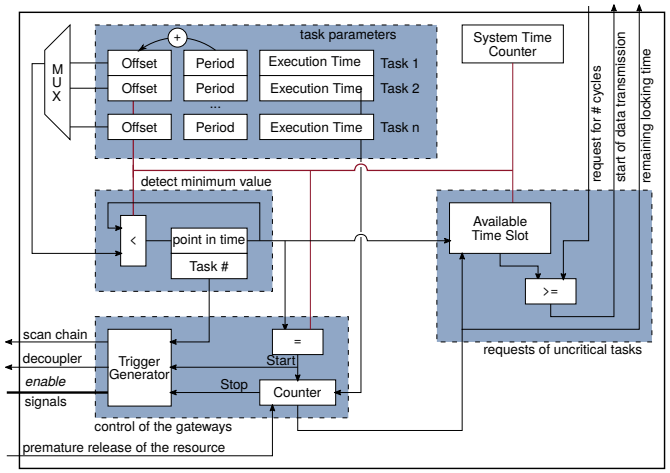


Figure 5: *IO-Guard*'s application-specific hardware structure

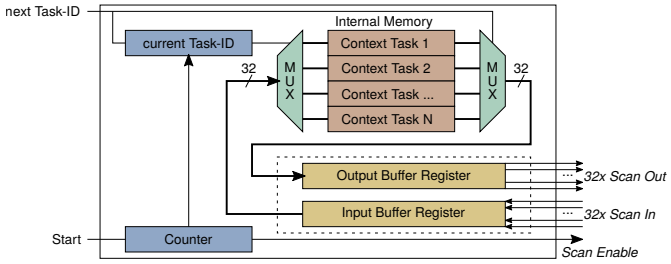


Figure 6: The Scan Chain Controller Hardware Structure

different criticality levels or operating modes. Note that our approach factors in all overheads (e. g., context switching); thus, execution times subsume the total (max.) transaction time, providing a safe upper bound.

a) Critical Operations: Our design results in an entirely predictable I/O access for critical tasks: DEUs and IO-Guards share a common time base (i. e., system timer). The IO-Guard continuously updates all offset registers to indicate the tasks' next I/O cycles. A minimum value search identifies the next transaction windows by a (deterministic) comparison of all offset registers. When reaching a transaction window (i. e., start of the transaction minus all overheads), the control logic issues a context switch (i. e., saves current context and restores the correct configuration) and connects the respective DEU, thereby granting the critical task exclusive access to the resource. The *stop signal* is triggered if the WCET was exceeded (i. e., a fault occurred) or the transaction underran its window (i. e., better than worst-case).

b) Uncritical Operations: To recall, I/O transactions are generally not preemptable; accordingly, interference by uncritical tasks must be avoided. Our application-aware synthesis of the IO-Guards and the predictability of the critical tasks' transaction windows enable an anticipatory reservation of the I/O unit. Therefore, uncritical requests are checked by their (estimated) transaction time: if there is a large enough free slot, the uncritical transaction is granted. Here, we exploit that the transaction length is already part of the usual I/O interface in many cases (e. g., buffer length) and therefore readily available; we discuss our approach's transparency aspects in Section IV-D. Contrarily, when too large, the request is rejected by pretending a busy unit, which is signaled by the appropriate interrupt. Note that the available time can also be queried to optimize the overall I/O utilization (e. g., to adjust the message length or prioritize shorter transactions).

If the uncritical transaction exceeds the allotted time frame, the IO-Guard aborts the transaction. This is signaled by the ordinary error flags and interrupts of the respective I/O unit.

C. Context Saving and Preemptability

While I/O transactions are indivisible, preempting the configuration is an inherent aspect of $LOW_{I/O}$ and key to application-transparent and anticipatory I/O reservations. Note that configuration and (start of) transaction do not represent a contiguous critical section execution-wise. Therefore, interference due to disrupted configurations is hard to bound. Furthermore, a configuration may involve indeterministic polling of status bits (e. g., bus rest), which is plagued by large WCET bounds. Consequently, we provide hardware mechanisms to save and restore a unit's complete hardware state.

Therefore, we infer the relevant state by analyzing the unit's hardware model and synthesizing so-called scan chains to save the context. Scan chains are commonly used as a design for test (DFT) strategy to check manufactured ICs. They can be generated automatically in many EDA tools: all conventional flip-flops are replaced by flip-flops with scan chain ports, so each flip-flop has an additional *scan_enable*, *scan_input* and *scan_output* port. In the second step, the added inputs and outputs are wired as a chain. In our application-specific design, we statically determine the number of replaced flip-flops, and thus the length of the chain, which is an essential parameter for the hardware generation and the resulting timing model (e. g., cycle-accurate triggering of control signals). In $LOW_{I/O}$, we opt for a parallel approach with 32 chains to minimize latencies. Thus, switching the periphery controller's configuration in our running example is reduced from 299 to 10 clock cycles. Further improvements with higher parallelization are possible at increasing costs.

Figure 6 details the scan chain controller: Upon access, the IO-Guard sends the respective task ID and a trigger signal to start the context switch. Consequently, the configuration is shifted word by word from and to the I/O unit with the input data words stored in the internal memory. Finally, the loaded task ID is stored in the *Current Task ID* register. The internal memory layout is application-specific and contains one configuration for each critical task. The uncritical tasks share one common configuration, as their multiplexing is already in the OS's responsibility.

As DEUs and COTS cores run in parallel, context switches must be synchronized between the two domains. Consequently, the IO-Guard closes the gateways for uncritical operations and decouples all ports using the *Decoupler* (cf. Figure 4). The latter keeps all input and output signals to a constant level,

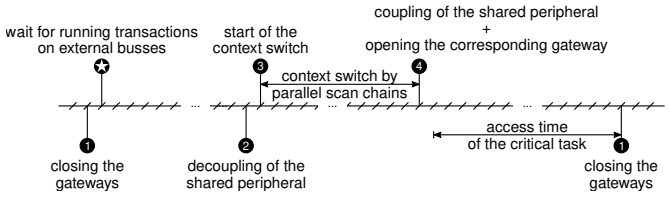


Figure 7: Chronological progression of a context switch. Each tick represents one clock cycle.

so that no data transfers or interrupts are triggered while the configuration is switched.

a) Critical Configurations: The configuration of critical transactions is stored in the IO-Guard as part of the system initialization or reconfiguration. Accordingly, no further action is required during normal operation.

b) Uncritical Configurations: The COTS cores are connected by default, leaving register accesses of uncritical tasks to a shared periphery unaltered. Whenever a critical transaction is active, the COTS gateway is closed. Subsequently, all register accesses get trapped by the gateway to avoid inconsistencies or bus errors. A specific trap is therefore issued, which must be handled by the operating system. In particular, the instruction addressing the peripheral register must be re-executed when the unit becomes available again.

c) Timing Aspects and Latency Hiding: Providing deterministic base mechanisms is a vital aspect of our approach. Figure 7 details our timing model by the example of a context switch: First, all gateways are closed ❶, for which, in the worst-case, the IO-Guard must wait for active AXI transfers for up to 5 clock cycles. At this point, in the worst case, an uncritical task has exceeded its budget and just managed to write one last character into the SPI output buffer. Accordingly, the guard at instant ✪ must first wait for the bus to come to rest. In our running example, this takes up to 1,600 cycles. Before activating the context switch, all output ports must be put into tristate mode ❷ by the decoupler costing one cycle. After that, the context switch is executed ❸. Its duration depends on the number of registers to be saved and the degree of parallelism. In our example, the SPI comprises 299 registers handled by 32 scan chains in 10 cycles. Subsequently, one cycle is necessary to reconnect the outputs and open the gateway for the DEU ❹. From then on, the critical task can access the unit unhindered. In sum, the worst-case overhead is 17+1,600 clock cycles in this example. Note that these numbers can automatically be inferred from static analysis.

To eliminate the enormous impact of bus latencies from the critical tasks' timing model, we hide these costs at the uncritical side. The issue is in the uncontrolled overrun of the assumed budget by uncritical tasks, which we prevent by initiating the context switch for a critical transaction just this latency earlier. Still, the uncritical task can fully utilize its budget.

D. Application Transparency and Implementation Aspects

For the uncritical tasks and their I/O transactions, $LOW_{I/O}$ is easy to integrate into existing settings with only two additional events to be handled: (1) An uncritical transaction is aborted by the guard due to exceeding its assumed budget. A bus error generated by the IO-Guard indicates this event, which we consider to be handled by the driver anyway. Ultimately, this corresponds to a communication error. However, attention should be paid to the application logic, which should be designed with retransmission in mind.

(2) Preemption of an active uncritical configuration with the respective transaction not yet started. This occasion is indicated by a special memory trap, which must be handled by a generic interrupt handler: The OS must suspend the current task (i. e., driver) and reactivate it upon guard release. Note that the instruction pointer must be decremented such that the last register access is executed again. For processors with an out-of-order execution pipeline, the sequence of executed instructions must be observed and restored. For the ARM Cortex-A9 processor in Figure 4, a *Data Synchronization Barrier* can be used. Overall, we consider these extensions easy to implement.

Out of this paper's scope is further optimization of the average utilization and response times for uncritical I/O transactions. This includes, in particular, the previously described facilities for querying the available free transaction slots as well as the signaling of free resources by the IO-Guard. These require extensions to the I/O subsystem of the operating system to prioritize appropriate transaction lengths, for example.

E. Instruction-level Timing Model

The primary outcome from $LOW_{I/O}$ is an automatically generated instruction-level timing model. Due to the hardware-generation approach, this model is tight; that is, it does not involve unnecessary pessimism, which is likely when documentation on the bus controller and/or bus accesses is limited. This model then serves as input to an existing WCET analysis tool, such as AbsInt's aiT [40] or Platin [41] from the T-CREST project [42]. Using the correct-by-construction model, the $LOW_{I/O}$ approach re/enables tight WCET/WCRT bounds for critical tasks by standard schedulability analyses.

V. Evaluation

Reducing the request to acquisition latency to zero for critical tasks comes at the expense of increased hardware resources and enlarged latencies for uncritical tasks. This chapter evaluates our approach focusing on hardware overhead, functionality, and effect on uncritical tasks.

A. Utilization of FPGA Logic

The $LOW_{I/O}$ approach requires a hardware extension of existing peripheral components. This Section analyzes an implementation of the example illustrated in Figure 4 on a *Xilinx Zynq 7020* SoC device to give an overview of required FPGA resources for these hardware extensions. The Zynq 7020 is a lower-end device with 53200 available lookup tables (LUTs)

and 106000 registers. Table I lists the total number of applied LUTs and registers.

The relative numbers show that only a small amount of FPGA resources are needed compared to the whole chip size.

Furthermore, the resource usage of some parts (see Figure 4) depends on the number and tasks' parameters that require access to the shared peripheral (colored brown). Others depend on the parameters of the shared peripheral (i. e., number of pins/registers, colored green). The number of critical tasks requiring access to the shared peripheral defines the number of gateways and consequently their resources. Moreover, the peripheral access parameters of each critical task have to be stored within the IO-Guard (see Figure 5). Therefore, the IO-Guard's register utilization depends on the number of critical tasks with access to this peripheral. In our example (two DEUs, see Figure 4) implemented on the Zynq 7020, the IO-Guard's application-depended hardware occupies about 1.7% of the available LUTs and about 0.4% of the registers.

The peripheral's extension by the scan chain depends on the peripheral's number of registers that will be replaced by special scan-chain registers with additional control and data ports. This correlation also applies to the scan-chain controller, which holds the contexts of the different tasks. Furthermore, the resource usage of the decoupler is determined by the number of input and output pins of the peripheral, which is negligible in comparison to the other components. While scan-chain registers need to be emulated on FPGAs at the cost of additional LUTs, these registers can be implemented efficiently in ASIC designs.

In total, our example requires 906 additional LUTs (1.7% of the total available LUTs), primarily used for control logic and the IO-Guard's context memory.

B. Delay on Uncritical Tasks

$LOW_{I/O}$ proactively reserves an I/O unit for critical tasks to eliminate latencies when accessing the communication hardware. Uncritical tasks might temporarily be denied access to the I/O unit in case their transmission cannot be guaranteed to finish in time to achieve the reservation of the I/O unit. While this approach cuts latencies on critical tasks, uncritical tasks suffer additional delays. In this Section, we exemplarily evaluate these additional delays by comparing the time between request and acquisition of the SPI bus for both a system without I/O Guard, as well as the corresponding system with I/O Guard.

Table I: Required hardware resources of the shared SPI example on a *Xilinx Zynq 7020* device.

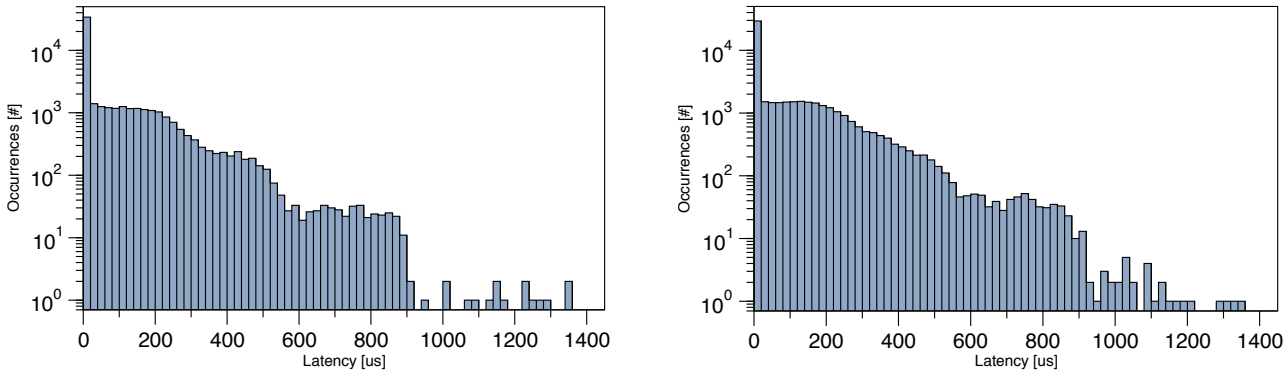
	LUTs	**Registers**
IOGuard	284 (0.5%)	391 (0.4%)
Gateways	28 (0.1%)	8 (0.0%)
Peripheral Extension by Scan Chain	257 (0.5%)	0 (0.0%)
Context Memory in Scan Chain Controller	331 (0.6%)	57 (0.1%)
Decoupler	6 (0.0%)	0 (0.0%)
Total	906 (1.7%)	456 (0.4%)

Due to the high number of evaluation runs required, this evaluation is based on a simulation of our FPGA prototype system made up of two A9 cores executing the uncritical tasks and three DEUs. The simulated system uses, just like the FPGA hardware platform, DEUs running at a frequency of $100\,\mathrm{MHz}$ and an SPI bus with a data transmission rate of $2\,\mathrm{Mbit/s}$. To verify the correctness of the simulator and the significance of the simulation results, we compared the simulation results of a schedule against the values obtained from the measurement of the same schedule on our prototype platform. Figure 8 illustrates the latencies of the uncritical tasks between request and acquisition of the communication bus as histogram plots. The simulated and measured results show a similar distribution. The maximum latencies differ by about 7%, confirming the validity of our simulation results.

The systems under evaluation are generated randomly with varying SPI utilizations of both DEUs and A9 cores. For every pair of DEU/A9-utilization values not exceeding 100% utilization, we randomly generate and simulate ten systems whose average delay (in DEU cycles) between request and acquisition of the SPI bus is eventually plotted in Figure 9. For this evaluation, we assume that the critical tasks exhibit a communication pattern that resembles the behavior of typical, deeply-embedded hard real-time systems: Communication requests occur at high frequency but have a short duration, which is common for interacting with, for instance, sensors. This communication pattern also implies that the DEUs' overall SPI utilization is minor; therefore, we only simulate configurations with the DEUs using up to 30% of the available communication time. Each hyperperiod of the generated systems lasts $1\,000\,000\,000\,\mathrm{DEU\,cycles}$ (equaling $10\,\mathrm{s}$ with our DEUs running at $100\,\mathrm{MHz}$), and every simulation comprises ten hyperperiods (i. e., $100\,\mathrm{s}$ for every simulation).

Figures 9a and 9b show the values for systems without I/O Guard: Without I/O Guard, we assume that the critical tasks' requests are configured to have a higher priority than requests issued by the uncritical tasks running on the A9 cores. The latencies on such critical tasks primarily depend on the A9 communication utilization, as the critical tasks are planned such that they do not overlap with other critical tasks and, by that, are primarily delayed by ongoing transmission by uncritical tasks. For uncritical tasks' delays (see Figure 9b) essentially increase with increasing utilization of the communication bus.

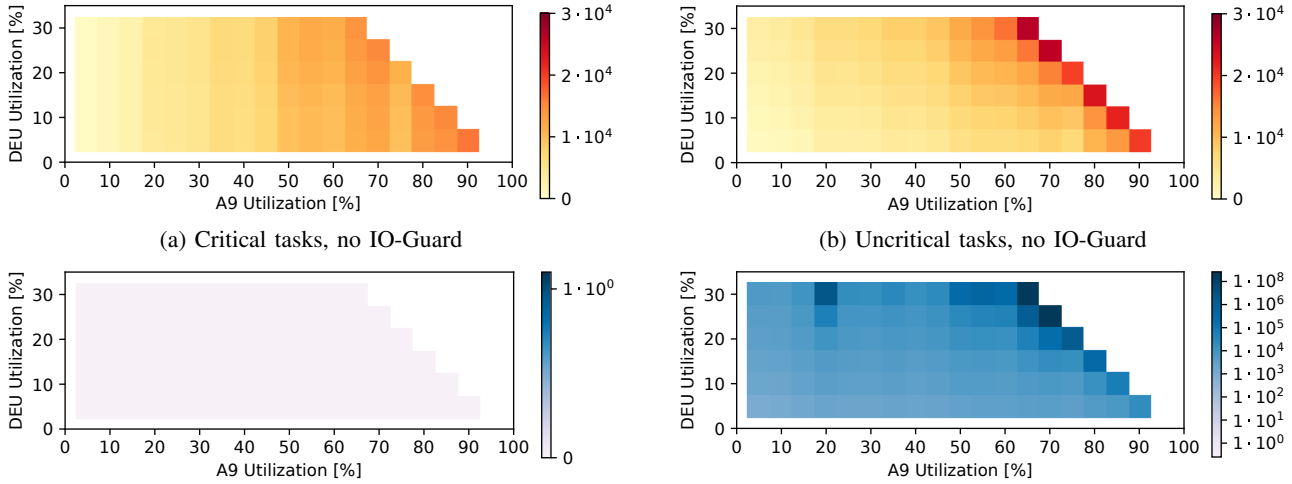
When enabling the I/O Guard, the critical tasks' communication requests are executed immediately with guaranteed latency of zero, see Figure 9c. This strong guarantee of determinism, however, is at the expense of the communication requests issued by the uncritical tasks, as illustrated in Figure 9d: One source of additional delays on the uncritical tasks is the additional overhead of context switching induced by $LOW_{I/O}$ (see Section IV-C). The other, more prominent source of delays is the reservation of the communication bus: Increasing the DEU communication utilization and, by that, increasing the time the A9 cores are prohibited access to the communication bus drastically increases the delay on



(a) Measured latencies of uncritical tasks

(b) Simulated latencies of uncritical tasks

Figure 8: To evaluate the accuracy of the simulated results, this figure shows a comparison of measured and simulated latencies of uncritical tasks. In both cases, an identical schedule was used on a hardware system with IO-Guard.



(a) Critical tasks, no IO-Guard

(b) Uncritical tasks, no IO-Guard

(c) Critical tasks, with IO-Guard, zero latency

(d) Uncritical tasks, with IO-Guard

Figure 9: Average latency (in DEU cycles) between request and acquisition of the SPI bus (i. e., bus-access latency)

A9 communication requests when the DEUs reach 30 % of the available communication time. These increasing delays are caused by the rising number of non-preemptable periods with reserved communication bus, which subdivide the time the communication bus remains idle in slots of fixed sizes. Communication requests exceeding most of these slots suffer potentially long delays, especially when the overall utilization is uncommonly close to 100 %.

From this simulation, we conclude that $LOW_{I/O}$ is able to guarantee zero-latency communication requests for critical tasks running on the highly predictable DEUs while retaining increased but practically feasible delays on the communication requests issued by uncritical tasks. $LOW_{I/O}$ requires additional hardware units to be synthesized, whose implementation only occupies a tiny fraction of the resources available on even lower-end FPGA systems, making $LOW_{I/O}$ suitable for practical use.

VI. Conclusion

While much effort has been made in the area of memory-centric or packet-switched I/O for real-time systems with mixed criticalities, a significant aspect has been left unaddressed: low-level and transaction-based communication interfaces, such as SPI or I^{2}C. In this paper, we presented $LOW_{I/O}$, an approach that exploits target-specific knowledge of communicating tasks on SoC-based hardware platforms. This knowledge is used for tailoring an SoC-based hardware platform that guarantees the timely completion of highly critical tasks and serves tasks of minor criticality with the best effort. Our evaluation results validate the interference-free communication with an acceptable hardware effort.

Acknowledgment

This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 146371743 – TRR 89 “Invasive Computing” and grants no. SCHR 603/15-2 “LARN”, SCHR 603/10-2 “COKE”, and SCHR 603/9-2 “AORTA”.

REFERENCES

[1] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Proceedings of the 19th Real-Time and Embedded Technology and Applications Symposium (RTAS '13)*, 2013, pp. 55–64.

[2] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *Proceedings of the 19th Real-Time and Embedded Technology and Applications Symposium (RTAS '14)*, 2014, pp. 155–166.

[3] A. Burns and R. Davis, "Mixed criticality systems - a review," *Department of Computer Science, University of York, 12th Edition*, pp. 1–81, March 2019.

[4] Z. Jiang, N. Audsley, P. Dong, N. Guan, X. Dai, and L. Wei, "MCS-IOV: Real-time I/O virtualization for mixed-criticality systems," in *Proceedings of the Real-Time Systems Symposium (RTSS '19)*, 2019, pp. 326–338.

[5] E. Missimer, K. Missimer, and R. West, "Mixed-criticality scheduling with I/O," in *Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS '16)*, 2016, pp. 120–130.

[6] G. Carvajal and S. Fischmeister, "An open platform for mixed-criticality real-time ethernet," in *Processings of the Design, Automation & Test in Europe Conference & Exhibition (DATE '13)*, 2013, pp. 153–156.

[7] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "PRET DRAM controller: Bank privatization for predictability and temporal isolation," in *Proceedings of the 7th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '11)*, 2011, pp. 99–108.

[8] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: A predictable SDRAM memory controller," in *Proceedings of the 3rd International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '07)*, 2007, pp. 251–256.

[9] S. Goossens, J. Kuijsten, B. Akesson, and K. Goossens, "A reconfigurable real-time SDRAM controller for mixed time-criticality systems," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '13)*, 2013, pp. 1–10.

[10] J.-M. Irazabal and S. Blozis, *AN10216-01 I2C MANUAL*, 2003.

[11] M. Heene, S. Hill, and J. Jelemensky, "Queued serial peripheral interface for use in a data processing system," 1989, patent.

[12] Electronic Industries Association, Engineering Department, "Interface between data terminal equipment and data communication equipment employing serial binary data interchange," 1969.

[13] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Proceedings of the 28th Intl. Real-Time Systems Symposium (RTSS '07)*, 2007, pp. 239–243.

[14] Xilinx, Inc., *Zynq-7000 SoC Technical Reference Manual*, 2018.

[15] S. Vaas, P. Ulbrich, M. Reichenbach, and D. Fey, "Application-specific tailoring of multi-core SoCs for real-time systems with diverse predictability demands," *Journal of Signal Processing Systems*, Jul. 2018.

[16] ——, "The best of both: High-performance and deterministic real-time executive by application-specific multi-core SoCs," in *Proceedings of the 11th Conference on Design and Architectures for Signal and Image Processing (DASIP '17)*, 2017.

[17] R. Pellizzoni and M. Caccamo, "Adaptive allocation of software and hardware real-time tasks for FPGA-based embedded systems," in *Proceedings of the 12th Real-Time and Embedded Technology and Applications Symposium (RTAS '06)*, 2006, pp. 208–220.

[18] R. Pellizzoni and M. Caccamo, "Real-time management of hardware and software tasks for FPGA-based embedded systems," *IEEE Transactions on Computers*, vol. 56, no. 12, pp. 1666–1680, 2007.

[19] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha, "Coscheduling of CPU and I/O transactions in COTS-based embedded systems," in *Proceedings of the 29th Real-Time Systems Symposium (RTSS '08)*, 2008, pp. 221–231.

[20] R. Pellizzoni and M. Caccamo, "Impact of peripheral-processor interference on WCET analysis of real-time embedded systems," *IEEE Transactions on Computers*, vol. 59, no. 3, pp. 400–415, 2010.

[21] M. Paolieri, E. Quinones, F. J. Cazorla, and M. Valero, "An analyzable memory controller for hard real-time CMPs," *IEEE Embedded Systems Letters*, vol. 1, no. 4, pp. 86–90, 2009.

[22] M. Hassan, H. Patel, and R. Pellizzoni, "PMC: A requirement-aware dram controller for multicore mixed criticality systems," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 4, 2017.

[23] B. Cilku, A. Crespo, P. Puschner, J. Coronel, and S. Peiro, "A TDMA-based arbitration scheme for mixed-criticality multicore platforms," in *Proceedings of the International Conference on Event-based Control, Communication, and Signal Processing (EBCCSP '15)*, 2015, pp. 1–6.

[24] SAE International, "Time-triggered ethernet AS6802," 2016.

[25] L. Ecco, S. Tobuschat, S. Saidi, and R. Ernst, "A mixed critical memory controller using bank privatization and fixed priority scheduling," in *Proceedings of the 20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '14)*, 2014, pp. 1–10.

[26] Z. Owda and R. Obermaisser, "Mixed-criticality transactional memory controller for embedded systems," in *Proceedings of the 14th International Conference on Industrial Informatics (INDIN '16)*, 2016, pp. 104–110.

[27] H. Kim, D. Broman, E. A. Lee, M. Zimmer, A. Shrivastava, and J. Oh, "A predictable and command-level priority-based dram controller for mixed-criticality systems," in *Proceedings of the 21st Real-Time and Embedded Technology and Applications Symposium (RTAS '15)*, 2015, pp. 317–326.

[28] A. Oliveira, J. Martins, J. Cabral, A. Tavares, and S. Pinto, "Tz- virtio: Enabling standardized inter-partition communication in a trustzone-assisted hypervisor," in *Proceedings of the 27th International Symposium on Industrial Electronics (ISIE '18)*, 2018, pp. 708–713.

[29] Infineon Technologies AG, "AURIX 32-bit microcontrollers for automotive and industrial applications," Tech. Rep., 2020.

[30] N. Asmussen, M. Roitzsch, and H. Härtig, "M^3x: Autonomous accelerators via context-enabled fast-path communication," in *Proceedings of the Annual Technical Conference (ATC '19)*, 2019, pp. 617–632.

[31] C. Dietrich and D. Lohmann, "OSEK-V: application-specific rtos instantiation in hardware," in *Proceedings of the 18th Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '17)*, 2017, pp. 111–120.

[32] OSEK/VDX Group, "Operating system specification 2.2.3," Feb. 2005.

[33] Open Bionics Labs. [Online]. Available: https://openbionicslabs.com/

[34] V. Pandiyan, W. Caesarendra, T. Tjahjowidodo, and H. H. Tan, "In-process tool condition monitoring in compliant abrasive belt grinding process using support vector machine and genetic algorithm," *Journal of manufacturing processes*, vol. 31, pp. 199–213, 2018.

[35] F. Scheler and W. Schröder-Preikschat, "The RTSC: Leveraging the migration from event-triggered to time-triggered systems," in *Proceedings of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC '10)*, 2010, pp. 34–41.

[36] F. Scheler and W. Schröder-Preikschat, "The real-time systems compiler: Migrating event-triggered systems to time-triggered systems," *Software: Practice and Experience*, vol. 41, no. 12, pp. 1491–1515, 2011.

[37] F. Franzmann, T. Klaus, P. Ulbrich, P. Deinhardt, B. Steffes, F. Scheler, and W. Schröder-Preikschat, "From intent to effect: Tool-based generation of time-triggered real-time systems on multi-core processors," in *Proceedings of the 19th International Symposium on Real-Time Distributed Computing (ISORC '16)*, 2016, pp. 134–141.

[38] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO '04)*, 2004, pp. 75–86.

[39] S. Vaas, M. Reichenbach, U. Margull, and D. Fey, "The R2-D2 toolchain — automated porting of safety-critical applications to FPGAs," pp. 1–7, 2016.

[40] AbsInt. aiT WCET analyzers. https://www.absint.com/ait/.

[41] P. Puschner, D. Prokesch, B. Huber, J. Knoop, S. Hepp, and G. Gebhard, "The T-CREST approach of compiler and WCET-analysis integration," in *Proceedings of the 9th Workshop on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS '13)*, 2013, pp. 33–40.

[42] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.