PSIC: Priority-Strict Multi-Core IRQ Processing

Malte Bargholz
Leibniz Universität Hannover
maltebargholz@gmail.com

Christian Dietrich
Technische Universität Hamburg
christian.dietrich@tuhh.de

Daniel Lohmann
Leibniz Universität Hannover
lohmann@sra.uni-hannover.de

***Abstract*—While processing external events, in the form of *interrupt requests (IRQs)*, is a key concern of digital control systems, processing these events can be of different importance for a system's functionality. Therefore, it is necessary, especially for real-time systems, to ensure that the handling of low-priority IRQs does not interfere with high-priority *interrupt-service routines (ISRs)* to prevent *priority inversions*. While prioritizing ISRs on single-core machines is a long-solved problem, priority-strict IRQ handling in multi-core systems is, as we will show, quite challenging with current interrupt controllers.**

With PSIC, we propose a hardware/software co-design that ensures the priority-strict execution of the top-m ISRs on an m-core machine at minimal interruption-induced overheads. We developed a drop-in replacement for an off-the-shelf interrupt controller that delivers IRQs in strict priority order while achieving low delivering delays at moderate hardware costs. Combined with a minimal IRQ software subsystem, which requires no inter-core synchronization, PSIC guarantees a priority-strict ISR execution on multiple cores.

I. Introduction

In event-driven control systems, external events are signaled to the CPU by means of *interrupt requests (IRQs)*. Upon an IRQ, the processor performs a (software-invisible) call to a subroutine, the *interrupt-service routine (ISR)*, which *interrupts* the current control flow and, on completion, resumes to the interrupted program. On single-core systems, both, IRQ delivery and IRQ processing, are typically *priority strict*: Each IRQ source can be assigned an individual (and commonly unique) priority and triggered IRQs are *delivered* in order of their relative priority. The processor, in turn, accepts the IRQ for *processing* only if its priority is higher than the currently served ISR. Hence, ISR executions can nest according to their priority. The *interrupt controller (IC)*, a (conceptually) dedicated hardware component, implements this IRQ-acceptance protocol by buffering the external IRQs and forwarding the IRQ with the highest priority to the CPU.

Priority strictness is an important property for real-time systems: The IC implements a simple form of fixed-priority preemptive scheduling, which we can employ to offload *rate-monotonic scheduling (RMS)* or *deadline-monotonic scheduling (DMS)* [16] of external (and possibly also internal [11]) events to the hardware. Intuitively, a priority-strict IC for a multi-core system would provide global fixed-priority preemptive scheduling [7]: Given that there are m (identical) CPUs, at any point in time, the ISRs of the top-m triggered IRQs get executed on a CPU. Unfortunately, the existing ICs are not priority-strict in the multi-core case, but they either fall back to global partitioned scheduling (e.g., Infineon Aurix [26]), or interrupt multiple cores and offload the coordination to the software (e.g., ARM [1], RISC-V [27] or MPC5676 [17]), which leads to priority inversions and overheads.

978-1-6654-0627-7/22/$31.00 ©2022 IEEE

A. Contributions

So, instead of retroactively fixing the symptoms of a non-priority-strict IC in software with all associated costs, we propose a combined HW/SW approach for priority-strict IRQ processing. Our key contributions are:

1) Identification of shortcomings in available ICs for prioritized multi-core interrupt processing.
2) Design of the first priority-strict multi-core interrupt controller at moderate hardware costs.
3) A lock-free HW/SW IRQ-processing subsystem with short and bounded violations of priority strictness.

We describe the shortcomings of existing ICs (Sec. II), before we present our own priority-strict multi-core IC (Sec. III). Afterwards, we quantify the induced hardware costs (Sec. IV) and the end-to-end IRQ latency, before we discuss the related work (Sec. V) and our design decisions (Sec. VI).

II. Systematic Violations of Priority Strictness

First of all, we want describe prioritized multi-core IRQ processing formally and highlight the problematic scenarios. By showing the shortcomings of existing ICs, we argue that *only* priority-strict IRQ delivery in combination with adequate software routines can achieve priority-strict ISR processing.

A. System Model

In a system with n IRQ sources and m identical CPUs, each IRQ source $i \in \mathcal{I}$ is configured with an arbitrary, but bounded, priority $p_i > 0$. When an external event at an IRQ source *triggers* an *interrupt request (IRQ)*, it inherits the source's priority. Although the source can buffer further external events, each source has at most one pending IRQ, which we consider *pending* until the IRQ handling completes. The IC *delivers* the IRQ to a CPU, which *interrupts* the control flow, activates the ISR, which then completes *completes* the IRQ.

The set $\mathcal{P} = \{i, j, \ldots\}$ contains the pending and not-yet-completed IRQs, while the CPU configuration $\mathcal{C} = \langle j, i, -, \ldots \rangle$ indicates the currently executing ISR ("–": no ISR execution). Without loss of generality, we assume that ambiguous priorities are resolved systematically and that $p_i = i$. In combination, $(\mathcal{P}, \mathcal{C})$ describe the full IRQ-processing state at one point in time. For example, $(\{15, 10, 3\}, \langle 10, 15 \rangle)$ describes a two-CPU

	ARM GIC	Aurix IR	MPC5676	I/O APIC	PLIC	PSIC
Global IRQ Delivery	(✓)	×	(✓)	✓	(✓)	✓
Lowest-Priority Delivery	(✓)	×	×	(✓)	(✓)	✓
IRQ Migration	×	✓	×	×	✓	✓

TABLE I: Feature Matrix of available interrupt controllers

system with three pending IRQs, where CPU 0 processes ISR 10 and CPU 1 works on ISR 15, while IRQ 3 makes no progress. Furthermore, we define the *CPU priority* as the priority of the currently processed IRQ or zero if no ISR is running.

We say that the system (currently) processes IRQs *priority strict*, iff its CPUs process the top-m elements of $\mathcal{P}$ ($\text{top}_m(\mathcal{P}) \subseteq \mathcal{C}$). In hard real-time systems, violations of priority strictness are only tolerable if they are temporary and bounded in length, such that we can account for them as overheads in the real-time analysis. Therefore, we have to avoid systematic violations of priority strictness (e.g., priority inversion) and should aim for minimal system-software overheads to achieve low worst-case bounds. In the following, we want to look at situations where off-the-shelve ICs (see also Tab. I) induce systematic violations and, therefore, force the system software to use mitigation strategies that demote the systematic violations to temporary ones.

B. Global IRQ Delivery

For priority strictness, IRQs cannot be bound to specific CPUs but they must be scheduled globally to avoid situations where an CPU would be available, but the IC configurations prohibits the delivery of an IRQ to this CPU. For example, if all IRQs in a two-CPU system are bound to the CPU 0, the system is not priority strict if more than one IRQ is pending: $(\{1, 3, 5\}, \langle 5, – \rangle)$.

Looking at commercially-available multi-core ICs, many already have shortcomings in this basal property: On the *Interrupt Router* (IR) of the Infineon Aurix [26], each IRQ source is connected to exactly one CPU. While ARM's *General Interrupt Controller* (GIC) [1] and the *platform-level interrupt controller (PLIC)* of RISC-V [27] deliver an IRQ globally, they might interrupt multiple CPUs, which then execute the same ISR. All ISR execution then try to claim the IRQ, only one being successfull, which introduces an unavoidable interference to the system. The *Interrupt Controller* (INTC) of MPC5676 [17] always delivers IRQs to both CPUs and leaves synchronization to software. Only the APIC system of IA-32/AMD64 machines with its I/O APIC [13] is able to deliver an IRQ prioritized to a single CPU.

C. Lowest-Priority Interruption

Furthermore, IRQs have to be delivered to the CPU with the lowest *sufficient* priority: To prevent priority inversions from low-priority IRQs, we raise the CPU priority during the ISR execution. Thereby, for newly ariving IRQs multiple CPUs can have a sufficiently low priority for acceptance. However, the IC systematically violates priority strictness if it delivers the IRQ to a CPU that has not the lowest of the sufficient priorities. For example, in state $(\{3, 4\}, \langle 4, 3 \rangle)$, both CPUs have a sufficiently low priority to accept and IRQ with $p_i = 5$. However, if we deliver the IRQ to CPU 0, the system transitions into the violating state $(\{5, 4, 3\}, \langle 5, 3 \rangle)$.

From the ICs that support global IRQ delivery, GIC and PLIC only guarantee that the IRQ is sent to one (or many) of the CPUs with sufficiently low priority, which results in the described systematic violation. If the INTC delivers globally, it delivers the IRQ to both CPUs. Only the APIC system supports a delivery mode (`001`) that targets the CPU with the lowest priority. However, lowest-priority IRQ delivery and CPU-priority check are decoupled: Once delivered to the lowest-priority CPU, ISR execution only starts when the CPU priority becomes sufficiently low to accept the IRQ. Thereby, a delivered IRQ is stuck at the CPU, although another CPU could drop its priority even further becoming an even better target for the IRQ.

D. IRQ Migration and ISR Preemption

Lastly, we argue that priority strictness directly requires IRQ/ISR migration between CPUs. Clearly, for priority strictness, high-priority IRQs must interrupt already executing low-priority ISRs. However, when another CPU has a sufficiently low priority, the interrupted ISR must migrate to that CPU, even if the interrupting high-priority ISR still executes. Without IRQ migration, priority strictness is systematically violated: An IRQ with $p_i = 5$ in state $(\{3, 4\}, \langle 4, 3 \rangle)$ results in the state $(\{3, 4, 5\}, \langle 4, 5 \rangle)$. On completion of ISR 4, CPU 0 must continue the execution of ISR 3 to remain priority strict. However, without ISR migration, IRQ 3 sticks to CPU 1, which results in the violating state $((\{3, 5\}, \langle –, 5 \rangle)$.

Therefore, priority strictness requires us to migrate IRQs *and* ISR between CPUs. On the software side, we can migrate the ISR's CPU state (i.e., registers) by a context switch, but the IC also must provide migration of all IRQ-related state to another CPU interface, if necessary.

On GIC systems, ISRs cannot migrate as the interrupted CPU must also signal the IRQ completion, which results in an atomically executed priority drop to the interrupted IRQ. On MCP5676 and APIC systems, the CPU-local interfaces keep track of interrupted ISRs but provide no software access to the stored state. On the Aurix IR, nested IRQs and IRQ migration are possible since all of the IRQ-related state is accessible from software. And on a PLIC system, ISR migration is unproblematic since every CPU can complete every IRQ and the IC does not keep track which CPU executes an ISR.

E. Problem Statement

As we see, the commonly available multi-core ICs induce different systematic violations of priority strictness. For real-time systems, we have to use mitigation techniques to demote these systematic violations to temporary ones. Examples for such techniques are ISR threads [14] or the retro-active redistribution of IRQs via inter-processor interrupts. However, this

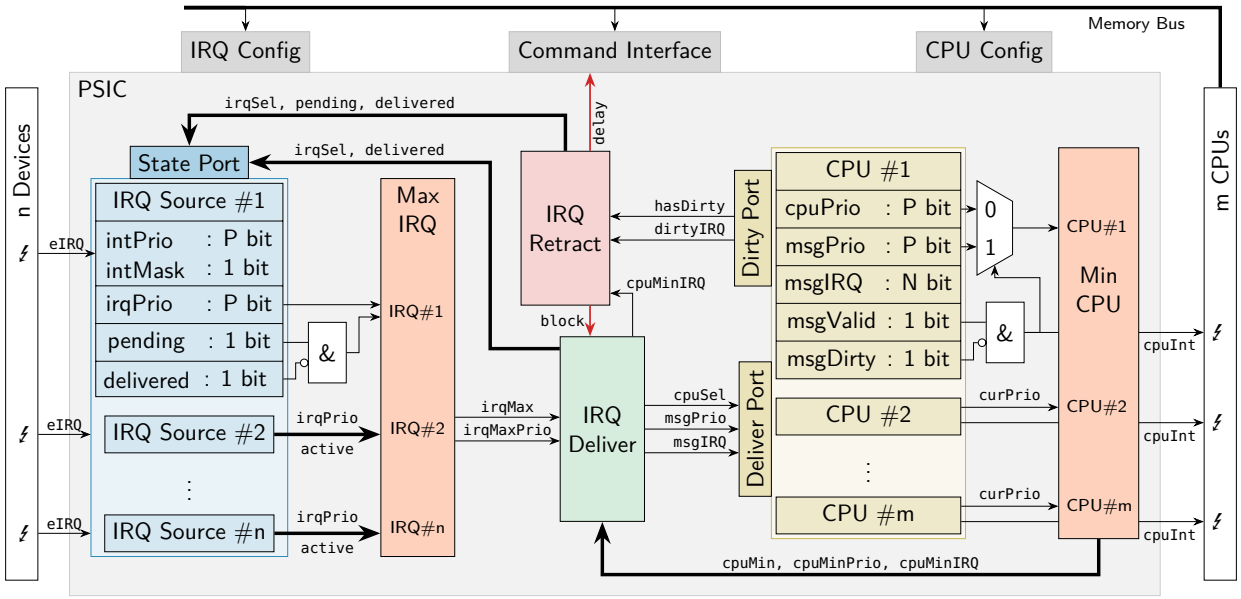


Fig. 1: The PSIC Interrupt Controller

induces overheads into the hot path of IRQ processing, which directly prolongs the worst-case path. Even worse, software-managed ISR distribution requires a consistent view on the global ISR-execution state, which features a synchronization challenge that can become a scalability burden if solved in software alone [15, 6].

III. Priority-Strict Multi-Core IRQ Processing

As ISR migration is mandatory, PSIC is a co-design of an multi-core IC that delivers IRQ globally to the lowest-priority CPU and a minimal software layer that handles, without using global synchronization (i.e., spinlocks), ISR preemption and IRQ migration. Thereby, we deliver IRQs priority strict and execute ISRs with minimal priority-strictness violations.

A. The PSIC Interrupt Controller

In the following, we will describe the design of the PSIC IC (see Fig. 1), highlight situations where race conditions put priority strictness at risk, and show our solutions to these pitfalls. Thereby, we argue that PSIC always delivers the top-m IRQs (out of n IRQs) to m CPUs, even in the presence of concurrent subsystem reconfigurations.

1) The PSIC Interfaces: The controller exposes interfaces to three different sides: (1) n edge-triggered IRQ sources (`eIRQ`), (2) m IRQ output wires (`cpuInt`) that interrupt the CPUs, and (3) The memory-mapped configuration and command interface (MMIO). While IRQ in- and CPU outputs are asynchronous logic signals, the memory bus delivers read- and write requests sequentially and PSIC is able to delay their completion.

At the MMIO interface (see Tab. II), the user can configure and control the operation of PSIC. We designed the interface to be similar to the RISC-V PLIC specification [27] in order to ease porting of existing software stacks. At a CPU interface, the user can read/write the CPU priority atomically without

MMIO Register	Access	Description
cpu.cpuPrio	r/w	Get/set CPU priority (immediately)
cpu.trigger	w	Trigger an IRQ
cpu.claim	r	Claim a delivered IRQ
cpu.redeliver	w	Request the redelivery of an IRQ
cpu.complete	w	Complete an IRQ
irq.intPrio	r/w	Get/set priority of future IRQs
irq.intMask	r/w	En/Disable an IRQ source

TABLE II: PSIC Configuration and Commands

synchronization and priority changes have an immediate impact such that a high-priority ISRs is never interrupted by a low-priority IRQ. Furthermore, we can `trigger` new IRQs and `claim`, `redeliver`, or `complete` already delivered ones. For each IRQ source, we can configure the priority of future IRQs or temporarily disable (mask) the source.

Like the PLIC, the PSIC chip, after being configured by the system software, routes pending IRQs to CPU interfaces, which interrupt the CPU via the `cpuInt` wire. The CPU starts the ISR, which claims the IRQ by reading the IRQ id from the command register. After completion, the ISR signals completion by writing the IRQ id to the PSIC command register. We also introduced a `trigger` command, which allows the software to release IRQs for a given IRQ source, and the `redeliver` command, which requests the redistribution of an already delivered IRQ to another CPU.

2) PSIC Operation: In its operation, PSIC covers the *global-* and *lowest-priority delivery* requirement discussed in Sec. II. Technically, we fulfill these requirements by guaranteeing that: (1) The minimal number of low-priority CPUs are interrupted if new high-priority IRQs arrive. (2) After an interruption, `claim` always yields one of the top-m pending IRQs. (3) CPU-priority changes have an immediate impact and inhibit the delivery of priority-inverted IRQs.

For each IRQ source, we store its priority (`intPrio`) and a mask bit (`intMask`) to temporarily disable the source (see Fig. 1). Furthermore, we buffer a single pending IRQ per source and its `irqPrio`, which it inherits from the source at release time. With two bits, we track the IRQ-delivery state: `pending` marks a filled IRQ buffer, while `delivered` indicates that the IRQ was already delivered to a CPU interface. Throughout PSIC, the IRQ is identified by its IRQ-source id and its priority at release time.

In PSIC, the number of sources (n) and the width of the priority field (P) are independently configurable. By default, we allocate twice as many priority levels than IRQ sources ($P = \lfloor \log_2 n \rfloor + 2$), since priority-space scarcity makes it hard to uniquely map interrupts and software-only tasks into a unified priority space [9].

At the CPU interface, we have a P-bit–wide `cpuPrio` register, which is the *single source of truth* for a CPU's configured priority. Furthermore, each CPU interface has a *message box* that has room for one delivered IRQ (`msgIRQ`, `msgPrio`) and the `msgValid` bit marks as filled, while the `msgDirty` bit indicates that the IRQ must be redelivered.

The central part of PSIC's logic is the delivery of IRQs from the sources to the message boxes, from where a CPU can claim it. For this, the `Max-IRQ` component, which is a ($\lfloor log_2 n \rfloor + 1$)-deep and P-bit wide comparator tree, finds the highest-priority IRQ that is pending, not masked, and not already delivered, and forwards it (`irqMax`, `irqMaxPrio`) to the `IRQ-Deliver` component.

From the CPU side, `Min-CPU`, a ($\lfloor log_2 m \rfloor + 1$) deep tree, select the lowest-priority CPU, whereby current CPU priority (`curPrio`) is either configured priority `cpuPrio` or the priority of the message-box IRQ (`msgPrio`). Please note that `msgPrio` is always higher than `cpuPrio` and that the delivery of an IRQ immediately raises the actual CPU priority.

PSIC delivers the top-m IRQs in a multi-cycle *arbitration*: In each cycle, the `IRQ-Deliver` component compares `irqMaxPrio` to `cpuMinPrio` and delivers the highest-priority IRQ to the lowest-priority message box (`cpuMin`). On delivery, (1) we mark the IRQ source via an one-hot access as delivered, (2) we (optionally) *retract* an already delivered low-priority IRQ from the selected message box and immediately clear its delivered flag, and (3) we fill the message box with (`irqMax`, `irqMaxPrio`) and mark it valid and not dirty. As delivery immediately raises the CPU priority, we exclude the IRQ source and the receiving message box in the next arbitration round. Therefore, `cpuMinPrio` and `irqMaxPrio` converge until a steady state is reached and the arbitration ends. While arbitration is ongoing, PSIC delays all commands that modify the pending or delivery state of an IRQ (`claim`, `complete`, ...). If `cpuPrio` remains untouched and no further IRQ is triggered, an arbitration takes at most m cycles.

The four MMIO commands have the following effect: (1) `trigger` releases an IRQ by mimicking an active `eIRQ`. (2) `claim` clears a CPU's message box and *atomically* increase `cpuPrio` to `msgPrio`, whereby the software takes over responsibility for the event from the hardware. Please note that claiming does not affect the `curPrio` or pending/delivered state of the IRQ source. (3) `redeliver` clears the `delivered` bit of an IRQ source, which provokes a redelivery of the IRQ. (4) `complete` clears `pending` and `delivered` at the IRQ source, making room for a new IRQ. Please note, that only `claim` influences the CPU priority, while any CPU can `complete` an IRQ without changing its priority. If priority stacking is desired, it has to be done in software.

B. Synchronization at the HW/SW Boundary

In Sec. II, we argue that priority-strict IRQ processing is a global synchronization challenge. By centralizing prioritized multi-core IRQ delivery in the IC, we already avoid most shared state between CPUs. However, we also must take a look at the hardware/software boundary and prevent race conditions between CPUs and PSIC.

At the MMIO interface, PSIC relies on the memory bus to deliver commands (see Tab. II), which are all performed by a single read or write, sequentially to achieve a total ordering. While command interleaving between CPUs is arbitrary, we force a consistent ordering for each CPU by using a single per-CPU memory address to issue the commands (`trigger`, `claim`, `redeliver`, `complete`), which we bit-pack with their argument into a single 32-bit word. With RISC-V's preserved program order [28], this allows processors to issue their commands in the intended order, even in the presence of instruction reordering and without fence instructions.

The other important aspect are priority changes (`intPrio`, `cpuPrio`, and `claim`) during an ongoing arbitration. Since `intPrio` only changes the priority of future IRQs, it has no influence on the current arbitration round and is unproblematic. `claim`, which only updates the CPU priority from the message-box priority, also has no influence as the update does not influence `curPrio`.

Only for `cpuPrio`, we have to take a deeper look: Conceptually, PSIC sorts CPUs according to their priority and delivers the top-m IRQs in decreasing order. If the CPU priority changes, the CPU can "jump" to any position in this priority-sorted CPU sequence, whereby previous delivery decisions could become invalid. For example, if an IRQ 4 arrives in the CPU state $\langle 3, 100 \rangle$, we deliver it into the message box of CPU 0. If now CPU 1 drops its priority to zero, while the arbitration is still going on or before the CPU claimed the IRQ, we have to reconsider this delivery decision to end up with the state $\langle 3, 4 \rangle$. We achieve this by invalidating all filled message boxes on every `cpuPrio` by setting the `dirty` flag. With the `IRQ-Retract` component, which blocks delivery and commands while being active, we remove the dirty IRQs from the message boxes by marking them as not delivered at their IRQ source. At worst, a `cpuPrio` results in a retraction and re-arbitration of the top-m IRQs ($2 \cdot m$ cycles).

Summarized, PSIC fulfills the requirements for priority-strict IRQ delivery to the CPU interface, even in the presence of concurrently-issued reconfiguration commands.

```
context_t  ctx[MAX_IRQ];
irq_t   active[MAX_CPU];

void ISR_3() {
  enable_int();
  // ... workload ...
  disable_int();

  setup(&ctx[3], &ISR_3);
  cpu.complete(3);
  cpu.setPrio(0);
}
```

```
void PSIC_ISR_entry() {
  id = cpu.getCurrent();
  irq_t prev = active[id];
  save(&ctx[prev]);

  irq_t next = cpu.claim();
  if (prev != 0)
    cpu.redeliver(prev);
  active[id] = next;
  load(&context[next]);
}
```

Listing 1: The PSIC ISR-Trampoline

C. Preemptable Interrupt Service Routines

While the PSIC IC already delivers the top-m IRQs priority strict, we also must support IRQ/ISR migration to avoid systematic priority-strictness violations (see Sec. II). Nevertheless, even with PSIC, we will introduce a small, unavoidable temporary violation if a running ISR is interrupted.

On the IC side, we do not have to migrate state between CPU interfaces as PSIC does not associate IRQ-specific information to the claiming CPU interface. Therefore, every CPU can complete any IRQ, whereby PSIC is indifferent whether ISRs migrate between CPUs before they complete.

For the software side, we make ISRs migrateable by mapping each ISR to a light-weight thread [14, 12] with small a co-routine context (stack, general-purpose registers, instruction- and stack-pointer, processor-status word). These ISR threads are *not* managed by the OS scheduler but are only preempted and activated by an intermediate PSIC-provided ISR trampoline (see Lst. 1). Our usage of ISR thread is inspired by Hofer et.al. [12], who used a similar technique to allow for blocking system calls during ISR execution.

At boot, we allocate a co-routine context for each ISR (`ctx`) and initialize it (not shown) such that the first resume starts the ISR execution (see Lst. 1, `ISR_3()`). During the actual ISR workload, we unblock the interrupts in the processor-status word to allow for further, higher-priority, interruptions. Please note that this has no influence on PSIC. After the ISR finishes, we reinitialize its context, complete the IRQ, and drop the CPU priority to zero.

The actual ISR-entry function (`PSIC_ISR_entry()`) only preempts the currently running control flow and switches to the activated ISR thread. After saving the current CPU context, which might be an ISR or the main program (`prev=0`), we claim the IRQ at the CPU interface and redeliver, if necessary, the interrupted IRQ, before loading the context of the triggered ISR. Also, `save()`/`load()` save and restore also the interruption-blocking state in the processor-status word.

With PSIC, we avoid software-based inter-core synchronization altogether (i.e., spinlocks) and get away with delaying interrupts CPU locally, while we save and restore the CPU contexts, thus provoking at most a temporary priority-strictness violation. This is possible as every unfinished IRQ always has exactly one owner: On release, the IRQ source owns the IRQ, which transfers ownership on delivery to the CPU interface. The trampoline function then takes ownership by claiming the IRQ and its context. On preemption, a CPU temporarily owns two IRQs but immediately hands back the low-priority IRQ to PSIC with `redeliver`. Thereby, no further inter-CPU synchronization is necessary all software-state synchronization is CPU local and results in bounded worst-case delays as our trampoline is not only lock free but also loop free.

IV. Evaluation

We base our prototype on the Rocket chip generator [5], which is an open-source implementation of the RISC-V I64G ISA specification [28] and is able to generate a cycle-accurate RTL simulation as Verilog code for multi-core systems. We built PSIC as a drop-in alternative to the standard PLIC IC. In the following, we will quantify PSIC's hardware costs, the IRQ-delivery latency, and measure the end-to-end overhead of PSIC-assisted, preemptable multi-core interrupt processing.

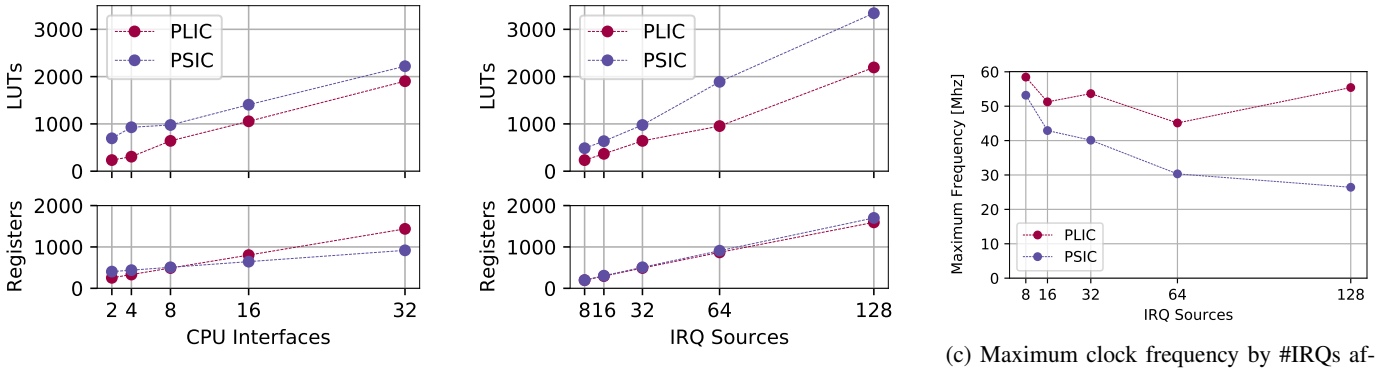
A. Hardware Costs

For the hardware costs, we synthesized the generated Verilog code for a Xilinx XC7000 series FPGA with Vivado 2019.1. Both the original PLIC and our PSIC are synthesized for up to 32 CPU interfaces and up to 128 interrupt-sources. For each instance we recorded the amount of *lookup tables (LUTs)* and registers used and the maximum frequency at which it can be clocked. At this point, due to missing licenses, we did not perform the final routing step. However, for small systems (up to 4 CPUs and 128 IRQ source) the difference before and after routing was negligible (< 10 LUTs and registers). For reference, a single I64G Rocket core requires around 20k LUTs and 9k registers.

For Fig. 2a, we fix the number of IRQ sources (n=32) and compare the hardware costs to the original PLIC. With regards to logic, PSIC starts with a high base overhead of 196 percent for two CPUs, which, however, drops to 17 percent for 32 CPUs. For two CPUs, we require 59 percent more registers, which however decreases with the number of CPUs and even becomes negative for more than eight CPU interfaces. PSIC requires less registers for many CPUs as we only store one `intMask` per IRQ source, while PLIC has one `intMask` per IRQ and per CPU. Both register and LUT overhead scale almost linearly with regard to CPU interfaces and the high base overhead of PSIC becomes negligible for large multi-core systems.

For Fig. 2b, we fixed the number of CPUs to 8 and varied the number of IRQ sources. For eight IRQ sources, PSIC requires 108 percent more LUTs, which decreases to 52 percent for 128 IRQ sources. For registers, PSIC's overhead is negligible and for 128 IRQ sources, we require only 7 percent more registers. This register overhead stems from the extended IRQ-source state (`delivered`, `irqPrio`).

While PSIC's area requirements scale well with the number of CPUs and IRQs, the required comparator trees impact the maximum clock frequency. To quantify this impact, we synthesized and routed systems with 4 CPUs and a varying number of IRQ sources for a XC7X020 FPGA on which the



(a) By #CPU Interfaces (#IRQ=32)

(b) By #IRQ Sources (#CPU=8)

(c) Maximum clock frequency by #IRQs after routing for a XC7X020 FPGA. (#CPU=4)

Fig. 2: Hardware Costs for the Xilinx XC7000 series FPGA. The two driving cost factors are the amount logic (LUTs) and amount of state (Registers). The x-axes are linear with the number of CPU interfaces and IRQ sources.

complete design can be maximally be clocked at 100 Mhz (see Fig. 2c). While the PLIC always stay above 45 MHz, the PSIC suffer from longer propagation paths. For eight IRQ sources the maximum frequency of PSIC is only moderately lower (53.19 MHz, -9%) than the PLIC (58.46 MHz). However, for 128 IRQs, PSIC's maximum clock rate drops to 26.43 MHz while the PLIC still achieves 55.42 MHz. Nevertheless, for the targeted 4 CPU system RISC-V system this is still higher than the 25 MHz that the complete system achieves on the XC7X020 FPGA.

An analysis of the critical path shows that it starts at `intMask`, continues through the `Max-IRQ` and the `IRQ-Deliver` components, and finally passes through the `State Port` to write a `delivered` register. More colloquial, the critical path of PSIC is the setting of the delivered bit for the selected IRQ with the maximum priority. In Sec. VI, we will outline how a possible mitigation for this issue.

B. IRQ Interference

In order to demonstrate the impact of (missing) priority-strict IRQ delivery on a multi-core system, we constructed a benchmark scenario, where multiple CPUs execute, at different (low) priorities, a cyclic test while an external timer periodically (every t_i µs) triggers a sufficiently high-priority IRQ that could be delivered to all CPUs. When a CPU is interrupted, the ISR claims the IRQ, executes 16 NOPs, and completes the IRQ, before resuming to the cyclic test, which continuously measured how long it takes to count up to a constant value ($C = 65535$). For this benchmark, we did *not* use ISR threads (see Sec. III-C) but only compare the impact of priority-strict IRQ delivery. However, t_i and C are chosen such that the ISR completes before the next periodic event.

We execute the test on a 4-core, 25 MHz system with 128 IRQ sources running on a XC7X020 FPGA. The CPUs are prioritized by their id (1-4), while the IRQ has priority 5 and, thus, can interrupt all CPUs. We compare the PLIC, which interrupts all CPUs with a sufficiently low priority, to PSIC, which delivers the IRQ only to the lowest-priority CPU.

		CPU1	CPU2	CPU3	CPU4
$t_i = 200$	PLIC	5489	5489	5487	5484
	PSIC	5475	5243	5243	5243
$t_i = 50$	PLIC	6267	6261	6253	6243
	PSIC	6228	5247	5247	5247

TABLE III: Time taken (µs) for one loop with $C = 65535$

Tab. III shows the average cycle-count duration over 64 runs. For $t_i = 200$µs, every CPU in the PLIC system suffers uniformly from the interruptions (+4.69 %), while PSIC only induces latency on CPU 1 (+4.42 %). With a higher timer frequency ($t_i = 50$µs), the PLIC interference increases to 19.32 percent.

Furthermore, PSIC generally induces a smaller overhead (18.7 % for $t_i = 50$µs) on the interrupted core than the PLIC (19.32 %). In PLIC systems, each interrupted CPU tries to claim the interrupt, which leads to multiple interfering memory-bus requests, of which only a single one will successfully return an IRQ id. With PSIC, only a single CPU is interrupted and issues a `claim` command.

C. End-to-End Latency

We also perform an end-to-end test of our priority-strict IRQ processing in combination with the ISR threads and migration activated. For this, we measured the interrupt latency of a software-generated interrupt: On the lowest-priority CPU, we trigger an IRQ (`trigger`) and measure, with RISC-V's `mcycle` register, the time until the ISR workload starts to execute.

We repeatedly (n=4800) execute the test on a 4-core, 25 MHz system with 128 IRQ sources running on a XC7X020 FPGA and ignore the first five results to only measure with warm CPU caches. On average, the PSIC system has an end-to-end ISR latency of 6.11 ± 0.09 µs. Over all runs, we observed a maximum interrupt latency of 6.68 µs.

To put these numbers in perspective, we also measure the time for a normal co-routine-context switch that saves the current CPU context to memory and loads another one. As shown in Sec. II-D, the priority-strict execution of ISRs in a

multi-core system always requires us to migrate ISRs, which will also entail CPU-state migration. Please note, that also a non-priority strict IRQ subsystem requires a "half" context switch to save the interrupted CPU state.

We measured a regular context switch 4800 times on the 25 MHz RISC-V machine and we, again, ignoring the first five executions. On average, a full context switch takes 3.49 ± 0.04 µs, or about 87 cycles. On top of this, PSIC adds 66 cycles to the latency.

For a complete picture, we also must look at arbitration delays and priority-change–induced re-arbitration (see Sec. III-B) cycles. Since the worst case is hard to trigger, we give the theoretical worst-case IRQ-delivery delay from `eIRQ` until a CPU can claim the IRQ. Without interference form other IRQs or CPUs, PSIC takes 3 cycles to deliver an IRQ to an CPU. In general, the worst-case delay d_{max} (in cycles), where k is the number of CPU-priority changes during the arbitration is:

$$d_{max} = \underbrace{2}_{A} + \underbrace{m}_{B} + \underbrace{(m-1)}_{C} + \underbrace{2 \cdot m \cdot k}_{D}$$

In the worst-case scenario, our IRQ of interest i is triggered together with at least $(m-1)$ IRQs of lower priority. After two cycles (A), arbitration starts and takes one cycle per IRQ (B). Before arbitration finishes and the CPU could claim, $(m-1)$ high-priority IRQs are triggered and PSIC performs another $(m-1)$ deliveries (C). At this point, our IRQ i became the lowest-priority element of $top_m(\mathcal{P})$. Any further IRQ would either evict i from the top-m set, delaying its delivery until the higher-priority ISR have completed, or leave $\mathcal{P}$ unchanged. To further prolong the worst case, the CPUs perform k change priorities, each earning a full retraction and arbitration $(2m)$ (D). In the worst case, when priority changes are exactly $2m$ cycles apart, this leads to a theoretically unbound IRQ delivery delay.

However, in real systems the number of priority changes is bounded. For example, our ISR trampoline (see Sec. III-C) updates the CPU priority exactly once for an IRQ. As there are at most m executing ISR k is at most m and we end up with an upper bound of $2m^2+2m+1$. In the case of four CPUs, the worst case IRQ-delivery latency is 41 cycles. Combined with the *observed* worst-case of the ISR trampoline, the PSIC IRQ processing on a 4-core system has a worst-case IRQ latency of 208 cycles (or 8.32 µs at 25 MHz) when caches are warm (or the trampoline code is placed in scratchpad memory).

V. Related Work

On the hardware side, real-time–targeted execution platforms often focus on the un-interrupted control flow and leave out IRQ delivery: While the T-CREST project [24] designed a time predictable platform that included many worst-case optimized HW components (i.e., processor, caches, network-on-chip, memory controller), they left out the IC as a source of systematic disturbance. The FlexPRET architecture [29] supports the efficient and predictable execution of mixed-criticality systems by mapping OS threads to hardware threads, but has no special support for multi-core systems, or inter-core thread migration; their prototype does not include an IC.

For priority-strict IRQ delivery, only ICs for single-core systems were proposed: Foyo and Mejia-Alvarez [9] first employed a custom IC with enough flexibility to provide a unified priority space for threads and IRQs. Strnadel [25] let the IC monitors the system load and delays IRQ delivery to avoid IRQ overloads. The EventIRQ [18] IC inserts ISRs directly into an OS queue and only interrupts the CPU if the IRQ priority is sufficiently high.

Another related area of specialized coordination hardware are scheduling co-processors [4, 3, 2, 21], which replace a software-implemented OS scheduler. While PSIC can provide a similar service for global fixed-priority scheduling, when threads are mapped onto IRQ sources, as done by SLOTH [12], our approach does not require a restructuring of the whole system but it is only a drop-in replacement for a regular IC.

On the software side, the rate-monotonic priority inversion problem between IRQs and threads is addressed from two directions: avoidance or integration. We *avoid* priority inversions if the OS delays lower-priority IRQs [10, 8] while high-priority threads (or ISRs) execute. For example, Patel et al. [23] propose a timer subsystem that arms *core-local* timers only if the associated ISR has a higher priority than the currently running thread.

With the *integration* of ISRs into the OS scheduling, the impact of rate-monotonic inversions is limited to a short time period. While the OS still services the IRQ immediately, the ISR only starts an ISR thread, which the OS then schedules regularly [14, 22]. This technique is also part of the `PREEMPT_RT` patch set for Linux [19].

A notable combination of avoidance and integration is the SLOTH project [12], which use a unified priority space for IRQs and threads and offload the entire fixed-priority scheduling to the IC. Thereby, there is no difference between thread and ISR and the all scheduling is performed by the IC. With MultiSloth, they also provide a multi-core variant [20] that, however, only supports partitioned scheduling, where thread/ISRs do not migrate between cores.

VI. Discussion

A. Length of the Critical Path

In our hardware-cost evaluation (see Sec. IV-A), we have seen that the maximum clock frequency decreases with a rising number of IRQs due to the comparator-tree depth and the delivery one IRQ per cycle. While this did not cause problems on 4-cores (@ 25 MHz) system with 128 IRQ sources, it might negatively impact larger configurations. Nevertheless, we could split up the critical path at the `IRQ Deliver` component and insert an intermediate register between the `Max IRQ` and the `Min CPU` component, such that each IRQ delivery happens in two cycles. In cycle one, we find the high-priority IRQ and the low-priority CPU, and deliver it in cycle 2. Thereby, we should be able to trade delivery latency for operation frequency.

However, such a split could bring up new synchronization problems, especially with command processing or priority

changes, as both might invalidate the comparator-tree result. Our initial experiments suggest that such a split of the critical path can be done without requiring significantly more invalidation logic.

B. Bounded Interrupt Latency

Another issue, we want to discuss, is the theoretically unbounded IRQ delivery latency within the IC (see Sec. IV-C). Since CPU-priority updates causes a retraction and re-delivery of the top-m IRQs, an adversarial CPU that changes its priority every $2m - 1$ cycles could, in theory, stall the IRQ delivery forever. While this scenario does not occur during the priority-strict processing of IRQs, we wanted to quantify the problem. We added an adversarial CPU, which changes its priority in a tight loop, to the benchmark from Sec. IV-C. The results show no discernible difference in the average ($\pm 0.33\,\%$) or the maximum ($+1.8\,\%$) IRQ latency. Therefore, we argue that, while our hardware design allows an adversarial CPU to mount a denial-of-service attack, such an attack is not feasible in end-to-end integrations of PSIC.

C. Interface incompatibilities

We designed PSIC to be a drop-in replacement for RISC-V's PLIC with the noticeable exception of IRQ-source masking. While the PLIC can mask IRQ sources on a per-CPU level, PSIC only allows a global masking of IRQ sources. We deliberately broke with the PLIC specification here, as per-CPU masking of IRQ sources is, if combined with global priority-strict multi-core IRQ delivery, not trivial.

Due to masking, each CPU is only target for a subset of all IRQs. Since these subsets can overlap, it becomes unclear that priority-strict IRQ processing actually means. We can either deliver IRQs strictly from high to low if possible, or we can try maximize the priority-sum of delivered IRQs by formulating an IRQ–CPU-assignment optimization problem. In both cases, while we believe that the latter is an NP problem, an integrated view on all pending IRQs and all CPUs would be required, which is hard to implement efficiently in hardware. Thus, we argue, that global IRQ masking is the only acceptable choice for priority-strict multi-core interrupt-controllers. In order to remain compatible with the PLIC interface, we map the same `intMask` bit once for each CPU. Thereby, disabling an IRQ for one CPU disables the source for the whole system and, at least, avoids synchronization problems in legacy software.

D. Software subsystem

The PSIC IC delivers IRQs in a priority-strict fashion, while our software layer adds ISR migration, yielding global preemptive fixed-priority scheduling of ISRs. However, our proposed IC also allows for other operation modes, making PSIC versatile for different system designs. For example, if interruptions are disabled during ISR execution, PSIC performs global non-preemptive fixed-priority scheduling of ISR. If the overhead of ISR threads is undesirable, one can just use the priority-strict IRQ delivery and nest ISRs. While these usage scenarios do not fulfill our definition of priority strictness, they could result in lower system overheads or in lower latencies for high-priority IRQs.

VII. Conclusion

With the transition to multi-core real-time systems, delivering interrupt requests faces the problem of providing a strict and analyzable delivery semantic while solving a system-wide synchronization challenge. We showed that the currently available interrupt controllers induce systematic violations of priority strictness when delivering IRQs, which requires software-based mitigation techniques that prolong the critical IRQ latency. With PSIC, we presented a HW/SW co-design that consists of a priority-strict multi-core interrupt controller with a thin layer of system software to support ISR preemption and migration without the need for further software-based inter-core synchronization. PSIC achieves, at moderate hardware costs, a priority-strict IRQ processing with an average end-to-end, from the external signal until the user-defined ISR starts, latency of 153 cycles, while the worst-case takes 208 cycles for a 4-core RISC-V system. Thereby, PSIC provides a solid base for real-time systems with global IRQ processing at fixed priorities.

Acknowledgments

We would like to thank Matthias Wolf and Christian Bewermeyer for their work on initial drafts of this idea. Furthermore, we also thank our anonymous reviewers for their constructive feedback. This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 391305160 (LO 1719/4-1).

References

[1] *ARM Generic Interrupt Controller Architecture Specification – GIC 3.0/4.0*. ARM Limited. 2017.

[2] J. Adomat, J. Furunas, L. Lindh, and J. Starner. "Real-Time Kernel in Hardware RTU: A Step Towards Deterministic and High-Performance Real-Time Systems". In: *Proc. 8th Euromicro Work. on Real-Time Systems*. 1996. DOI: 10.1109/EMWRTS.1996.557849.

[3] J. Agron, W. Peck, E. Anderson, D. Andrews, E. Komp, R. Sass, F. Baijot, and J. Stevens. "Run-Time Services for Hybrid CPU/FPGA Systems on Chip". In: *Proc. 27th IEEE Intl. Symp. on Real-Time Systems*. 2006. DOI: 10.1109/RTSS.2006.45.

[4] O. Arnold, E. Matus, B. Noethen, M. Winter, T. Limberg, and G. Fettweis. "Tomahawk: Parallelism and Heterogeneity in Communications Signal Processing MPSoCs". In: *ACM Trans. on Embedded Computing Systems* 13.3s (2014). ISSN: 1539-9087. DOI: 10.1145/2517087.

[5] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[6] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. "An Empirical Comparison of Global, Partitioned, and Clustered Multiprocessor EDF Schedulers". In: *Proc. 31st IEEE Intl. Symp. on Real-Time Systems* (San Diego, CA, USA, Nov. 30–Dec. 3, 2010). IEEE Computer Society Press, 2010. ISBN: 978-0-7695-4298-0. DOI: 10.1109/RTSS.2010.23.

[7] R. I. Davis and A. Burns. "A Survey of Hard Real-Time Scheduling for Multiprocessor Systems". In: *ACM Computing Surveys* 43.4 (2011). DOI: 10.1145/1978802.1978814.

[8] E. Dodiu, V. G. Gaitan, and A. Graur. "Improving Commercial RTOS Performance Using a Custom Interrupt Management Scheduling Policy". In: *Proc. Intl. Conf. on Applied Computing*. World Scientific, Engineering Academy, and Society (WSEAS), 2010. ISBN: 9789604742363.

[9] L. E. L. del Foyo and P. Mejia-Alvarez. “Custom Interrupt Management for Real-Time and Embedded System Kernels”. In: *Proc. Embedded Real-Time Systems Implementation Work.* IEEE Computer Society Press, 2004.

[10] L. E. L. del Foyo, P. Mejia-Alvarez, and D. de Niz. “Predictable Interrupt Scheduling with Low Overhead for Real-Time Kernels”. In: *Proc. 12th IEEE Intl. Conf. on Embedded and Real-Time Computing Systems and Applications*. IEEE Computer Society Press, 2006. ISBN: 0-7695-2676-4. DOI: 10.1109/RTCSA.2006.51.

[11] W. Hofer, D. Lohmann, F. Scheler, and W. Schröder-Preikschat. “Sloth: Threads as Interrupts”. In: *Proc. 30th IEEE Intl. Symp. on Real-Time Systems* (Washington, D.C., USA, Dec. 1–4, 2009). IEEE Computer Society Press, 2009. ISBN: 978-0-7695-3875-4. DOI: 10.1109/RTSS.2009.18.

[12] W. Hofer, D. Lohmann, and W. Schröder-Preikschat. “Sleepy Sloth: Threads as Interrupts as Threads”. In: *Proc. 32nd IEEE Intl. Symp. on Real-Time Systems* (Vienna, Austria, Nov. 29–Dec. 2, 2011). IEEE Computer Society Press, 2011. ISBN: 978-0-7695-4591-2. DOI: 10.1109/RTSS.2011.14.

[13] *Intel ® 82093AA I/O Advanced Programmable Interrupt Controller (I/O APIC)*. Intel. 1996.

[14] S. Kleiman and J. Eykholt. “Interrupts as Threads”. In: *ACM SIGOPS Operating Systems Review* 29.2 (1995). ISSN: 0163-5980.

[15] J. Lelli, D. Faggioli, T. Cucinotta, and G. Lipari. “An experimental comparison of different real-time schedulers on multicore systems”. In: *Journal of Systems and Software* 85.10 (2012).

[16] J. W. S. Liu. *Real-Time Systems*. Prentice Hall PTR, 2000. ISBN: 0-13-099651-3.

[17] *MPC5676R Microcontroller Reference Manual. Rev 5*. NXP. 2012.

[18] F. Mauroner and M. Baunach. “EventIRQ: An Event Based and Priority Aware IRQ Handling for Multi-tasking Environments”. In: *2017 Euromicro Conf. on Digital System Design (DSD)*. 2017.

[19] P. McKenney. *A Realtime Preemption Overview*. http://lwn.net/Articles/146861/. 2005.

[20] R. Müller, D. Danner, W. Schröder-Preikschat, and D. Lohmann. “MultiSloth: An Efficient Multi-Core RTOS using Hardware-Based Scheduling”. In: *Proc. 26th Euromicro Conf. on Real-Time Systems* (Madrid, Spain). IEEE Computer Society Press, 2014. ISBN: 978-1-4799-5798-9. DOI: 10.1109/ECRTS.2014.30.

[21] T. Nakano, Y. Komatsudaira, A. Shiomi, and M. Imai. “VLSI Implementation of a Real-Time Operating System”. In: *Proc. Design Automation Conference*. 1997. DOI: 10.1109/ASPDAC.1997.600361.

[22] G. Parmer and R. West. “Predictable Interrupt Management and Scheduling in the Composite Component-Based System”. In: *Proc. 29th IEEE Intl. Symp. on Real-Time Systems*. IEEE Computer Society Press, 2008. ISBN: 978-0-7695-3477-0. DOI: 10.1109/RTSS.2008.13.

[23] P. Patel, M. Vanga, and B. B. Brandenburg. “TimerShield: Protecting High-Priority Tasks from Low-Priority Timer Interference (Outstanding Paper)”. In: *Proc. Real-Time and Embedded Technology and Applications Symp. (RTAS)*. 2017.

[24] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, et al. “T-CREST: Time-predictable Multi-Core Architecture for Embedded Systems”. In: *Journal of Systems Architecture* 61 (2015).

[25] J. Strnadel. “Load-adaptive monitor-driven hardware for preventing embedded real-time systems from overloads caused by excessive interrupt rates”. In: *Intl. Conf. on Architecture of Computing Systems*. Springer. 2013.

[26] *TC29x B-step - User’s Manual, V1.3 2014-12*. Infineon Technologies AG. 2014.

[27] *The RISC-V Instruction Set Manual – Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified”*. 2019.

[28] *The RISC-V Instruction Set Manual, Volume I: Unpriviledged ISA – Document Version 20191213*. 2019.

[29] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee. “FlexPRET: A processor platform for mixed-criticality systems”. In: *Proc. 20th IEEE Intl. Symp. on Real-Time and Embedded Technology and Applications*. IEEE Computer Society Press. DOI: 10.1109/RTAS.2014.6925994.