IRQ Coloring: Mitigating Interrupt-Generated Interference on ARM Multicore Platforms

Diogo Costa
Centro ALGORITMI / LASI,
University of Minho, Portugal

Daniel Oliveira
Centro ALGORITMI / LASI,
University of Minho, Portugal

Bruno Morelli
Huawei Pisa Research Center,
Pisa, Italy

Fabrizio Tronci
Huawei Pisa Research Center,
Pisa, Italy

Sandro Pinto
Centro ALGORITMI / LASI,
University of Minho, Portugal

Luca Cuomo
Huawei Pisa Research Center,
Pisa, Italy

Ida Maria Savino
Huawei Pisa Research Center,
Pisa, Italy

José Martins
Centro ALGORITMI / LASI,
University of Minho, Portugal

Alessandro Biasci
Huawei Pisa Research Center,
Pisa, Italy

Abstract

Mixed-criticality systems, which consolidate workloads with different criticalities, must comply with stringent spatial and temporal isolation requirements imposed by safety-critical standards (e.g., ISO26262). This, per se, has proven to be a challenge with the advent of multicore platforms due to the inner interference created by multiple subsystems while disputing access to shared resources. With this work, we pioneer the concept of Interrupt (IRQ) coloring as a novel mechanism to minimize the interference created by co-existing interrupt-driven workloads. The main idea consists of selectively deactivating specific ("colored") interrupts if the Quality of Service (QoS) of critical workloads (e.g., Virtual Machines) drops below a well-defined threshold. The IRQ Coloring approach encompasses two artifacts, i.e., the IRQ Coloring Design-Time Tool (IRQ DTT) and the IRQ Coloring Run-Time Mechanism (IRQ RTM). In this paper, we focus on presenting the conceptual IRQ coloring design, describing the first prototype of the IRQ RTM on Bao hypervisor, and providing initial evidence about the effectiveness of the proposed approach on a synthetic use case.

2012 ACM Subject Classification Computer systems organization → Real-time system specification; Computer systems organization → Embedded software

Keywords and phrases IRQ coloring, Interrupt Interference, Mixed-Criticality Systems, Hypervisors, Bao, Arm

Digital Object Identifier 10.4230/OASIcs.NG-RES.2023.2

Funding *Diogo Costa*: Supported by Centro Algoritmi grant IRQ-COL-HYP-HUAWEI_01.
Daniel Oliveira: Supported by FCT grant 2020.04585.BD.
José Martins: Supported by FCT grant SFRH/BD/138660/2018.

1 Introduction

Currently, two major trends pose significant challenges for the certification of Mixed-criticality Systems (MCS) [13]. Firstly, with the increasing digitalization trend, there is a need to integrate an ever-growing number of rich functionalities for connectivity, visualization, and monitoring. Rich features have to co-exist with safety-critical workloads, and virtualization

technology – due to the proven advantages for the size, weight, power, and cost (SWaP-C) – has been leveraged to provide the required spatial isolation [23, 24, 5, 11, 20]. Secondly, there is a well-established trend toward multicore. Modern high-end embedded computing platforms (typically powered by Arm Cortex-A) have evolved to highly heterogeneous designs that host multiple cores, featuring complex memory hierarchies and a myriad of accelerator such as general-purpose graphic processing units (GPUs), neural-processing units (NPUs) and even Field-Programmable Gate Array (FPGA). Although existing hypervisors have naturally evolved to support multicore designs, the problem is that the temporal isolation guarantees are generally weak due to the absence of mechanisms in the overall design to minimize the interference generated by shared microarchitectural resources [9].

The problem and difficulties deriving from the reciprocal interference generated by the contention of shared microarchitectural computational resources, such as (i) caches, (ii) bus, and (iii) main memory is a well-understood problem among the real-time systems community [11, 20, 16, 28, 27, 29, 2, 26, 7, 18, 17]. It is widely-recognized that without proper resource management strategies, interference can introduce variable delays that may hamper the timing predictability and real-time guarantees [1, 4, 3], compromising the desired Freedom From Interference (FFI) for MCSs. This issue is not linked to a particular piece of software, i.e., operating system (OS) and/or hypervisor, but rather a fundamental problem on the overall design of the lowest layer of software of the system stack. This problem is further exacerbated by interrupts and interrupt-driven workloads. Interrupts are typically asynchronous, and even synchronous events can divert the overall execution flow. Interrupt handlers normally have a different code locality, inherently leading to stress the shared components due to the expected last-level cache (LLC) misses and concurrent accesses to main memory. Even worst, a storm of interrupts (i.e., extremely high interrupt frequency) can be triggered for different reasons, leading to very effective Denial-of-Service (DoS) attacks.

Most of the proposed techniques to minimize interference created by shared resources in multicore real-time systems focus on (i) cache partitioning (via locking or coloring) [11, 20, 16, 15, 21], and (ii) memory throttling [28, 27, 2, 21, 6, 8]. Cache partitioning is a well-established technique that splits and assigns subsets of the shared cache to a specific OS process or Virtual Machine (VM). Cache locking avoids the eviction of cache lines by marking them as locked; cache coloring segments the available cache by reserving specific cache sets or ways to given cores. Memory throttling is a technique that limits the number of memory accesses of a specific workload in a given time window. In the case of virtualization, static partitioning hypervisors are the zeitgeist to implement such techniques e.g., Bao [20], Jailhouse [24], Xen Dom0-less [14], all implement cache coloring.

Despite the recognized efforts of the academic community, existing mechanisms are not perfect in terms of effectiveness and present several limitations. Firstly, cache locking support was deprecated and is not available in today's platforms powered by Armv8-A processors (and is not expected to be supported in next-generation platforms powered by Armv9-A and Armv8-R). Secondly, cache coloring has several drawbacks: (i) requires a virtual memory infrastructure (i.e., the existence of an MMU); (ii) precludes the use of (2 MiB) huge pages (which helps minimizing the overhead of the second stage translation); (iii) can impact the performance; and (iv) leads to significant memory waste and fragmentation. Finally, memory throttling completely suspends the CPU when a specific memory access threshold is reached in a given time window, thus not providing intermediate states or well-defined degradation modes. Furthermore, none of the existing mechanisms take interrupts into consideration.

In this paper, we pioneer and introduce the concept of Interrupt coloring (a.k.a. IRQ coloring), a novel mechanism to minimize the interference created by co-existing interrupt-driven workloads[1] and mitigate the effect of cascading failures when FFI cannot be completely guaranteed. The overall IRQ Coloring concept encompasses two artifacts: (i) the IRQ Coloring Design-Time Tool (IRQ DTT) and (ii) IRQ Coloring Run-Time Mechanism (IRQ RTM). The basic idea consists of selectively deactivating (or deferring) specific ("colored") interrupts if the QoS of critical workloads drops below a specific threshold. By selectively masking interrupts per the online assessment of the QoS of critical workloads, we provide fine-grained control to mitigate interference on critical workloads without fully suspending non-critical workloads. In this paper, we focus on describing the overall IRQ coloring concept, prototyping the IRQ RTM on the Bao hypervisor, and providing evidence about the effectiveness of the proposed approach on a synthetic use case mimicking an automotive application scenario. To the best of our knowledge, we are the first to propose this concept. Huawei has already filled and submitted a patent application.

2 Related Work

Several mechanisms to minimize interference have been proposed by academia and the research community. Table 1 summarizes and puts into perspective several works published in real-time venues. All works are compared across six dimensions: (i) target computer architecture (Arch); (ii) implementation leveraging COTS hardware (COTS); (iii) target system software component (System); (iv) target shared resource (Resource); (v) resource partitioning/budgeting at design-time (Static); and (vi) resource partitioning/budgeting and optimization at run-time (Dynamic). Next, we overview the related work, grouping them based on the target resource, i.e., last-level cache (LLC), memory bus (e.g., Dynamic Random Access Memory (DRAM) or SRAM), or both (cache and memory).

Table 1 Gap Analysis Table. ●: "yes". ○: "no".

	Arch	COTS	System	Resource	Static	Dynamic
MemGuard (2013) [29]	x86	●	OS	DRAM	○	●
Mancuso et al. (2013) [19]	Armv7-A	●	OS	LLC	●	●
Hassan et al. (2014) [12]	x86	○	OS	LLC	●	○
PALLOC (2014)[27]	x86	●	OS	DRAM	●	○
Kim et al. (2017) [15]	x86 & Armv7-A	●	Hyp.	LLC	●	○
Modica et al. (2018) [21]	Armv7-A	●	Hyp.	LLC & DRAM	●	●
Crespo et al. (2018) [6]	PowerPC	●	Hyp.	DRAM	●	●
Pinto et al. (2019) [22]	Armv8-M	●	Hyp.	SRAM	●	○
Kloda et al. (2019) [16]	Armv8-A	●	Hyp.	LLC & DRAM	●	○
Bao (2020) [20]	Armv8-A	●	Hyp.	LLC	●	○
BRU (2020) [8]	RISC-V	○	OS	DRAM	○	●
DNA (2021) [10]	x86	●	Hyp.	LLC+DRAM	●	●
IRQ Coloring	Armv8-A (Armv8-R)	●	Hyp.	IRQs	●	●

Cache Partitioning. Multicore platforms typically include a shared LLC and one or more levels of private caches. The cache partitioning technique splits and assigns subsets of the shared cache to a specific workload. Cache locking avoids the eviction of cache lines by

[1] processes in the case of OS and VMs in the case of hypervisor

marking them as locked, while cache coloring segments the available cache by reserving specific cache sets or ways to specific cores. R. Mancuso et al. [19] proposed a mechanism that introduces a novel mechanism known as "Colored Lockdown" by combining coloring and locking techniques. Kim et al. [15] proposed a multicore virtualization cache management system that uses the hypervisor page coloring mechanism to assign portions of the cache to VMs. M. Hassan et al. [12] address the problem of maintaining cache coherence in multicore real-time systems by modifying the Modified-Shared-Invalidate [25] coherence protocol. J. Martins et al. [20] implemented built-in support for cache coloring on Bao.

Memory Bandwidth Partitioning. Aiming to achieve temporal isolation through memory bandwidth regulation, H. Yun et al. [29] proposed MemGuard. To improve isolation and real-time performance, H. Yun et al. [27] proposed a mechanism that allocates private DRAM banks. A. Crespo et al. [6] proposed a controller technique, at the hypervisor level, to manage the execution of critical partitions for PowerPC multicore platforms. F. Farshchi et al. [8] presented a Bandwidth Regulation Unit (BRU), a customized RISC-V plug-and-play hardware module for per-core memory bandwidth control at fine-grained time intervals. From a different perspective, Pinto et al. [22] developed a lightweight hypervisor and implemented a static predictable shared resource management infrastructure for low-end Armv8-M microcontrollers.

Cache & Memory Bandwidth Partitioning. The previous works focus on mechanisms that individually target the Last-Level Cache (LLC) or the main memory. However, minimizing the interference impact in a single microarchitectural component is not enough. Hence, several authors attempted to provide methods that target the complete memory hierarchy, i.e., LLC and main memory. Modica et al. [21] proposed a cache coloring-based technique to achieve spatial isolation. Regarding the DRAM memory controller, the authors implemented a memory bandwidth reservation technique combined with the hypervisor's scheduling logic to enhance temporal isolation. Kloda et al. [16] introduced a framework of software-based techniques to enhance memory access determinism in high-performance embedded systems. The authors proposed a Direct memory access (DMA)-friendly cache coloring combined with an invalidation-driven allocation technique. Recently, R. Gifford et al. [10] proposed a solution to mitigate two undesirable outcomes in current MCS: latency induced by shared resource interference and Worst-Case Execution Time (WCET) of critical tasks. Furthermore, the authors presented two techniques to mitigate the identified challenges: dynamic allocation (DNA) and deadline-aware allocation (DADNA).

3 Motivation: Interrupt-generated Interference in MCSs

In the context of MCS, interference generated by co-located interrupt-driven workloads is a particularly overlooked topic. Interrupts are typically used to notify the CPU about the occurrence of sporadic events, without requiring the CPU to stall while polling for that event. In the meantime, the CPU is free to perform any additional required computational operation. Generically, upon the occurrence of an interrupt (and omitting the low-level details of the overall interrupt entry process), the CPU execution is redirected to the so-called interrupt handler, which acknowledges the reception of the interrupt and then invokes an event-dependent piece of code. Typically, in safety-critical systems, interrupts are linked to application workloads, which are dispatched upon the occurrence of a particular event. For example, in Industrial Control Systems (ICS), the implementation of a PID controller, or in automotive systems, an antilock braking system.

The problem. Interrupts are typically asynchronous, and even synchronous events lead to unpredictable diversions in the overall computational execution flow. Upon the occurrence of an interrupt, the execution flow is redirected to the respective interrupt handler, which typically has a completely different code locality than the main execution path. Furthermore, interrupts are typically linked to application workloads, which are dispatched upon the occurrence of a particular event. This inherently stresses the microarchitectural shared components due to the expected LLC misses and subsequent accesses to the main memory. Even worst, a storm of interrupts can be triggered for different reasons (e.g., device driver bug, malfunction in a particular hardware device), which can create a DoS attack.

The evidence. We performed a few experiments to collect evidence supporting the identified problem. We mounted a synthetic use case of a system consisting of Bao hypervisor and two VMs running atop: a critical ASIL-D VM, and a QM VM. For the ASIL-D VM, we use a custom baremetal application that continuously writes into a buffer with the size of the LLC (1 MiB). A periodic CPU timer interrupt triggers this application. For the other QM VM, we use the very same baremetal application, triggered continuously by a software-generated interrupt. For the ASIL-D VM, we assigned 1 CPU, while for the QM VM we assigned 1, 2, or 3 CPUs, depending on the amount of interference we want to create (1 Interf VM, 2 Interf VM, and 3 Interf VM, respectively). Figure 1 depicts the assessed results. We collected the workload execution time and IRQ handling time, which corresponds to the interrupt latency, for the critical ASIL-D VM. The interference with 3 CPUs (3 Interf VM) can impact the workload execution time of the ASIL-D VM up to 2.48x.

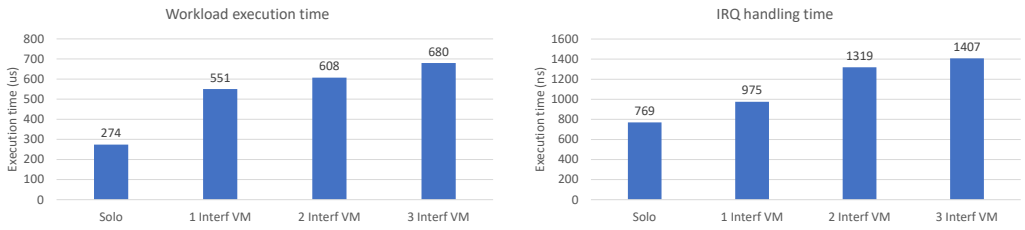

Figure 1 Interrupt-generated Interference - workload execution time and IRQ handling time with multiple VMs.

4 IRQ Coloring

IRQ coloring is a new concept that dictates that interrupts assigned to workloads (e.g., VMs) are classified according to a specific criticality level. The basic idea consists of selectively deactivating/deferring interrupts of non-critical workloads if the QoS of critical workloads drops below a specific threshold. By selectively masking interrupts per the online assessment of the QoS of critical workloads, it is expected that the overall interference from non-critical VMs to critical VMs is mitigated without fully suspending non-critical workloads.

Conceptual Design. Figure 2 illustrates the conceptual design of the IRQ Coloring technique, which follows a budget-based approach. A set of budgets of interrupts (B_i) are assigned to each workload for a well-defined period of time (P). During the period (P), each VM can trigger a certain number of interrupts; however, if the VM exceeds the specified budget (B_i), the degradation mode is updated, and some interrupts are disabled (1). Since the

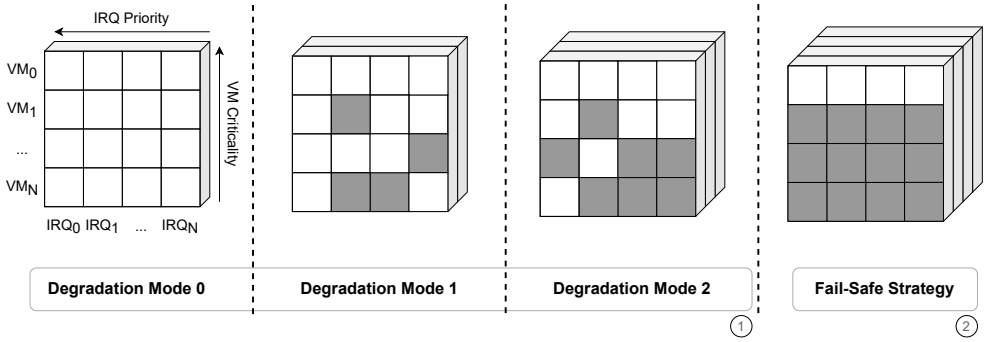

Figure 2 IRQ coloring conceptual design.

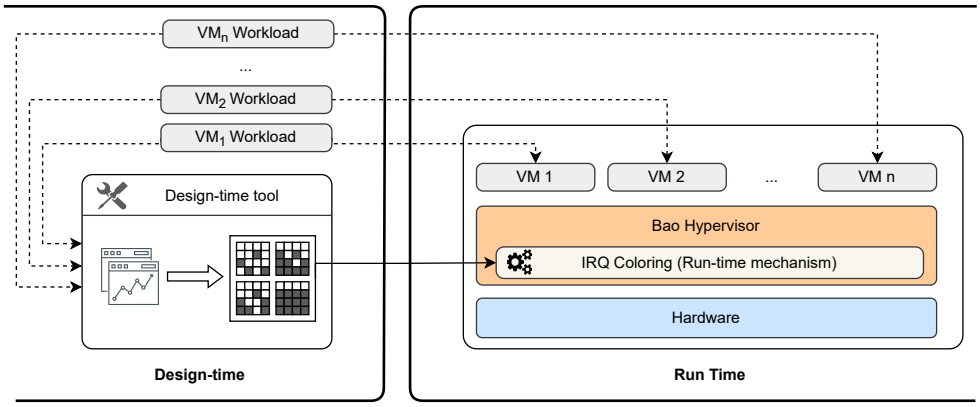

Figure 3 IRQ coloring system overview.

impact of each interrupt (assigned to a particular VM) in the overall QoS of the system is not equal and linear, the weight of each interrupt in the overall budget B_i will be measured by assessing specific QoS metrics. The conceptual design takes this into consideration and is reflected in Figure 2). Lastly, if applying different degradation modes is not enough to guarantee the QoS of the higher-criticality VM, the system will enter fail-safe mode. At this stage, only interrupts from the higher-criticality VM are kept enabled (2).

System Overview. Defining which and when interrupts must be disabled is the main research question of the IRQ Coloring technique. The design goals encompass: (i) achieving the required QoS on higher criticality VMs, (ii) maintaining intermediate execution of lower criticality VMs (intermediate states under specific degradation modes), and (iii) minimizing the performance impact incured by the overall mechanism. To achieve these three goals, we conceived a system with two major components, with the bulk of logic performed at design time. Figure 3 presents the high-level system view, encompassing the IRQ Coloring Design-Time Tool (IRQ DTT) and the IRQ Coloring Run-Time Mechanism (IRQ RTM).

1. IRQ Coloring Design-Time Tool (IRQ DTT). The IRQ DTT goal is twofold. Firstly, based on the target VM workload, set the profile of each interrupt-driven workload, which helps estimate the worst-case execution time (WCET) by providing information about the execution time and the microarchitectural state (i.e., caches, shared bus). Secondly, based on the established profile of the workload and assigned interrupts altogether with the

specification of VMs criticality, produce an optimized configuration table (representing the masking map to be enforced in each degradation mode) which will feed the IRQ RTM. The output of the DTT will then be used to feed the RTM implemented at the hypervisor level, and it consists of two artifacts: (i) the budgets assigned to each degradation mode of the diverse VMs, and (ii) the masking maps to be used in the multiple degradation modes (i.e., the content of the optimized configuration table).

2. IRQ Coloring Run-Time Mechanism (IRQ RTM). The IRQ RTM, implemented as a mechanism at the hypervisor level, will mainly collect specific metrics from the hardware performance counters (e.g. Performance Monitor Unit (PMU)), and based on the optimized table produced from the IRQ DTT, will selectively disable interrupts upon the occurrence of specific events. Next, we describe the IRQ RTM implemented in Bao in more detail.

4.1 IRQ Coloring Run-Time Mechanism (IRQ RTM)

The implementation of the IRQ RTM has two main assumptions, which we highlight below.

Assumption 1. Workloads Profiling

The interference generated by co-located VMs running onto the same hardware platform is dependent on the VMs' workloads. We assume that workloads are available, known a priori, profiled offline, and this data is directly or indirectly passed to the IRQ DTT.

Assumption 2. Masking Maps

We assume that the IRQ DTT, based on the profile of the VMs workloads, produces an optimized configuration table (a.k.a. masking map) with "colored" interrupts and associated budgets.

The IRQ RTM relies on a budget-based implementation, which means that each VM can trigger a given number of interrupts in a given period of time (P). Typically, these approaches assume that a VM can process workloads until the defined budget is exhausted. After that, typically the CPU enters idle mode. However, IRQ coloring points to intermediate guarantees, i.e., VM interrupts will be gradually disabled in order to minimize the generated interference created on the critical VM. In this way, instead of assigning a budget to each VM, we assign a buffer of budgets (i.e., a buffer with D values, each representing an activation point of a different degradation mode). Thus, the system configuration must contemplate the setup of the table $B_{N,D}$, where N corresponds to the number of co-existing VMs into the system, and D corresponds to the total number of degradation modes. The high-level implementation of the IRQ RTM is shown in Algorithm 1.

Figure 4 depicts the IRQ RTM architecture, implemented as a plug-in mechanism in Bao. The IRQ RTM actuates between the GIC distributor (GICD) and the CPU Interfaces (GICC). To simplify the run-time operation, each CPU uses the PMU to track the number of triggered interrupts, triggering an interrupt, at the hypervisor level, when the counter overflows, i.e., when the assigned budget (B) is exceeded. When the PMU interrupt is triggered, the IRQ RTM is in charge of masking a set of interrupts based on the masking map generated by the IRQ DTT. At this point, the run-time mechanism is triggered, leading to the update of the degradation mode and the masking of the interrupts corresponding to the degradation mode map (IRQ_MAP_D). The process repeats until processing all the

Algorithm 1 IRQ Coloring RTM - Implementation.

funtion periodic_timer_handler;
begin
for $VM = 1, 2, \ldots$ **do**
    $VM_{Degradation\_mode} = 0$
    Set PMU to generate overflow interrupt at VM_{B_0}
    for $IRQ = 1, 2, \ldots$ **do**
        Unmask IRQ
    end for
end for
Re-schedule timer period

funtion pmu_overflow_interrupt_handler;
begin
for $IRQ = 1, 2, \ldots \in IRQ\_MAP_{Degradation\_mode}$ **do**
    Mask IRQ
end for
$VM_{Degradation\_mode} \leftarrow VM_{Degradation\_mode} + 1$
if $VM_{Degradation\_mode} < Max\_Degradation\_modes$ **then**
    Set PMU to generate overflow interrupt at $B_{VM,Degradation\_mode}$
end if

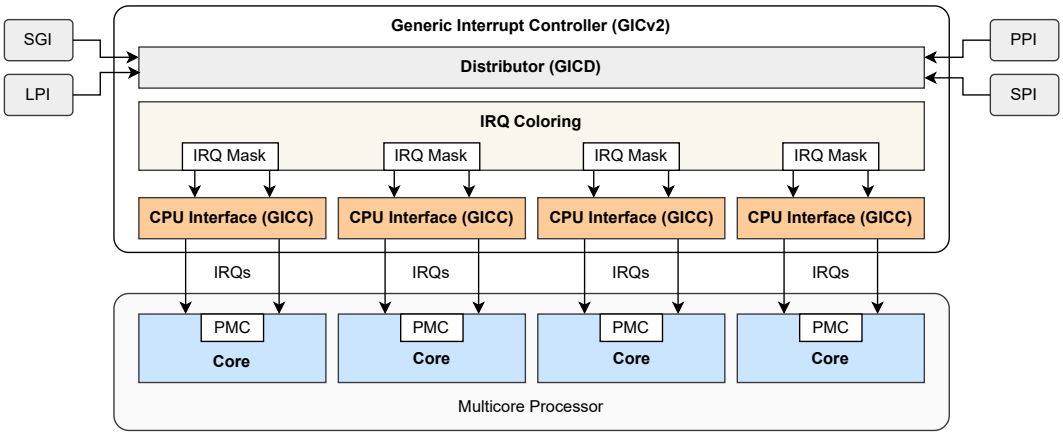

Figure 4 IRQ Coloring RTM: system architecture.

D degradation modes. If a non-critical VM reaches this point, the IRQ RTM triggers the fail-safe strategy, i.e., all interrupts from all system VMs, except the most critical one, are disabled. In this case, the interrupts are re-enabled when the periodic timer expires.

5 Evaluation

In this section we describe the evaluation setup (Section 5.1) and present and discuss the evaluation results (Section 5.2).

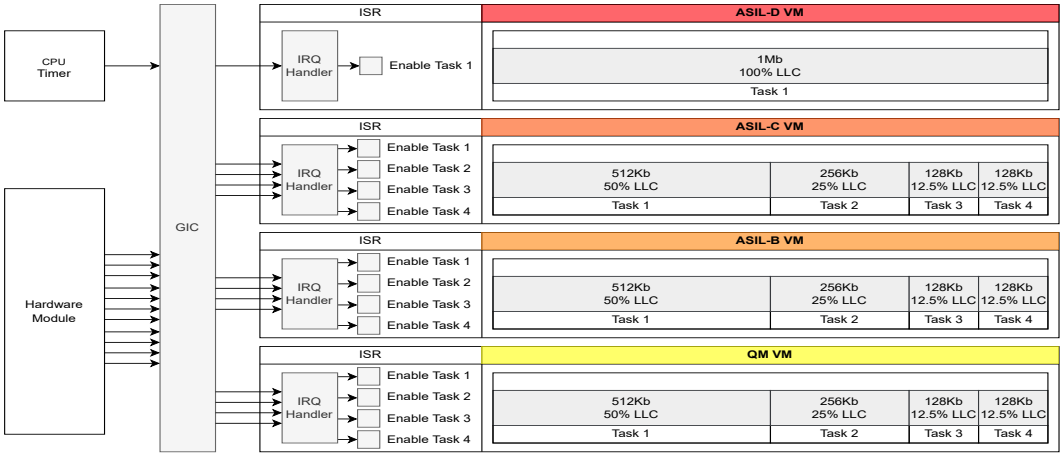

Figure 5 IRQ Coloring evaluation (synthetic) use case.

5.1 Evaluation Setup

Hardware Platform. Experiments were carried out on a Xilinx ZCU104 evaluation board, featuring a Zynq Ultrascale+ ZU7EV SoC. This platform features a quad-core Arm Cortex-A53 running at 1.2GHz. Each CPU has a private 32KiB L1 instruction and data caches and an unified L2 1MiB cache. The cluster features the GIC-400 (GICv2).

Use Case. To evaluate the raw effectiveness of the IRQ coloring mechanism, we mounted a synthetic use case, aiming at mimicking a real-world automotive ECU consisting of four different sub-systems: (i) a critical ASIL-D workload, emulating an electric power steering system; (ii) an ASIL-C workload, emulating an active suspension system or an engine management system; (iii) an ASIL-B workload, emulating for example a radar cruise control; and (iv) a QM workload, emulating a non-critical third-party software. We implemented the four VMs on top of the Bao [20]. Furthermore, to mimic the I/O interrupt generation, we have implemented a custom hardware device on the programmable logic of the Zynq UltraScale+ SoC, which can trigger up to 16 simultaneous interrupts.

VM Workload. The workload of each VM is summarized in Figure 5. For the ASIL-D VM, we use a custom baremetal application that continuously writes into a buffer with the size of the LLC (1 MiB). This application is triggered by a periodic CPU timer interrupt at each 500 us. For the other VMs, i.e., ASIL-C, ASIL-B, and QM, we use an identical baremetal application with a small difference. Each VM has assigned four different interrupts, all triggered by the custom hardware module deployed on the programmable logic of the Zynq UltraScale+ SoC. In this case, rather than writing into the entire buffer, each interrupt triggers the workload that writes in a subset of the buffer. To evaluate the effect of different workloads, the buffer is partitioned into four parts: (i) one with 512KiB (50% of the LLC); (ii) one with 256KiB (25% of the LLC); and (iii) two with 128KiB each (12.5% of the LLC). Since the Cortex-A cluster of the ZCU104 has 4 CPUs, we assign one CPU to each VM.

Measurement Tools. We use the Arm PMU to collect microarchitectural events on benchmark execution. The selected events include execution cycle count, data L1 and L2 cache misses and cycles of bus accesses. The PMU cycle counter is used to calculate the execution time.

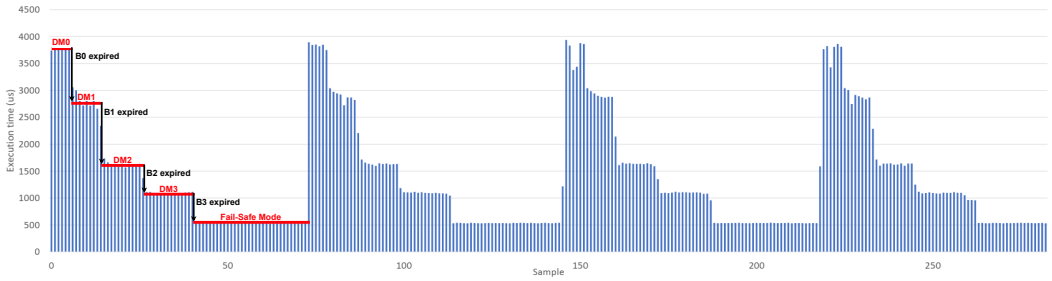

Figure 6 IRQ RTM - Execution time of ASIL-D workload for multiple degradation modes.

5.2 Evaluation Results

Intermediate degradation modes. To validate the intermediate degradation modes implemented with the IRQ RTM, we used a simplified version of the use case described above. For this particular experiment, we have used only the ASIL-D and QM VMs, with respective identified workloads. We have also defined four degradation modes: (i) degradation mode 1 (3 interrupts on QM VM active), which is activated after spending a total budget of 1000 interrupts; (ii) degradation mode 2 (2 interrupts on QM VM active), which is activated once the total budget of 4000 is reached; (iii) degradation mode 3 (1 interrupt on the QM VM active), which is activated after triggering 5000 interrupts; and, finally, (iv) fail-safe strategy (all interrupts are disabled on the QM VM) which is activated after triggering another 5000 interrupts. The budgets are re-established after a well-defined period of 150ms. Figure 6 presents the execution time of the ASIL-D VM workload. We started by assessing the ASIL-D VM running without interference, which corresponds to the baseline. For this case, the workload execution time was, on average, 547 us. By enabling the QM VM interrupts, the average ASIL-D workload execution time increases by 6.99x (3823 us) - DM0. Then, after expiring the first budget (B0), the first degradation mode is activated, reducing the performance degradation by 26.8% (2796us) - DM1. Then, after activating additional degradation modes, the ASIL-D workload performance is boosted due to reduction of the QM VM interference. For instance, when the budget of the degradation mode 1 expires, the second degradation mode - DM2 - is activated, allowing a reduction of the relative performance overhead of 57.32% (1632us); when the third degradation mode - DM3 - is activated, it reaches a reduction of 70.87% (114us). Ultimately, triggering the fail-safe strategy brings the ASIL-D workload performance to native execution, i.e., 550 us.

Relative performance overhead. After validating the effectiveness of intermediate degradation modes, we resort to the original use case with four VMs to understand the average performance impact of the IRQ coloring (*+IRQ_col*) compared to the vanilla system (baseline) and the cache coloring technique (*+cc*). For the cache coloring, we assigned two colors to each partition (25% of the L2 cache to each VM). Figure 7 depicts the ASIL-D workload execution cycles, as well as a few microarchitectural events, i.e., L1 misses, L2 misses, and bus cycles. We can draw two main conclusions. Firstly, in contrast to the cache coloring, the IRQ coloring technique does not incur any noticeable impact on the overall performance of the ASIL-D VM. Secondly, when the ASIL-D VM is under the interference of the QM VM, the average performance overhead is considerably smaller in the case of the IRQ coloring, which indicates that this technique is more effective than the cache coloring. As expected, per the results collected for the microarchitectural events, this is explained by the reduced number of L1 and L2 cache misses (the contention on the L1 and L2 caches is smaller), as well as the reduced number of accesses to the main memory.

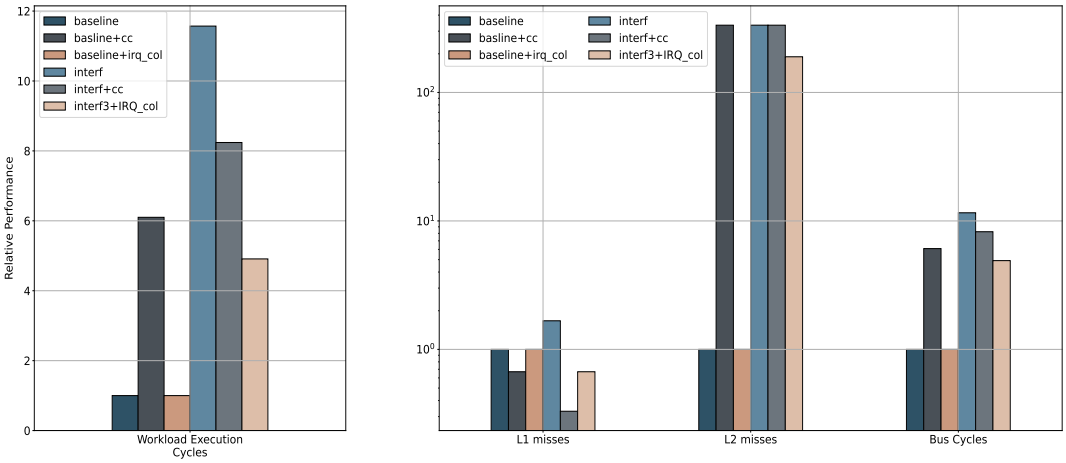

Figure 7 Average Relative performance overhead and microarchitectural events for the vanilla system, for the cache coloring, and IRQ coloring.

6 Conclusion

With this paper, we propose the concept of Interrupt (IRQ) coloring as a novel mechanism to minimize the interference created by co-existing interrupt-driven workloads. We focused on presenting the conceptual IRQ coloring design, describing the prototype of the IRQ RTM on Bao, and evaluating the implemented mechanisms on a synthetic use case. Preliminary results have demonstrated advantages compared to other state-of-the-art techniques (e.g., cache coloring), and findings are encouraging additional research to advance the maturity of the technique, as well as a comprehensive evaluation under more realistic setups. Additionally, we are currently designing the DTT infrastructure. The full combination and integration of RTM and DTT have the potential to further unlock novel designs and optimize and explore trade-offs for reducing interference in multicore platforms. As part of our roadmap, we also plan to iterate and complete the formal model for the IRQ coloring mechanism.

References

1 Jaume Abella, Carles Hernández, Eduardo Quiñones, Francisco J Cazorla, Philippa Ryan Conmy, Mikel Azkarate-Askasua, Jon Perez, Enrico Mezzetti, and Tullio Vardanega. Wcet analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, 2015.

2 Michael G Bechtel and Heechul Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention, 2019.

3 Francisco J Cazorla, Leonidas Kosmidis, Enrico Mezzetti, Carles Hernandez, Jaume Abella, and Tullio Vardanega. Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. *ACM Computing Surveys (CSUR)*, 52(1):1–35, 2019.

4 Francisco J Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, et al. Proartis: Probabilistically analyzable real-time systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):1–26, 2013.

5 Jon Perez Cerrolaza, Roman Obermaisser, Jaume Abella, Francisco J Cazorla, Kim Grüttner, Irune Agirre, Hamidreza Ahmadian, and Imanol Allende. Multi-core devices for safety-critical systems: A survey. *ACM Computing Surveys (CSUR)*, 53(4):1–38, 2020.

6 Alfons Crespo, Patricia Balbastre, José Simó, Javier Coronel, Daniel Gracia Pérez, and Philippe Bonnot. Hypervisor-based multicore feedback control of mixed-criticality systems. *IEEE Access*, 6:50627–50640, 2018.

7 Dakshina Dasari, Benny Akesson, Vincent Nelis, Muhammad Ali Awan, and Stefan M Petters. Identifying the sources of unpredictability in cots-based multicore systems. In *8th IEEE international symposium on industrial embedded systems (SIES)*, pages 39–48, 2013.

8 Farzad Farshchi, Qijing Huang, and Heechul Yun. Bru: Bandwidth regulation unit for real-time multicore processors. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 364–375, 2020.

9 Gabriel Fernandez, Jaume Abella Ferrer, Eduardo Quiñones, Christine Rochange, Tullio Vardanega, and Francisco Javier Cazorla Almeida. Contention in multicore hardware shared resources: Understanding of the state of the art. In *14th International Workshop on Worst-Case Execution Time Analysis*, pages 31–42, 2014.

10 Robert Gifford, Neeraj Gandhi, Linh Thi Xuan Phan, and Andreas Haeberlen. Dna: Dynamic resource allocation for soft real-time multicore systems. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 196–209, 2021.

11 Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In *31st Euromicro Conference on Real-Time Systems (ECRTS)*, volume 133, pages 27:1–27:25, 2019.

12 Mohamed Hassan, Anirudh M Kaushik, and Hiren Patel. Predictable cache coherence for multi-core real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 235–246, 2017.

13 Thomas A Henzinger and Joseph Sifakis. The embedded systems design challenge. In *International Symposium on Formal Methods*, pages 1–15, 2006.

14 Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. In *5th IEEE Consumer Communications and Networking Conference*, pages 257–261, 2008.

15 Hyoseung Kim and Ragunathan Rajkumar. Predictable shared cache management for multi-core real-time virtualization. *ACM Transactions on Embedded Computing Systems (TECS)*, volume 17, 2017.

16 Tomasz Kloda, Marco Solieri, Renato Mancuso, Nicola Capodieci, Paolo Valente, and Marko Bertogna. Deterministic memory hierarchy and virtualization for modern multi-core embedded systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–14, 2019.

17 Ondrej Kotaba, Jan Nowotsch, Michael Paulitsch, Stefan M Petters, and Henrik Theiling. Multicore in real-time systems–temporal isolation challenges due to shared resources. In *16th Design, Automation & Test in Europe Conference and Exhibition*, 2013.

18 Andreas Löfwenmark and Simin Nadjm-Tehrani. Understanding shared memory bank access interference in multi-core avionics. In *16th International Workshop on Worst-Case Execution Time Analysis*, 2016.

19 Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54, 2013.

20 José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems. In *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*, volume 77, pages 3:1–3:14, 2020.

21 Paolo Modica, Alessandro Biondi, Giorgio Buttazzo, and Anup Patel. Supporting temporal and spatial isolation in a hypervisor for arm multicore platforms. In *IEEE International Conference on Industrial Technology (ICIT)*, pages 1651–1657, 2018.

22 Sandro Pinto, Hugo Araujo, Daniel Oliveira, Jose Martins, and Adriano Tavares. Virtualization on trustzone-enabled microcontrollers? voilà! In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 293–304. IEEE, 2019.

23 Sandro Pinto, Jorge Pereira, Tiago Gomes, Adriano Tavares, and Jorge Cabral. Ltzvisor: Trustzone is the key. In *29th Euromicro Conference on Real-Time Systems (ECRTS)*, 2017.

24 Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. Look mum, no vm exits! (almost). *arXiv preprint*, 2017. `arXiv:1705.06932`.

25 Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. In *Synthesis lectures on computer architecture*, volume 6, pages 1–212, 2011.

26 Theo Ungerer, Francisco Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quinones, Mike Gerdes, Marco Paolieri, Julian Wolf, et al. Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010.

27 Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, 2014.

28 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *24th Euromicro Conference on Real-Time Systems*, pages 299–308, 2012.

29 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.