PRL: Standardizing Performance Monitoring Library for High-Integrity Real-Time Systems

Jeremy Giesen[*,†], Enrico Mezzetti[*], Jaume Abella[*], Francisco J. Cazorla[*]
[*]Barcelona Supercomputing Center (BSC)
[†]Universitat Politecnica de Catalunya

***Abstract*—The use of complex processors is becoming ubiquitous in High-Integrity Systems (HIS). To deal with processor's increased complexity, Performance Monitoring Counters (PMCs) are increasingly used to reason on software behavior and provide the necessary evidence to support software certification. However, the use of PMCs in HIS is relatively recent and hence far from being standardized. As a result, software engineers are forced to resort to highly-customized, low-level programming of platform-specific PMC control registers, which is both error prone and time consuming. To cover this gap, we propose building on the PAPI library, a standardized performance monitoring solution in the mainstream domain, and develop a PMC Reading Library (PRL) for configuring and collecting traceable events while capturing HIS specific requirements and peculiarities. We instantiate PRL in a reference automotive configuration to show that PRL meets key HIS requirements: negligible footprint, limited and predictable overhead, and accuracy collecting hardware events by filtering out the impact of interrupts and context switches.**

***Index Terms*—Performance counters, embedded systems**

I. Introduction

The use of multicores[1] as main computing solution is consolidating in HIS, including automotive and avionics. This trend is driven by the use of performance-demanding software applications involved in most new cutting-edge functionalities of HIS embedded products, like AD (autonomous driving).

HIS undergo a strict verification and validation (V&V) process to guarantee their deployment behavior is correct. PMCs, present in most modern processors in the performance monitoring unit (PMU), are gaining traction in HIS to achieve both functional and timing V&V for software running on multicore processors. PMCs can, for instance, help analyzing functional issues related to the coherence [15], [16]. More generally, measurement-based and hybrid timing analysis tools are transitioning from exclusively timing information to include trackable events [19]. Overall, PMCs are at the heart of requirement-based testing in HIS to assess that the execution behavior stays within the allotted bounds [19].

In mainstream domains, the use of PMCs is widespread: a score of software tools and libraries [1], [2] support performance monitoring and debugging on top of modern PMU for a wide spectrum of processor families. As a representative example, Performance Application Programming Interface (PAPI) [6] is a widely adopted, modular library supporting mainstream targets and operating systems that has surged as reference solution for application profiling.

[1]In this work we use the term multicore or multicore processor to refer to MPSoC or multiprocessor system on chip.

In HIS the availability of a consolidated, reusable performance monitoring interface is a fundamental enabler for timing and functional V&V. Standardized PMC libraries would allow to abstract away from low-level hardware and software configuration details, and to get rid of onerous, over-tailored, and error-prone custom solutions. However, libraries and tools from the mainstream domain have not been ported to HIS for a twofold reason. First, HIS have been traditionally perceived as a niche market, deploying specific combinations of hardware and real-time operating systems, which has discouraged any effort towards standardization and reuse. And second, performance monitoring solutions for HIS must meet specific requirements determined by the usage scenario (i.e., granularity and precision) and the inherent predictability and analyzability concerns in HIS (e.g., footprint and overhead).

Another challenge in HIS is that validation often occurs close to the end of the software development process where the system is (almost) fully integrated so as to capture the interactions among software components, e.g. multicore timing interference. The collected PMC measures, however, mix together the contribution of several software components that have been executing within the observation window, thus including several tasks and the operating system itself. A fundamental requirement is thus the ability to filter out the contribution of the different software elements.

In this paper, we define a baseline PMC Reading Library (PRL) that can be effectively deployed to collect reliable and consistent PMC readings while meeting the specific HIS requirements. We analyze the specific requirements of a monitoring solution for HIS and propose a reference implementation for a PMC configuration and reading library for HIS building on a subset of the PAPI library. This library has been tailored and extended to meet HIS requirements and to support the collection of observations at task-level, by filtering out the contribution of the underlying operating system. We instantiate PRL to a reference automotive configuration comprising a TriCore target (AURIX TC297) and an OSEK-compliant RTOS (ERIKA Enterprise v2). We asses PRL against the identified requirements and show its use to support the analysis of multicore timing interference on an illustrative scenario.

The rest of this work is structured as follows: Section II discusses the requirements for a performance monitoring solution in HIS. Section III presents our PRL reference implementation, which we validate and assess in Section IV against a representative non-functional analysis scenario. Section V introduces related works, and Section VI concludes the paper.

II. Performance Monitoring Library for HIS

In HIS novel approaches are sought to complement existing timing analysis solutions [12], [20] and counter the challenges posed by multicores. PMCs are increasingly used to gain insight on how applications interact as a central element to provide evidence of expected behaviors. This includes approaches that build on PMC to modeling the impact of contention on software time and approaches that use PMCs to set quotas to the accesses tasks can do to different resources [5], [14].

The number of hardware events that can be collected with PMCs is increasing in modern platforms, thus satisfying an growing need for detailed insight on the behavior of critical software applications. For instance, the Arm A53 cores implements 63 event counters, while more modern processors of the Arm A Cortex family, namely the A57 and A72, can track 92 and 85 events respectively. Also, PMC support is not homogeneous across hardware platforms, on the opposite, actual PMC support is often implementation dependent and may even vary within devices in the same family of processors.

This has motivated efficient abstractions capable of providing a standard, reusable set of PMC functionalities. In the the mainstream domain this is captured via a standardization effort that has led to the definition of debug interface standards [3], kernel-level utilities [1], and dedicated cross-platform performance monitoring API. In this line, PAPI is an cross-platform middleware library for performance monitoring that can be considered a de-facto standard for performance profiling in mainstream computing. Despite their exceptional level of diffusion, PAPI and other common tools have not been ported to reference platforms and RTOS in the HIS domain, nor equivalent solutions have been developed. The lack of standard solutions for HIS configurations is not solely explained by the only recent interest in PMC support but it also ascribable to non-overlapping requirements on PMC support. As a contribution to filling this gap, we present PRL, a performance monitoring library specifically designed to meet HIS requirements and provide the necessary support to PMC-based timing analysis approaches.

A. Requirements on PMC support for HIS

A performance monitoring solution for HIS comes with a specific set of requirements and constraints stemming from both the specific operational constraints of embedded critical systems (e.g. memory footprint) and the use of PMCs to support diverse analysis approaches from HW characterization to multicore timing interference analysis [5], [10], [19] (e.g. scope, precision, and intrusiveness).

Granularity and scope of profiling: the profiling information is typically required at the level of single functions or run-time entities (e.g., tasks or runnables, in the automotive domain) with finer-granularity only sought for specific functional requirements or optimization purposes. From a timing analysis perspective, we are interested in associating PMC readings to specific unit of executions, which may require specific solutions under various instantiations of RTOS/Hypervisor. As already observed, blindly collecting observations within a given interval of time (function or task body) has the effect of factoring in multiple elements that are difficult to tell apart, including the execution of the real-time kernel and of any other functions that may be executing according to the specific execution model (e.g., preemptions). A PRL solution shall necessary support the capability to filter out the activity of both the operating system and other concurrent run-time entities.

Multicore support: From a multicore perspective, the PRL should also support the gathering of profiling concurrently from different cores.

Low and predictable overhead: stringent requirements in HIS are set on timeliness, analyzability, and (time) predictability [17]. The software layer realizing the PMC cannot be jeopardizing analyzability and timing behavior of the application under profiling. For these reasons, the profiling solution is required to cause minimum and predictable (e.g., low variability) overhead on the analyzed application, regardless of whether the PRL functionalities are enabled or not. PRL implementation shall follow the most conservative coding standard and rules (e.g., [13]) to comply with domain-specific requirements and favor PRL predictability and analyzability.

Privileges: Despite the PMC configuration may require some degree of integration with the underlying operating system, it is fundamental that PRL is offered as a user-level API. It should be possible to exploit the PMC library regardless of the execution privileges. Of course this does not prevent from implementing more complex solutions as, for example, virtualizing the PMU at the Hypervisor level [11].

Reduced footprint: Some classes of HIS are constrained to a limited amount of resources and fine-tuned memory allocations. For these reasons, the PRL should be implemented as a lightweight software layer with minimal memory requirements in both code and data structures.

Operation: monitoring APIs in mainstream domain support an interactive mode. The latter, however, is irrelevant in critical embedded systems where the PRL interface should support fully static configuration of the PMCs. Additionally, the way to extract PMC results from the system should be configurable and adaptable to the available debug support.

III. PRL

PRL is a lightweight software layer that supports a minimal yet sufficient set of performance monitoring functionalities and is reusable across diverse hardware and software stacks. Also PRL aims at identifying a common set functionalities that are generally supported by PMUs in different processors, yet it can be extended to build on configuration specific features.

The design and implementation of PRL is largely inspired by PAPI [6]. PAPI offers a multi-layered, portable, and efficient API to monitor application performances exploiting the typical hardware performance counters found on most modern microprocessors. PAPI has been designed, and successively evolved, to address the performance profiling requirements on generic mainstream targets, including high-level average performance metrics such as instructions per cycles, misses per

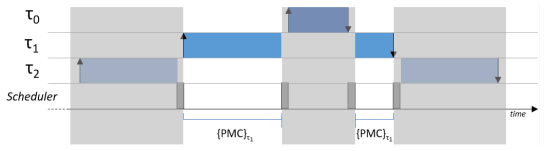


Fig. 1: Task-level event filtering.

cycles and alike. In the scope of HIS, the performance monitoring solutions is mainly intended to support timing analyses approaches and address more focused hardware events. At the same time, some of the metrics and functionalities supported by PAPI are not necessarily useful for HIS. PRL diverges from PAPI to avoid unnecessary complexity and to cover HIS specific profiling requirements.

The main focus on PRL is set on supporting timing analysis tasks on MPSoCs where PMCs are advocated as the main source of information for the characterization of multicore timing interference [10], [12], [14], [19]. This brings few practical requirements on PRL design and functionalities. The library shall be able to collect events and associate them to specific run time entities (task, runnable or software partition). The library shall not only provide access to focused counters (e.g., capturing the stalls suffered or caused by a task when accessing a shared hardware resource) but shall also be able to associate hardware events to each task. This is a particularly challenging when profiling is performed close to the end of the development process, where multiple tasks are deployed by an operating system or hypervisor on multiple cores. The library shall support *simultaneous multicore profiling* of *per task events*, thus being able to track the events in several cores at the same time, while distinguishing between tasks scopes and isolating the impact of the operating system. The last aspect, which applies at both intra-core and inter-core levels, is particularly critical to timing and response time analyses but typically disregarded in mainstream PMC libraries.

The relevance of event filtering is exemplified in Figure 1 on a relatively simple single-core scenario. Per-task event filtering allows to capture PMCs that are relative to τ_1 execution while plain PMC profiling would capture the effect of the execution of τ_0 (preempting τ_1) and the underlying scheduler, hence including all events in τ_1 execution window. The difference between the two is the same happening between task execution time and response time. The capability of filtering events directly, rather than an aftermath, is an essential feature that distinguishes PRL from PAPI. PRL is conceived to be easily integrated with the RTOS and other libraries, and allows associating and profiling event sets to concrete run-time entities.

PRL's design follows a three-layered structure, as illustrated in Figure 2. Two layers are exposed to the user as high-level and low-level PRL APIs. The high-level API provides the minimal set of library functions to profile the behavior of any piece of software. The low-level API instead provides support for higher configurability and control over the profiling activity. In fact, this layer is conformant with the low-level API in PAPI. The two layers are not mutually exclusive as the outer layer directly interacts with the inner layer. A third

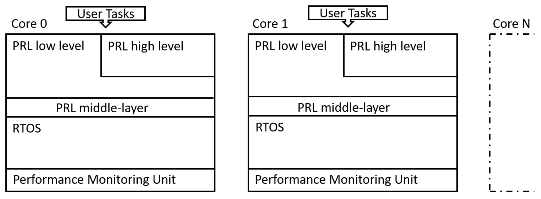


Fig. 2: PRL architecture description.

layer, finally, act as a middle-layer between the PRL and the underlying run-time. This layer is not visible to the user and is responsible of the integration with the RTOS.

PRL main concept is the *Event Set*, borrowed from PAPI, consisting in a user-defined group of hardware events that are gathered over the monitored execution. The user is responsible of specifying the set while PRL internally configures the PMU to properly configure the performance monitoring counters. The user interacts with PRL through a set of intuitive calls made from a program's body. PRL shares the same signature as the PAPI library. This signature-level equivalence allow users accustomed with PAPI to straightforwardly use PRL.

The PRL design specifically supports integration with the underlying run-time by exposing a middle-level layer, whose objective is to guarantee PRL is separately accounting for the events triggered/incurred by each task under analysis. Such feature is necessary in order to profile multiple tasks at the same time on the same and different cores. PRL supports the association of event sets and event counts to each entity or task in the run-time with reduced overhead and footprint data structures that allow to save the event set of a given task whenever the task is preempted by an interrupt or a higher priority task, and to restore it when the task resumes execution.

IV. Evaluation

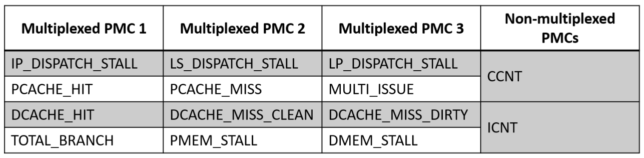
PRL has been instantiated on a hardware and software configuration representative of the automotive domain, one of the main market driver in HIS [4]. We assess the concrete implementation against the requirements identified in Section II-A. The evaluation strategy unfolds into two main sets of experiments. First, we use small, carefully-designed synthetic benchmarks to validate PRL with respect to correctness, accuracy, overhead, and scalability in a single-core scenarios. Second, we experiment on a multicore scenario to test PRL on a more computationally representative scenario.

Portability and adaptability to different target configuration inspired our PRL design as both are pivotal for the widespread adoption of a performance monitoring library the in HIS. Below we focus on the instantiation of PRL to one specific configuration. We deploy PRL to a TriCore TC297 (1st gen), which equips 3 cores and a crossbar that grants access to four 4MB independent Program Flashes (PFlash), one Data Flash (DFlash) and a Local Memory Unit (LMU) composed by a 32KB RAM. All cores are superscalar, with integer, load-store, and (zero overhead) loop pipelines. The target PMU comprises two fixed PMCs for cycles and executed instructions, complemented by four multiplexed PMCs to measure a set of events,

TABLE I: Absolute PMC values with and without PRL on TC297 and ERIKA v2.

	No RTOS		Erika v2								
	Direct PMU	PRL	Direct PMU			PRL					
	Isolation		Isolation	1 Preempt	1 Preempt*	Isolation	1 Preempt	1 Preempt*	2 Preempt	3 Preempt	4 Preempt
CCNT	400,000,004	400,000,045	400,047,005	400,064,890	404,061,862	400,005,313	400,005,303	400,005,297	400,005,291	400,005,307	400,005,332
ICNT	400,000,005	400,000,035	400,026,605	400,040,692	404,037,065	400,001,211	400,001,211	400,001,211	400,001,211	400,001,211	400,001,211
pcache_hits	200,000,001	200,000,006	200,015,397	200,019,808	201,018,781	200,002,045	200,002,037	200,001,937	200,002,021	200,002,041	200,002,065
pcache_miss	1	1	18	46	48	2	2	2	2	2	2
multi_issue	100,000,002	100,000,001	100,005,069	100,008,303	101,007,178	100,000,403	100,000,403	100,000,404	100,000,405	100,000,404	100,000,407
dcache_hits	0	0	400	728	732	0	0	0	0	0	0
dcache_miss	0	0	0	2	2	0	0	0	0	0	0
total_branches	100,000,000	100,000,002	100,006,869	100,010,463	101,009,574	100,000,404	100,000,403	100,000,405	100,000,404	100,000,405	100,000,407
total_mem_stalls	100,000,006	100,000,019	100,004,835	100,005,852	101,005,114	99,999,832	99,999,852		99,999,852	99,999,854	99,999,855
pmem_stalls	7	19	3,160	2,911	3,149	231	252	252	251	250	249
dmem_stalls	99,999,999	100,000,000	100,001,675	100,002,941	101,001,965	99,999,601	99,999,600	99,999,644	99,999,601	99,999,604	99,999,606
IP_stalls	199,999,997	200,000,000	200,001,138	200,004,002	202,002,053	199,999,799	199,999,800	199,999,821	199,999,800	199,999,799	199,999,797
LS_stalls	1	1	9,579	12,045	12,003	1,204	1,203	1,004	1,202	1,201	1,200
LP_stalls	0	0	620	613	678	1,182	1,200	752	1,196	1,183	1,171

as summarized in Figure 3. These include events related to the data and program cache (hits/misses) and different stall cycles. IP-, LS- and LP-DISPATCH_STALL are incremented every cycle in which the Integer, Load-Store and Loop dispatch units are stalled, respectively. This same logic applies to PMEM- and DMEM-STALL which are incremented on each cycle the fetch or Load-Store units are requesting instructions or data respectively and the memories are stalled.



Multiplexed PMC 1	Multiplexed PMC 2	Multiplexed PMC 3	Non-multiplexed PMCs
IP_DISPATCH_STALL	LS_DISPATCH_STALL	LP_DISPATCH_STALL	CCNT
PCACHE_HIT	PCACHE_MISS	MULTI_ISSUE	
DCACHE_HIT	DCACHE_MISS_CLEAN	DCACHE_MISS_DIRTY	ICNT
TOTAL_BRANCH	PMEM_STALL	DMEM_STALL	

Fig. 3: PMCs of the TC297 and TC397.

When is comes ot the RTOS layer, we addressed Erika Enterprise OSEK-compliant RTOS [8] v2, currently in production in various automotive, industrial and HVAC systems. It was developed with Hard Real-Time support for multi-core microcontrollers. ERIKA v2 presents a reduced flash footprint and requires one copy of the RTOS per core.

A. Validation experiments

We compare the PMC values gathered with and without PRL when executing the same application under different scenarios. As target application, we considered synthetic benchmarks that trigger a predefined set of hardware events. In this work we report the results for a store data-intensive benchmark in which most of the instructions are stores that miss in the Data cache. While we experimented with several other benchmark variants, they provide no additional insights over the selected benchmark and are hereby omitted. Experimental results are reported in Tables I where PMC readings for AURIX PMU supported events are reported for each scenario.

1) Bare metal execution: The baseline scenario for assessing PRL consists in a bare metal, single core setup where the target application executes uninterrupted, i.e. without preemptions. We compare the PMC values obtained over the benchmark execution by directly configuring the AURIX PMU with low-level instructions at the beginning at the end of the task against those observed by exploiting PRL. This setup matches the *No RTOS* columns in Table I. By comparing the first and second columns we see that the overhead in all PMCs incurred because of the library (PRL column) is in all cases negligible, and PRL can be deemed as accurate as the PMU.

2) RTOS impact: We analyze the case when the benchmark under analysis is the only task running in a single core setup. Table I reports the PMCs observed in ISOLATION for both DIRECT PMU (column 3) and PRL (column 6). For ERIKA v2, the overhead from reading PMCs on top of the RTOS is larger than in the bare metal scenario. The overhead in terms of timing and events is stemming from the impact of the periodic interrupts. Nevertheless, PRL always incurs less overhead than Direct PMU, thanks to its capability to filter out the contribution of other run-time entities to the collected events (including the RTOS itself).

3) Preemption impact: With the next set of scenarios we evaluate the impact of preemptions on the performance profiling and the robustness of PRL. In general, several tasks are co-scheduled in the same core and depending on the scheduling algorithm they can be preempted. We assess the capability of PRL to isolate events from other tasks than the one under analysis and the incurred overheads on every context switch to save and restore counter values. To this purpose, we deployed two reduced version of the analysis benchmark as preempting tasks (preempt and preempt*). Results are reported in Table I for both DIRECT PMU and PRL whit different numbers of preemptions. We see that the Direct PMU solution cannot filter out the contribution of the preempting task and the PMCs values are including the execution of the preempted tasks, and the RTOS impact. In the PRL setups, instead, we observe that the impact of preemptions is essentially filtered out and masked by small overhead from kernel interrupts. This is because context switches happen within the kernel periodic interrupts and PRL guarantees isolation between task events.

4) Multicore scenario: In this case we run a workload comprising different applications, deploying kernels used in many applications in critical systems [18]. We experimented with increasing number of contenders targeting the LMU SRAM. We track the data memory and load-store dispatch unit stalls. To increase the impact on the data memory, the data caches are disabled. Each copy of the RTOS kernel is stored in the core-local program flash. An instance of PRL is loaded into each core-local program-scratchpad to reduce the overhead in the system and avoid additional event counts and cache pollution. The data structures accesses by the target programs are mapped to the 32KB LMU SRAM. Table II reports PMCs collected when increasing the number of benchmark instances running in parallel, with Exp1 matching the isolation scenario.

TABLE II: Multicore scenario on TC297 and Erika v2.

Exp	Core ID	ccnt	icnt	total_br	pmem_st	dmem_st	ip_st	ld_st
1	Core 0	47409	14089	1282	6	37155	0	25
2	Core 0	56629	14089	1282	6	46375	0	28
	Core 1	58706	14089	1282	6	44356	0	29
3	Core 0	71091	14089	1282	6	60837	0	32
	Core 1	65118	14089	1282	6	54864	0	30
	Core 2	66764	14089	1282	6	56510	0	31

These experiments demonstrate that PRL supports profiling in parallel on multiple cores and events that are not meant to depend on multicore execution (icnt, total branches) stays the same across all scenarios. We observe a notable increase in execution time (up to 40% with 3 corunners) which is correlated to the increase in stalls incurred while accessing the memory interface in the crossbar (dmem stalls). Pmem stalls in this case are unaffected as code, including the RTOS is on a reserved flash.

V. Related Works

PMCs has been increasingly considered in the scope of HIS, to feed and support timing analysis tasks [5], [10], [12], [14], [19], an inescapable concern for time critical systems. From the timing analysis perspective, PMCs has been mainly considered to support the analysis of multicore timing interference when accessing shared hardware resources [5], [10], [19] and to implement regulation mechanism to remove or control the impact such interference [14]. Both families of approaches rely on accurate, low-overhead information from the PMCs, which can often vary across platforms and software configurations. A standardized low-level performance monitoring library is beneficial to both types of approach, by providing access to the PMC functionalities without requiring the specific solution to deal with platform-specific issues.

In the mainstream domain, the Perf tools [1], perfmon [2], and PAPI [7] tools have quickly reached an exceptional level of diffusion as they offer a generic abstraction layer to configure and collect information from performance monitors. However, the abstractions and functionalities offered by those tools often result to be too generic and coarse-grained for the intended use of PMCs in the HIS domain. In this respect, PRL offers a lower-level abstraction (very close to the actual PMC layer) and better support for reliable, fine-grained profiling.

In the HIS domain, the use of PMCs for timing characterization is addressed in [12], where also PMC validation is considered to avoid potential inaccuracies. While the use of PMC information is crucial in several timing analysis approaches, no generic performance profiling library has been proposed. In [9] a preliminary porting of a subset of PAPI functions to a relatively simple bare-metal setting while, in this work, we analyze and implement RTOS level functions and support capturing per-task events.

VI. Conclusions

A standardized performance reading library is increasingly needed to support the use of PMCs for the validation of high-integrity systems. In this work, we present PRL to contribute to a reference performance reading library design capturing the specific requirements of high-integrity systems. We assessed our proposal a representative automotive configuration with the Infineon AURIX TriCore TC297 and the OSEK-compliant ERIKA Enterprise RTOS v2, against the high-integrity systems requirements and demonstrate its utility to support the analysis of multicore timing interference. As a future work, we plan to port PRL to different configurations and to extend the support for low-level PMCs, such those provided by the Multi Core Debug Solution (MCDS) layer in AURIX.

Acknowledgments

This work has been supported by the Spanish Ministry of Science and Innovation under grant PID2019-107255GBC21/ AEI/10.13039/501100011033, the European Unions Horizon 2020 Framework Programme under grant agreement No. 878752 (MASTECS), and the European Research Council (ERC) grant agreement No. 772773 (SuPerCom).

References

[1] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.
[2] perfmon2: Improving performance monitoring on Linux. http://perfmon2.sourceforge.net/.
[3] Nexus 5001. IEEE-ISTO 5001™-2012, The Nexus 5001™ Forum Standard for a Global Embedded Processor Debug Interface. https://bit.ly/2MIoJY1.
[4] Deloitte. *Semiconductors – the Next Wave Opportunities and winning strategies for semiconductor companies*, 2019.
[5] E. Díaz et al. Modelling multicore contention on the AURIX^{TM} TC27x. In *DAC*, 2018.
[6] K. London et al. The papi cross-platform interface to hardware performance counters. *Dept of Defense Users' Group Conference*, 2001.
[7] S. Browne et al. Papi: A portable interface to hardware performance counters. 1999.
[8] Evidence. *ERIKA Enterprise Manual V1.4.5*, 2012.
[9] J. Giesen et al. ePAPI: Performance Application Programming Interface for Embedded Platforms. In *WCET Workshop*, 2019.
[10] R. Inam et al. Bandwidth measurement using performance counters for predictable multicore software. In *IEEE ETFA*, 2012.
[11] P. Liu et al. Sysoptic: A fine-grained monitoring system for virtual machines based on pmu. In *SOSE*, 2019.
[12] E. Mezzetti et al. High-Integrity Performance Monitoring Units in Automotive Chips for Reliable Timing V&V. *IEEE Micro*, 2018.
[13] MISRA. *Guidelines for the Use of the C Language in Critical Systems*. 2013.
[14] J. Nowotsch et al. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *ECRTS*, 2014.
[15] R. Pujol et al. Empirical Evidence for MPSoCs in Critical Systems: The Case of NXP's T2080 Cache Coherence. In *DATE*, 2021.
[16] N. Sensfelder et al. On How to Identify Cache Coherence: Case of the NXP QorIQ T4240. In *ECRTS*, 2020.
[17] J. Stankovic and R. Krithi. What is predictability for real-time systems? *Real-Time Syst 2, 247–254*, 1990.
[18] H. Tabani et al. A cross-layer review of deep learning frameworks to ease their optimization and reuse. In *ISORC*, 2020.
[19] S. H. VanderLeest and S. R. Thompson. Measuring the impact of interference channels on multicore avionics. In *DASC*, 2020.
[20] R. Wilhelm and J. Reineke. Embedded systems: Many cores - many problems. In *SIES*, 2012.