

TAP: Task-Aware Profiling on Integrated Systems

Jeremy Giesen

*Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
Barcelona, Spain
jeremy.giesen@bsc.es*

Enrico Mezzetti

*Barcelona Supercomputing Center
Barcelona, Spain
enrico.mezzetti@bsc.es*

Jaume Abella

*Barcelona Supercomputing Center
Barcelona, Spain
jaume.abella@bsc.es*

Francisco J. Cazorla

*Barcelona Supercomputing Center
Barcelona, Spain
francisco.cazorla@bsc.es*

Abstract—Hardware Performance Monitors (HPM) are increasingly exploited for timing verification and validation of time-critical embedded systems (TECS). HPMs are typically collected at the lowest software level, which makes it difficult to unequivocally account events to specific run-time entities, a prerequisite for any form of analysis, without relying on ad-hoc support from the run-time or operating system layer. The latter, however, is either unavailable or not fully adequate for verification requirements. Moreover, timing-related concerns in the analysis of embedded systems are typically addressed in the final stages of the software development process where multiple tasks are fully or partially integrated on the platform and it is therefore hard, if not impossible, to enforce controlled testing scenarios where contributions to event counts can be dissected. In this work, we present TAP a generic concept for allowing Task-Aware Profiling of individual tasks in an already integrated system on MPSoCs with on-core and off-core HPM support. The proposed approach combines a lightweight user-level configurable API and minimally intrusive extensions to the operating system layer to enforce separation of contexts when collecting HPM. We implement and assess TAP on top of an Infineon AURIX MPSoC and the OSEK-compliant ERIKA Enterprise RTOS, offering a consistent and intuitive interface for governing and filtering the different sources of events. Our results on synthetic and automotive benchmarks show that TAP can transparently gather and filter the events of interest while incurring negligible overheads.

I. INTRODUCTION

The design and development of safety-critical embedded real-time systems are required to meet stringent verification and validation constraints, as defined by domain-specific functional safety standards and regulatory documents [1], [2]. In recent years, timing analysis techniques have been increasingly using low-level hardware performance monitors (HPM) as a source of complementary information to cope with the increasing complexity of hardware and software solutions. HPMs

have proven to be useful for multicore timing interference analysis and contention control [3]–[10].

HPM-based approaches assume the capability of correctly ascribing events to a specific unit of analysis to guarantee the observations collected over the test campaign are indeed relative to the intended context of execution and do not embed noise from external actors. However, such capability is not directly supported by the hardware monitoring unit, which operate at the lowest level of execution and cannot distinguish between software abstractions. This implies that all events happening in an observation window are by default ascribed to the task under analysis (TuA), while they can be triggered by the operating system or any other application concurrently executing in the same time window. At the same time, it is not always possible to enforce artificially controlled scenarios (e.g. based on stubbing) where the application under test is the only piece of software executing on the target platform and therefore the sole contributor to events. As a matter of fact, timing verification and validation are often postponed to the final stages of the development process where all (or most) components are available and analysis are performed on a fully or partially integrated systems.

Low-level profiling tools integrated with standard operating systems provide partial support for context-aware profiling of hardware events on mainstream hardware and software platforms. This is the case, for example, of perf [11] on standard linux-based operating systems, which allows to track events per process. However, those solutions are generally not available on embedded system platforms (e.g. Infineon AURIX processors) and real-time operating systems (e.g. ERIKA) and do not always provide the necessary granularity or accuracy to support timing V&V objectives. In particular, those solutions typically only support profiling contexts for core-level events and do not track system-level (global) events which, however, are increasingly supported in embedded system targets and are extremely fundamental to support multicore timing analysis.

Existing real-time analyses in embedded domains focus on early-stage testing scenarios where software units under analysis run in isolation [6], [12], [13] or HPM are cumulatively

This work has been supported by project PLEC2023-010240 funded by MICIU/AEI/10.13039/501100011033 and the European Union's Horizon Europe research and innovation programme under grant agreement No. 101070238. The authors also thank the support of the Department of Research and Universities of the Government of Catalonia to the Smart and Safe Autonomous Systems (SSAS) Research Group, (Code: 2021 SGR 00637).

collected for all events in a core [3], [7]. Instead, in this work we address the need for extensive context-aware profiling of hardware events in (either fully or partially) integrated embedded critical systems. We identify and develop on the need for profiling both core-level and system-level HPMs on a per-task basis during late integration stages and identify the design principles for a generic approach for enforcing separation of contexts between HPMs making it possible to associate both core and system-level events to the execution of specific software tasks or *runnables in automotive software*. We define TAP, a task-aware profiling solution that meets timing verification objectives by providing a lightweight and configurable user-level API, with minimal changes to RTOS layer, transparently combining both on-core and off-core events and enabling HPM profiling on fully or partially integrated systems. TAP allows collecting information on the execution of (multiple regions within a) specific task, filtering out those events that are not directly ascribable to the task execution (e.g. RTOS' interrupts and other tasks in the same or other computing node). We assess TAP in a representative automotive setup on top of OSEK-compliant ERIKA Enterprise RTOS [14] and with the PMU on-core and off-core support provided by the Infineon AURIX TC397 [15], a reference platform in the automotive domain. Our results confirm that the proposed solution allows separation of HPM contexts while incurring negligible overheads.

The remainder of this paper is organized as follows: Section II introduces TAP, its motivation, assumptions, and related challenges. Sections III and IV discuss the implementation and assessment of TAP on an automotive setup. Section VI presents the main conclusions of this work.

II. HPMs FOR TASK-AWARE PROFILING

The number and types of HPMs supported is constantly growing [16]–[18], partially due to the increased hardware complexity, but also in response to emerging requirements on the use of events for analysis purposes, beyond simple debugging and performance profiling [5], [7], [19], [20]. HPMs can be classified per *type* and *scope*. The type relates to the monitor semantics, either counting specific events (e.g., cache misses) or the number of cycles when a condition holds (e.g., stall cycles). The scope, instead, is relevant when an event can arise in multiple hardware components and identifies the elements that contribute to the associated PMCs (e.g., core-level memory access versus global ones).

Manufacturers have swiftly moved from supporting a handful of HPM to deploying specialized Performance Monitoring Units (PMU) with enhanced support for tracking and filtering HPMs, like in the Infineon MultiCore Debug Solution (MCDS) [16], the NXP Event Processing Unit [17], further complemented by specialized monitoring support like in the ARM Memory System Resource Partitioning and Monitoring [21]. PMUs interact with industrial debug support tools [22] for performance monitoring, testing, and program tracing. Limitations are placed on the number and types of events that can be tracked at the same time, sometimes requiring engineers

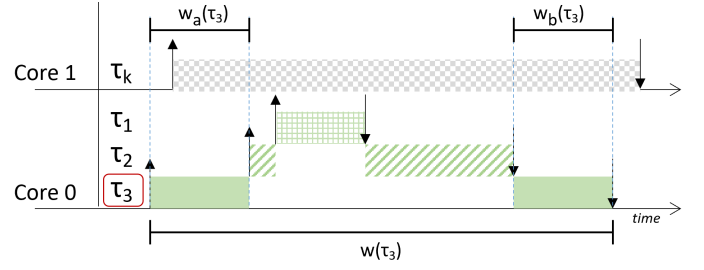


Fig. 1: Illustrative HPM collection scenario.

to master the complex allocation of (few) observation points in snooping units [16], especially for off-core monitors.

State-of-the-art approaches leveraging HPMs for timing verification and timing interference control (e.g., memory bandwidth regulation) build on the often implicit assumption that HPMs can be unequivocally associated to the target unit of analysis [3], [5], [7]. However, such assumption is not generally met on COTS PMUs, which normally operate at the hardware level where no concept of software abstraction exists. Therefore, whenever multiple elements can be executing concurrently or in parallel with the run-time entity or task under analysis, they can all potentially contribute to tracked HPMs. Context-aware monitoring is partially supported by specialized tools, like *perf* [11], on general purpose Linux-based systems and is not generally supported in specialized hardware and software setups as those deployed in embedded critical systems, the latter having specific requirements in terms of accuracy, operation, and overheads. However, process-aware profiling context in mainstream and ad-hoc solutions, such as the AURIX development studio or MCDS trace viewer from Infineon, guarantees accountability of on core-level events and collection of event traces, but fails to support any sort of context for system-level events, which make it fall short for relevant timing-verification concerns.

To illustrate these limitations, we consider the partitioned multicore scenario in Figure 1, where we schedule tasks τ_1 , τ_2 , τ_3 on Core 0, and another task τ_k on Core 1. Further assume we are interested in collecting HPMs for τ_3 , during its execution window w_3 ; w_i normally corresponds to a single τ_i activation or a function invocation in τ_i . Also, τ_3 happens to be the lowest priority task in Core 0 and suffers inter-task interference from tasks τ_1 and τ_2 . For core-level events (related to Core 0 activity) the selected HPM will be incremented during the whole observation window w_3 therefore possibly including the contributions of τ_2 and τ_1 if no process-aware profiling context is supported. Instead, we are interested in tracking increments to HPMs during sub-windows w_{3a} and w_{3b} . For system-level events (related, for example, to requests sent over the interconnect) existing solutions do not consider the execution context and selected HPMs will include the activity of all tasks in Core 0 and τ_k in Core 1 during w_3 .

The HPMs collected within τ_3 execution boundaries are useful to extract some time-related metrics for τ_3 as, for example, maximum observed response time [23] and maximum reaction time [24], but will not allow deriving reliable metrics in the scope of the direct contribution of τ_3 to the HPMs. These

TABLE I: PMU multiplexed configuration. PC and DC stand for program and data cache, respectively.

CCTRL	MICNT	M2CNT	M3CNT
000	PMU_IP_DI_STALL	PMU_LS_DI_STALL	PMU_LP_DI_STALL
001	PMU_PC_HIT	PMU_PC_MISS	PMU_MULTI_ISSUE
010	PMU_DC_HIT	PMU_DC_MISS_CLEAN	PMU_DC_MISS_DIRTY
011	PMU_TOTAL_BR	PMU_PMEM_STALL	PMU_DMEN_STALL

cache and the last row gives information on cycles in branch-resolution stages and (any) program and (any) data memories stalls. Two extra HPMs are exclusively reserved for cycles and instructions. TAP configures the multiplexed counters to monitor user-specified events.

MCDS: AURIX TC397 includes a MultiCore Debug Solution (MCDS) peripheral in debug versions that provides 3 major sources of hardware event information.

a) *Core-level HPM:* MCDS can be configured to track on-core events, like counters related to local-cache accesses, including hit/miss counts to caches, and access counts to DSPR/PSPRs (from local CPU). Some of these counters can already be gathered via the PMU.

b) *System-level HPMs:* tracking data/code accesses to PFlashes, DLMUs, LMUs and DSPR/PSPRs when accesses are managed by the crossbars. They provide either an aggregated count of all accesses of all masters accessing the device, or the number of accesses executed by a particular master.

c) *Configurable HPMs:* allow tracking data accesses (load/stores) made within a specified range, purposely matching in our implementation the address space of the different memories (slave interfaces). The MCDS allows defining two ranges per observation block (later described in this section), purposely pointing to the cacheable and non-cacheable addresses to allow counting the activations of the trigger signals when accessed either in cacheable or non-cacheable mode. The MCDS uses two types of observation blocks to track the configurable HPMs.

- Processor Observation Block(s) (POBs) operating in:
 - Master mode: These triggers become active when an access occurs from a particular CPU, to a particular memory range, via the processor data lane (Data Memory Interface, DMI). Code accesses are not observable using this mechanism.
 - Slave mode: These triggers become active when an access occurs to PSPR/DSPR/DLMU surrounding a CPU. Initiators of the accesses may be any on-chip master (CPU, DMA..), except the local CPU (the CPU whose surrounding memories are being observed).
- Bus Observation Block (BOB): These triggers become active upon accesses to memories that are not part of the island of any CPU.

Within each POB/BOB, activation of the trigger can be configured to occur depending on memory range, read or write transaction, and bus master initiating the transaction.

TC397 includes 3 POBs, POBx/POBy/POBz. Each is configurable to a single CPUx, in either POB-master or POB-slave mode. The configuration to each CPUx have the following

conditions: POBx: Attachable to cores 0 to 5. POBy: Attachable to core 1 to 5. And POBz: Attachable only to core 0. On the other hand, 1 BOB allows monitoring of accesses on SRI memories (SRI BOB). MCDS has also a separated BOB for SPB (SPB BOB targeting SPB activity). Manipulating the MCDS either for configuring the System-level HPMs, configurable HPMs, as well as start, stopping to track and output the counts produces a small yet known overhead on the response time of the task(s) running on the setup/system.

B. ERIKA Enterprise OSEK compliant RTOS

ERIKA Enterprise [14] RTOS is specifically tailored for embedded automotive systems by ensuring adherence to the OSEK standard. ERIKA is an open-source project with GPLv2 licensing mode, thus enabling the integration of TAP.

ERIKA preemptively schedules tasks based on priority levels, ensuring real-time responsive behaviour for both timing and event triggered task activations. The RTOS incorporates resource management mechanisms to handle CPU time, memory, and I/O devices effectively, as well as various inter-task communication mechanisms, compliant with the OSEK standard, to facilitate synchronization among tasks. The OSEK Implementation Language (OIL) specification is supported, facilitating the deployment of ad-hoc testing scenarios.

We target ERIKA v.3, which follows a symmetric multiprocessing model where a single instance of the RTOS is shared among all cores and the scheduler works on per-core queues. Queues and intra-core interrupts, including RTOS tick, are fully managed locally on each core. Beside these core-local elements, ERIKA kernel code base is normally mapped to pF_0 , in reason of its size and of the role of CPU0 in initializing the RTOS and enabling the other CPUs in the system.

C. TAP Implementation

Following the principles in Section II-A, and aiming at guaranteeing context-aware profile management for both core and system level HPMs, the proposed TAP implementation focuses on facilitating detailed analyses of key metrics for embedded critical systems such as execution time, stall cycles, resource requests, and many more, while minimizing overhead.

In terms of footprint, the proposed implementation builds on lightweight data structures (the profiling context requires just 176 Bytes of data) and marginally impact on the RTOS code base size. TAP code and data are mapped to core-local scratchpads, to reduce overhead and memory interference.

Profiling context configuration: The pivotal concept in TAP is the notion of *profiling context*, extending the task context with a set of HPMs associated to the specific task (or regions thereof). In order to implement the sought separation among the profiling of run-time entities, it is necessary to guarantee the profiling context is initialized, saved, and restored by the RTOS in a similar way as execution contexts. TAP defines a middleware layer on top of the RTOS that is engaged at specific point in the transition among task states. The TAP layer is involved at each context switch and manages the profiling context in accordance with the scheduler decisions.

TABLE II: MCDS HPMs against monitoring constraints.

Group	Dev	CPU	HPM
Local RAM & Scratchpad	PSPR_X	All	MCDS_[Dev]_SRI_RDIN
	DSPR_X DLMU_X	1 CPU	MCDS_[CPU]_[Dev]_RDIN
PFlash	PFlash_X	All	MCDS_[Dev]_SRILOCAL_RDIN
		1 CPU	MCDS_[CPU]_[Dev]_SRILOCAL_RDIN
LMU0	LMU0	All	MCDS_LMU0_SRI_RDIN MCDS_LMU0_SRI_WR
		1 CPU	MCDS_[CPU]_LMU0_SRI_RDIN MCDS_[CPU]_LMU0_SRI_WR
Core	CPU_X	1 CPU	MCDS_[CPU]_INST MCDS_[CPU]_IRA MCDS_[CPU]_STALL MCDS_[CPU]_BRANCH_HIT MCDS_[CPU]_BRANCH_MISS MCDS_[CPU]_ICACHE_HITS MCDS_[CPU]_ICACHE_MISSES MCDS_[CPU]_DCACHE_HITS MCDS_[CPU]_DCACHE_MISSES MCDS_[CPU]_PSPR_HIT MCDS_[CPU]_DSPR_HIT
Address range (SRI)	-	All	MCDS_[Range]_SRI_RDIN MCDS_[Range]_SRI_WR
Address range (DMI)	PSPR_X DSPR_X DLMU_X	All	MCDS_[Range]_DMI_RD MCDS_[Range]_DMI_WR

The profiling context is implemented as a lightweight data structure holding IDs of HPMs to monitor and a pointer ensuring the mapping between the data structure and RTOS Task Control Block (TCB). Additionally, the same data structure is used to keep the monitoring status (whether monitoring is on or off) as well as placeholders for storing the status of the selected HPMs. The data structure must be properly initialized before exploiting TAP profiling context separation. In line with real-time systems practice, the context initialization occurs at system initialization following a static system configuration.

The TAP library masters the inherent complexity of the monitoring support, providing a simplified and consistent abstraction to the user. When defining the profiling context, with the HPMs to be monitored, it's crucial to consider the inherent limitations of the hardware counters. These limitations encompass restrictions related to multiplexing, quantity, and exclusions. Core-local HPM, identified by the prefix `PMU_`, share the same counter as shown in table I and cannot be simultaneously active. Similarly, the MCDS offers a limited number of observation points to monitor the desired region. TAP intercepts incompatibilities that may arise from profile context definitions and notifies the user.

Table II provides a reasoned reorganization of MCDS HPMs into families or groups (parametric on given device or memory range) to make MCDS inherent limitations explicit.

Local RAM & Scratchpads: HPMs can be attached to one device type among PSPR, DSPR, DLMU. Once selected, all the devices of that type can be tracked either when triggered by a specific core or by all cores.

PFlash: HPMs tracking instruction fetches from a specific PFlash, when triggered by a specific core or by all cores.

LMU0: HPMs tracking instruction fetches or data read/writes issued to LMU0, when triggered by a core or by all cores.

Core: HPMs tracking core-local HPMs for a specific CPU. They partially overlap with local PMU HPMs.

Address range (SRI): HPMs tracking instruction fetches or

write operation within a configurable address range through the crossbar, without distinguishing the CPU generating them.
Address range (DMI): HPMs tracking (code and data) load or write operation within a configurable address range through a local port (not through crossbar). Therefore, defined ranges must map to local ports.

The last two groups have been wrapped into a predefined set of custom HPM in TAP matching CPUs and address ranges of the memory devices, for the sake of providing a consistent, simplified abstraction to the user.

TAP workflow: First, the profiling contexts of the target task must be initialized. Multiple monitoring regions in a task can be associated to the same profiling context, thus enabling the monitoring at finer granularity (e.g. runnable level). Context initialization is performed in the task preamble where the HPMs to be profiled are identified and used to properly configure the monitoring devices. Especially with the MCDS, this step implies following an allocation logic for the underlying monitoring resources. Once the profiling context has been properly configured, it is possible to specify the monitoring scope (full tasks or specific regions).

```

1 TASK(taskUnderAnalysis)
2 {
3     /*Beginning of the monitoring region*/
4     TAP_start_counters()
5     ...
6     /* Functional code */
7     ...
8     /*End of the monitoring region*/
9     TAP_stop_counters()
10    ...
11    TerminateTask()
12 }

```

Listing 1: TAP instrumentation in task body.

Listing 1 provides an example of full-task monitoring region. Lines 4 and 9 show the beginning and end of the monitoring region respectively, wrapping the functional code. The call to `TAP_stop_counters()` eventually freezes the TAP context and outputs the obtained HPMs values to the default output channel (MCDS buffer in this case).

Profiling context management in ERIKA: Understanding the task states and scheduler primitives is key for TAP to effectively manage the preemption of monitored regions to ensure the separation of profiling contexts. While a region is under active monitoring, hardware registers count the occurrences of the specified HPMs. However, upon task interruption, counters must be disabled the soonest possible to avoid monitoring HPMs related to RTOS operations. Whenever a preemption occurs, we may need to save current HPM values, and reconfigure the hardware to support a potentially new set of HPMs. These operations can be delayed (lazy management) until the newly selected task is being monitored.

The main modifications made to ERIKA to support profiling context management lay within two regions in the Interrupt Service Routine (ISR) and in the scheduler logic, as illustrated in Figure 3. First, we modified ERIKA ISR to disable monitoring as soon as an interrupt is taken (`TAP_disable_monitoring()`). This is

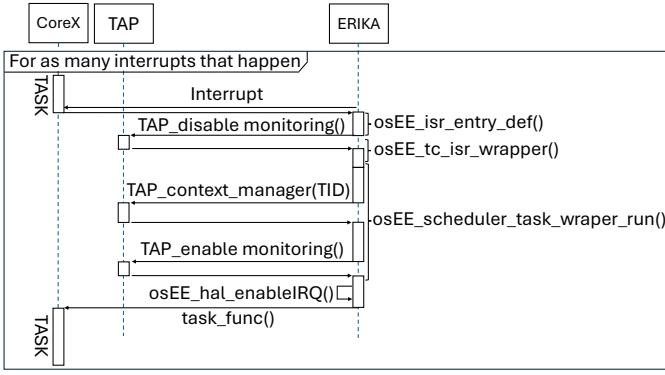


Fig. 3: Activity diagram for profiling context management.

done in the very first function invoked upon an interrupt, `osEE_isr_entry_def()`. In this function, the microcontroller stores its context by storing the status of a set of its registers before entering into the context management in `osEE_tc_isr_wrapper()`, which is responsible for activating the interrupt in the RTOS.

The second modification has been applied to intercept and react to scheduler decisions and is conveniently located at the end of the scheduling process in `osEE_scheduler_task_wrapper_run()`. In this function the RTOS updates the priority of the task to be resumed, re-enables the interrupts and resumes the task. The RTOS shares the new Task ID with TAP via `TAP_context_manager(TID)`, allowing TAP to take care of the profiling context management accordingly. Profiling context management occurs only when preemption occurs and may include: (i) storing of HPM values for the preempted task, and re-configuring PMU and MCDS for the newly selected task. TAP follows a lazy approach where many operations, including store and re-configurations are not done until strictly necessary (e.g., no need to re-configure if the new task is also the last one being monitored). After updating the TAP context, the interrupt is fully serviced, allowing ERIKA to re-enable interrupts via `osEE_hal_enableIRQ()`. Then, TAP can re-enable monitoring (if the selected task is under observation) just before the RTOS return control to the selected task (`task_func()`).

It is worth observing that while co-runner tasks on the same or other cores do not directly contribute to the HPMs of the monitored task, interference may still occur on, for example, cache statistics and stall cycles. In fact, for the AURIX TC397, this allows to use specific MCDS counters to track HPMs from other cores potentially injecting multicore timing interference.

D. Generalization

Porting TAP to the other microcontrollers within the automotive-predominant AURIX family is straightforward. On the other hand, extending TAP to other architectures or RTOS would require modifications to the interaction with HPMs and the RTOS management routines. Still, the lightweight interaction between the proposed library and the RTOS facilitates seamless porting to different RTOSes. Some key

considerations must be addressed to maintain efficiency and minimizing overhead:

- 1) RTOS interrupt and context management analysis. To minimize measurement noise, optimal places in the RTOS context management routines must be identified for timely disabling HPM tracking. Typically, this involves pinpointing the earliest feasible site, such as the ISR, and identifying the final control transfer point before resuming observation.
- 2) Handling of monitoring hardware. The library must efficiently manipulate the counters comprising the monitoring hardware. This may require platform-specific adaptations to ensure optimal performance. Also, the specific PMU-level logic and constraints must be understood and used to perform consistency check over profiling contexts.
- 3) Memory footprint optimization. To maintain a lean memory footprint, the library must be tailored to the characteristics of the monitoring hardware. This includes, optimizing reserved placeholders used for storing counter statuses during operation.

IV. EVALUATION

We provide a comprehensive assessment of TAP on a set of synthetic scenarios and representative kernels covering the main axis: (i) TAP effectiveness in tracking the activities of observed tasks only, even when they run with other co-runner tasks on the same and other cores; (ii) TAP induced overhead on RTOS operation and code footprint to manage all profiling related functionalities. We follow a scenario-based testing for validating the separation of profiling contexts and put TAP in action for contention analysis in Section IV-A. TAP overheads are analyzed for the exhaustive set of RTOS activity scenarios in Section IV-B. Without loss of generality, evaluations focus on full-task monitoring regions.

A. Separation of profiling contexts with TAP

We assess the effectiveness of TAP in filtering out the contributions of other tasks and the RTOS from the profiling context of a tasks under observation (UO). The same concepts can apply to code regions instead of full task bodies. We build TAP validation on two sets of core-level and system-level relevant Test Scenarios (TS).

Core-level Test Scenarios: Scenarios TS0-TS4 capture TAP behavior with respect to core level events and are therefore focusing on single core setups. For all scenarios we deployed multiple copies of same synthetic memory-intensive task. The synthetic task issues a total of 100,090,000 data accesses to memory from which 100,045,000 hit in the data cache, with just 45K clean and dirty misses. In preemptive scenario, we set up the scheduling table in ERIKA to generate exactly 4 preemptions on the task UO for each activation.

TS0 Non-preemptive, without filtering. The task UO runs solo without preemption and no contenders. The effect of RTOS interrupts, however, is still not filtered out.

TABLE III: TAP core-level profiling context.

	PMU_ICNT	PMU_CCNT	PMU_DC_HITS	PMU_DC_MISS
TS0	401,364,434	400,363,077	100,060,323	44,993
TS1	401,165,365	400,231,510	100,045,427	44,981
TS2	2,006,842,481	2,001,826,451	500,302,390	225,026
TS3	401,148,180	400,231,559	100,045,425	44,986
TS3bis	401,147,379	400,231,479	100,045,423	44,984

TS1 Non-preemptive, with filtering. As TS0 with the difference that we configure TAP filtering to remove the impact of the RTOS interrupts from the task UO profile.

TS2 Preemptive, without filtering. The task UO suffers four preemptions from another task running in the same core that is nUO. Both preempted and preempting tasks execute the same synthetic code. Without TAP filtering in place, the activity generated by the preempting task contributes to the HPM values of the preempted task UO. All this in addition to the impact of RTOS interrupts, similarly to TS0.

TS3 Preemptive, with filtering. Scenario similar to TS2 but, in this case, we configure TAP to filter out the impact of the preempting tasks and the RTOS interrupts.

System-level Test Scenarios: Scenarios TS4-TS6 capture TAP behavior with respect to system-level events and are therefore focusing on multicore setups, resembling fully or partially integrated systems, where separation of profiling context is an inescapable concern. In that setting, TAP support for context-aware profiling outmatches that of other tools.

TS4 Separate events. The task UO runs preemptively on Core 0 and sends requests to LMU0. Other tasks on the same core and on Core 1 perform accesses to a different memory device. TAP is filtering out the impact of RTOS interrupts and preemptions from the task UO profile.

TS5 Shared events, no system-level context. Same as TS4 except for tasks in Core 1 access the same memory device as the task UO in Core 0. TAP is filtering to the impact of the RTOS interrupts and preemptions from the task UO profile but is not preserving the profiling context with respect to system-level events. This setup corresponds to support provided in tools like `perf` or `PAPI`.

TS6 Shared events, system-level context. Same setup as TS5 but we enable TAP filtering to enforce profiling context segregation also for system-level events.

1) Core-level events: Scenario-based test results for core-level events are summarized in Table III. For each scenario, we report the following HPMs for the tasks UO: instruction count (ICNT), the cycle count (CCNT), and Data Cache statistics (PMU). For the latter we have reference expected values since the synthetic task triggers 100,090,000 accesses to memory from which only 45k are meant to miss the data cache. For each test scenario we report the mean value over 100 runs.

Scenario **TS0** sets a baseline result since the task UO is the only task running in the system. However, while no other task can contribute to the task profile, the RTOS is still generating periodic interrupts, whose impact is not filtered out. The RTOS impact can be appreciated comparing **TS0** with **TS1** where RTOS impact is filtered out, and we observe a decrease in

both cycles and instructions, while data cache statistics are not affected. This illustrates that the filtering capabilities of TAP are working. In scenario **TS2**, we address the impact of preemption (and RTOS) on the profiling context. According to the configuration and setup, the preempted and preempting tasks run the same synthetic task, and the task UO (the one being preempted) is preempted exactly 4 times. This explains why instructions and cycles count in **TS2** are 5 times the one recorded in **TS0**. The same applies to data statistics. Also, the impact of the RTOS interrupts are proportional to the new duration of the experiment. Scenario **TS3** challenges the capability of TAP to isolate the profiling context in case of task preemption. Comparing the HPM values against those collected in the previous scenario, we observe that instruction count, cycle count, and data access clearly filter out the impact of RTOS and preempting tasks. Besides some negligible PMU jitter, the observed values are matching those gathered under **TS1**. The extra scenario **TS3bis** report the results for one of the preempting task, also being profiled: values that are essentially matching those from **TS3**, confirming the separation between contexts also with multiple tasks UO.

2) System-level events: The use of TAP on an integrated system is meant to enable the collection of HPM-based profiling information that can support various non-functional analyses. In this section we assess TAP functionalities on a multicore setup deploying a set of specialized validation benchmarks developed by Infineon for the TC397. These memory intensive benchmarks perform intensive accesses to configurable memory regions and data and differentiates on the specific type of ISA memory operation instruction and instruction interleaving: `ldd` (load double-word), `ladd` (load double-word and add), `madd` (multiply-add), `madd_move` (multiply-add and move to register). These kernels have been configured and deployed to mimic realistic data-dependencies among tasks in a multicore setup, which complicates resorting to stubbing and isolation scenarios. The goal is to demonstrate the library's ability to preserve the profiling context of a task under such scenario while collecting information that can be exploited to reason on specific timing aspects such as multi-core interference, where it is fundamental to correctly account for system-level events so that only those events happening while the task UO executes are considered. General-purpose tools like `perf` will instead consider all system-level events happening during the whole task execution window.

TABLE IV: TAP system-level profiling context.

	cnt ₁	cnt ₂	cnt ₃	cnt ₄	cnt ₅	cnt ₆
TS4	350,012	400,013	350,012	400,013	4,502,915	8,283,602
TS5	1,300,371	2,708,848	350,012	400,013	7,701,439	15,817,136
TS6	674,932	1,178,511	350,012	400,013	7,707,339	15,823,537

Results for system-level events are reported in Table IV. In each multicore scenario (TS5-TS7), we consider `ldd` as task UO, configured to access LMU0, while other tasks are configured to access different devices depending on the test scenario. Captured HPMs include `NG_LMU0_SRI_RDIN` (cnt₁) and `NG_LMU0_SRI_WR` (cnt₂), system-scope HPMs collectively

TABLE V: TAP overhead across context mgmt. scenarios

	Inter- rupts	Task C.S.	Store HPM	Reload MCDS	ERIKA	ERIKA + TAP	rel Δ	abs Δ
CS1	✓	-	-	-	428	456	1.065	28
CS2	✓	✓	-	-	2174	2203	1.013	29
CS3	✓	✓	-	✓	2100	4996	2.379	2896
CS4	✓	✓	✓	✓	2190	5600	2.557	3410
CS5	✓	✓	✓	-	2188	2952	1.349	764

CS1 RTOS interrupt with no task switch;

CS2 A task nUO is preempted by another task nUO;

CS3 A task nUO is preempted by a task UO;

CS4 A task UO gets preempted by another task UO;

CS5 A task UO gets preempted by a task nUO;

capturing accesses to the LMU0; `CPU0ms_LMU0_DMI_RD` (`cnt3`) and `CPU0ms_LMU0_DMI_WR` (`cnt4`), capturing core-level outgoing requests to the same device. We also report wall-time cycles (`cnt5`) and cycle counts (`cnt6`).

The HPMs collected for task `ldd` in **TS4** show the same count for core-level (`cnt3`, `cnt4`) and system-level (`cnt1`, `cnt2`) accesses to LMU0, since only `ldd` generates accesses to LMU0. As shown in core-level test scenarios, the difference between `cnt5` and `cnt6` confirms TAP is filtering activity from RTOS and other tasks. Under **TS5** and **TS6**, instead, tasks in Core 1 send contending requests to LMU0. These accesses cause multicore timing interference on `ldd`, with evident impact on both observed execution time (+71% in `cnt5`) and end-to-end time (+91% in `cnt6`). However, in **TS5** we disabled TAP system-level context-aware profiling support, which causes system-level HPM to count events that happen during the whole execution window of `ldd`, even when the latter is not actually executing (`cnt1` and `cnt2`). From a contention analysis perspective, those requests include requests that were not directly contending with the task UO. Conversely, **TS6** leverages TAP filtering at system-level and `cnt1`, `cnt2` cumulatively account for the actual contending requests that were triggered by tasks in Core 1 during the execution of `ldd`.

B. Overhead assessment

We assess TAP induced overheads by identifying and measuring different points in RTOS task management where TAP integration changes have been applied. In particular, we focus on RTOS interrupts, which may or not result in a context switch depending on the schedule. Based on the profiling setup, the preempting and preempted tasks can be under observation (UO) or not (nUO), which in turn may affect the housekeeping activities performed by TAP to preserve separation among profiling contexts (e.g. stop/start counters, save profiling context, reload profiling configuration). Table V compares the latency of specific context management scenarios (CS) with and without TAP integration, explicitly referencing the context management operations from TAP standpoint.

Scenarios **CS1** and **CS2** are those scenarios where TAP is not directly operating, so it is important to keep the overhead as negligible as possible. In **CS1** the scheduler schedules the same task that was running before the interrupt. Hence, whether the current task is UO or nUO there is no context switch, no need to store HPM values or change the MCDS

configuration. In this scenario, which is the most frequent operation at RTOS level, TAP overhead is limited to 28 cycles only (.065 relative increase). In **CS2**, a preemption happens but no task UO is involved. Therefore, similarly to **CS1** no TAP specific operation needs to be performed and the observed overhead is limited to 29 cycles only (.013 relative increase).

Scenarios **CS3** to **CS5** involves profiling context management. They happen quite less frequently than **CS1**, hence relatively larger overhead is acceptable. In **CS3**, a task UO is preempting a task nUO. While no HPM values need to be stored, the new task requires to reset the MCDS configuration according to its profiling setup. Dynamically re-configuring the MCDS is the most expensive TAP function, despite the implemented optimizations, contributing ~ 2848 extra cycles. The overhead in **CS3** therefore results in 4996 cycles against 2100 with no TAP, hence incurring 2896 extra cycles (1.38 relative increase). Scenario **CS4** is the most intrusive one as both preempting and preempted tasks are UO, requiring to store the HPM values of the preempted task and update the MCDS configuration for the new task. Again, the most significant contributor to the overhead is MCDS re-configuration process while the residual overhead for storing HPM would be limited to ~ 530 cycles. Under **CS5**, HPM values must be stored as the preempted task is UO but no MCDS re-configuration is required since the preempting task is nUO. TAP observed overhead is limited to 764 cycles only (.35 relative increase). It is worth noting that **CS4** and **CS3** can only happen with multiple task UO at the same time. Additionally, TAP implements a lazy mechanism where MCDS re-configuration in **CS3** only applies when the new task was not the last task UO being scheduled.

V. RELATED WORKS

Software-level profiling solutions available in the mainstream domains are design to support (average) performance characterization and optimization tasks. Many of these tools are normally integrated within commodity operating systems release, like `perf` [11] on Linux-based distributions, or have developed to be a de facto standard in performance profiling, like `PAPI` [25]. Both tools provide an abstraction layer that identifies generic HPM-based statistics that are mostly platform-independent (i.e., the involved HPM are typically available in mainstream processors) at the cost of not capturing specific hardware support. Process-level monitoring in mainstream tools like `perf` typically covers events directly triggered by the task under observation while do not guarantee any context preservation for system-level events. In our implementation we are tracking all (core and system level) events that happen while the task is executing within a monitored region, which is crucial, for example, to capture timing interference information.

The work in [26] attempts to define a performance library that better captures embedded real-time system requirements while at the same time fitting the standardized API offered by `PAPI` [25] interface. The concept of filtering out spurious events while monitoring the timing behavior of a task is

discussed in [27] but without considering finer granularities. In this work, we define a generic concept that takes full advantage of the advanced hardware support and support efficient monitoring of multiple regions within multiple applications at the same time by ensuring separation of profiling contexts.

VI. CONCLUSIONS

Profiling information based hardware events is becoming pivotal to support diverse tasks, from performance optimization, to hardware validation, to bandwidth regulation. In this work we focus on the collection of Hardware Performance Monitors to support timing analysis tasks for embedded time-critical systems, but nothing prevents from using TAP for performance, debugging and system regulation. We identify task-level monitoring and profiling context separation as inescapable requirements for capturing time-related information. We develop the concept of Task-Aware Profiling in time-critical systems and implement it on a lightweight, low-overhead profiling library tailored to a representative automotive setup. We evaluate the proposed tool on synthetic scenarios and automotive validation kernels, showing how the proposed approach can effectively ensure separation between profiling contexts and provide insightful timing information without compromising the overall application performance.

REFERENCES

- [1] International Organization for Standardization, “ISO 26262:2018. road vehicles – functional safety,” <https://www.iso.org/standard/68383.html>, 2018.
- [2] Radio Technical Commission for Aeronautics, “DO-178C - Software Considerations in Airborne Systems and Equipment Certification,” Tech. Rep., 2011. [Online]. Available: <https://www.rtca.org/content/publications>
- [3] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, “Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms,” in *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*. IEEE Computer Society, 2013, pp. 55–64. [Online]. Available: <https://doi.org/10.1109/RTAS.2013.6531079>
- [4] J. Nowotsch *et al.*, “Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement,” in *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*. IEEE Computer Society, 2014, pp. 109–118. [Online]. Available: <https://doi.org/10.1109/ECRTS.2014.20>
- [5] A. Saeed *et al.*, “Memory utilization-based dynamic bandwidth regulation for temporal isolation in multi-cores,” in *28th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2022, Milano, Italy, May 4-6, 2022*. IEEE, 2022, pp. 133–145. [Online]. Available: <https://doi.org/10.1109/RTAS54340.2022.00019>
- [6] S. H. Vanderleest and S. R. Thompson, “Measuring the impact of interference channels on multicore avionics,” *CoRR*, vol. abs/2101.02204, 2021. [Online]. Available: <https://arxiv.org/abs/2101.02204>
- [7] F. Boniol, J. Brunel, K. Delmas, C. Pagetti, and V. Jegu, “Modelling and analyzing multi-core cots processors,” in *11th European Congress on Embedded Real Time Software and Systems (ERTS 2022)*, Toulouse, France, 2022. [Online]. Available: <https://hal.science/hal-03761937>
- [8] H. G. T. David Radack and P. Parkinson, “Civil Certification of Multi-core Processing Systems in Commercial Avionics,” 2018.
- [9] E. Díaz, E. Mezzetti, L. Kosmidis, J. Abella, and F. J. Cazorla, “Modelling multicore contention on the aurixtm tc27x,” in *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*. ACM, 2018, pp. 97:1–97:6. [Online]. Available: <https://doi.org/10.1145/3195970.3196077>
- [10] J. Jalle *et al.*, “Bounding Resource Contention Interference in the Next-Generation Microprocessor (NGMP),” in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, Jan. 2016. [Online]. Available: <https://hal.science/hal-01259133>
- [11] V. Weaver, “Linux perf event features and overhead,” in *International Workshop on Performance Analysis of Workload Optimized Systems (FastPath 2013)*, 2013. [Online]. Available: <https://web.eece.maine.edu/~vweaver/papers/fastpath2013.pdf>
- [12] Rapita Systems, “Multicore Avionics Certification for High-integrity DO-178C projects,” <https://www.rapitasystems.com/products/mach178>, 2017.
- [13] A. Mascareñas González, Y. Bouchebaba, and L. Santinelli, “Multicore shared memory interference analysis through hardware performance counters,” in *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, Toulouse, France, Jan. 2020. [Online]. Available: <https://hal.science/hal-02446031>
- [14] Evidence, “ERIKKA Enterprise Manual V1.4.5,” <https://erika.tuxfamily.org/drupal/documentation.html>, 2012.
- [15] Infineon, “AURIXTM TC3XX User’s Manual V1.4.0,” <https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc3xx/>, 2019.
- [16] N. Stollon, *Infineon Multicore Debug Solution*. Boston, MA: Springer US, 2011, pp. 219–230. [Online]. Available: https://doi.org/10.1007/978-1-4419-7563-8_14
- [17] NXP, “T2080 Advanced QorIQ Debug and Performance Monitoring Reference Manual,” 2014, T2080DEBUGRM, Rev. A 04/2014.
- [18] Intel, “Intel Resource Director Technology (Intel RDT) on 2nd Generation Intel Xeon Scalable Processors Reference Manual, Rev. 1.0,” <https://www.intel.com/content/dam/develop/external/us/en/documents/180115-intel-rdtdscadelake-serverreferencemanual-806717.pdf>, 2019.
- [19] J. Pattee, S. M. Anik, and B. K. Lee, “Performance monitoring counter based intelligent malware detection and design alternatives,” *IEEE Access*, vol. 10, pp. 28 685–28 692, 2022. [Online]. Available: <https://doi.org/10.1109/ACCESS.2022.3157812>
- [20] E. Mezzetti, L. Kosmidis, J. Abella, and F. J. Cazorla, “High-integrity performance monitoring units in automotive chips for reliable timing v&v,” *IEEE Micro*, vol. 38, no. 1, pp. 56–65, 2018. [Online]. Available: <https://doi.org/10.1109/MM.2018.112130235>
- [21] ARM, “Architecture Reference Manual Supplement Memory System Resource Partitioning and Monitoring (MPAM),” <https://developer.arm.com/documentation/ddi0598/latest/>, 2022, (ID042622).
- [22] Lauterbach GmbH, “Trace32 powertrace,” <https://www.lauterbach.com/>, 2022.
- [23] R. Wilhelm *et al.*, “The worst-case execution-time problem - overview of methods and survey of tools,” *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, 2008. [Online]. Available: <https://doi.org/10.1145/1347375.1347389>
- [24] M. Günzel, H. Teper, K. Chen, G. von der Brüggen, and J. Chen, “On the equivalence of maximum reaction time and maximum data age for cause-effect chains,” in *35th Euromicro Conference on Real-Time Systems, ECRTS 2023, July 11-14, 2023, Vienna, Austria*, ser. LIPIcs, A. V. Papadopoulos, Ed., vol. 262. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 10:1–10:22. [Online]. Available: <https://doi.org/10.4230/LIPIcs.ECRTS.2023.10>
- [25] S. Browne, C. Deane, G. Ho, and P. Mucci, “Papi: A portable interface to hardware performance counters,” in *Proceedings of Department of Defense HPCMP Users Group Conference*, June 1999. [Online]. Available: <https://icl.utk.edu/publications/papi-portable-interface-hardware-performance-counters>
- [26] J. Giesen, E. Mezzetti, J. Abella, E. Fernández, and F. J. Cazorla, “epapi: Performance application programming interface for embedded platforms,” in *19th International Workshop on Worst-Case Execution Time Analysis, WCET 2019, July 9, 2019, Stuttgart, Germany*, ser. OASIcs, S. Altmeyer, Ed., vol. 72. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 3:1–3:13. [Online]. Available: <https://doi.org/10.4230/OASIcs.WCET.2019.3>
- [27] J. Giesen, E. Mezzetti, J. Abella, and F. J. Cazorla, “PRL: standardizing performance monitoring library for high-integrity real-time systems,” in *39th IEEE International Conference on Computer Design, ICCD 2021, Storrs, CT, USA, October 24-27, 2021*. IEEE, 2021, pp. 344–348. [Online]. Available: <https://doi.org/10.1109/ICCD53106.2021.00061>