ePAPI: Performance Application Programming Interface for Embedded Platforms

Jeremy Giesen
Universitat Politècnica de Catalunya, Spain
Barcelona Supercomputing Center, Spain

Enrico Mezzetti
Barcelona Supercomputing Center, Spain

Jaume Abella
Barcelona Supercomputing Center, Spain

Enrique Fernández
Universidad de Las Palmas de Gran Canaria, Spain

Francisco J. Cazorla
Barcelona Supercomputing Center, Spain

Abstract

Performance Monitoring Counters (PMCs) have been traditionally used in the mainstream computing domain to perform debugging and optimization of software performance. PMCs are increasingly considered in embedded time-critical domains to collect in-depth information, e.g. cache misses and memory accesses, of software execution time on complex multicore platforms. In main-stream platforms, standardized specifications and applications like the Performance Application Programming Interface (*PAPI*) and `perf` have been proposed to deal with variable PMC support across platforms, by providing a shared interface for configuring and collecting traceable events. However, no equivalent solution exists for embedded critical processors for which the user is required to deal with low-level, platform-specific, and error-prone manipulation of PMC registers. In this paper, we address the need for a standardized PMC interface in the embedded domain, especially in view to support timing characterization of embedded platforms. We assess the compatibility of the PAPI interface with the PMC support available on the AURIX TC297, a reference automotive platform, and we implement and validate ePAPI, the first functionally-equivalent and low-overhead implementation of PAPI for the considered embedded platform.

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

Keywords and phrases Monitoring counters, embedded systems

Digital Object Identifier 10.4230/OASIcs.WCET.2019.3

Funding This work has been partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant TIN2015-65316-P, the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 772773), the European Union's Regional Development Fund (ERDF) within the framework of the ERDF (FEDER) program in Catalonia 2014-2020 under the grant SDESI (2016 PROD00115), and the HiPEAC Network of Excellence. Jaume Abella and Enrico Mezzetti have been partially supported by MINECO under Ramon y Cajal and Juan de la Cierva-Incorporación postdoctoral fellowships number RYC-2013-14717 and IJCI-2016-27396 respectively.

19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019).
Editor: Sebastian Altmeyer; Article No. 3; pp. 3:1–3:13

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Embedded critical systems must undergo a strict verification and validation (V&V) process to guarantee that the deployed system behaves correctly, both from the functional and non-functional perspectives. For time-critical systems, correctness also depends on the timely delivery of the results. Performance and economic considerations are pushing towards the adoption of complex, heterogeneous high-performance Commercial Off-The-Shelf (COTS) systems even in the most conservative embedded system domains. As a matter of fact, the computational requirements of increasingly advanced software functionalities in automotive [4] and avionics [3] systems, just to mention a few, can only be met with the use of multicore COTS platforms featuring multiple levels of caches and specialized hardware accelerators. However, the architectural complexity of such systems is hindering the effectiveness of consolidated WCET analysis approaches [2].

The problem emanates from the fact that, while several designs [8, 9, 17, 19, 22] have been proposed to better factor in multicore contention in task's WCET estimates, for cost reasons those solutions have not been fully adopted by chip providers yet. In particular, industry is reluctant to re-design and re-verify already-verified functional unit blocks (FUBs).

Software solutions have been proposed to handle multicore contention in COTS processors that have limited hardware support for time predictability [10, 20, 15, 6]. As a first step, these solutions use event monitors to track tasks generated activities (events). Events are mapped to Performance Monitoring Counters (PMC), a set of specialized, software-visible configurable registers. As a second step, these approaches build on limiting per task (core) maximum shared resources utilization. To that end, usually the operating system or the hypervisor monitors task's activities using the available PMCs and suspends or restrains tasks' execution when the assigned budget is exhausted. Tracked and bounded events to control contention among tasks include per-task (and possibly per-type) access counts to different shared resources like caches and memories. Overall, PMCs and Performance Monitoring Units (PMUs) in general, are instrumental to timing analysis on complex COTS high-performance hardware platforms. In particular, PMUs can be exploited to provide finer-grained metrics to support timing analyses by, for example, capturing several aspects of the execution, such as suffered contention and maximum latencies, that can be used to refine static analysis assumptions and to support measurement-based timing analysis [14].

PMC support varies greatly across hardware platforms and even across models of the same platform. This diversity complicates the definition of a structured approach for the use of PMCs to support platform analysis. A structural and reusable approach for configuring and reading PMCs is necessary to abstract away from the low-level hardware details and to guarantee a correct manipulation of the registers, especially with respect to PMU configuration. In this respect, a standardization effort has led to the definition of kernel-level tools, like the Linux set of utilities `perf` [1], or shared common libraries for configuring and using monitoring counters, like the Performance Application Programming Interface (PAPI) [13]. Whereas these tools have reached an exceptional diffusion in mainstream and high-performance platforms, no equivalent solution is currently available for embedded reference platforms and RTOSs.

In this paper, we take a first step towards filling this gap by addressing the implementation of a common abstract PMC library for use in the embedded domain to enable the collection of relevant hardware events in a platform-independent way. To that end, we consider the PAPI library and evaluate the extent at which it could be used for fine-grained platform analysis, especially with respect to timing characterization. After assessing the compatibility

of PAPI with the PMC support available on the Infineon AURIX™ TC297 [11] platform, a reference platform in the automotive domain, we define, implement and validate ePAPI, a functionally-equivalent, low-overhead port of PAPI to the referred platform.

The remainder of this paper is organized as follows: Section 2 provides some background on PAPI and the use of PMCs in embedded critical domains; Section 3 discusses the design choices behind the implementation of PAPI on the AURIX™ TC297; Section 4 describes our validation approach and provides empirical evidence on the correctness and efficiency of the port. Finally, Section 5 provides some conclusions.

2 Background

PMCs in the embedded domain. The need for performance counters originally developed within the scope of mainstream processors, as a means for hardware and software developers to perform low-level debugging. PMCs have been later used also for coarse-grained performance optimization. It is only recently, however, that PMCs have been increasingly considered in embedded systems to support timing [10, 20, 15, 6], thermal [12], and power [21] analyses. With respect to timing analysis, the main scope of this paper, PMCs have been especially considered to analyze the contention effects in multicore COTS platforms, either to build analytical models [10, 5, 6] or to monitor and enforce access or usage quotas [20, 15] on shared hardware resources. All these approaches, however, rely on ad-hoc, platform and RTOS/hypervisor specific low-level functions to configure and collect information from PMCs. In this paper, we support these same methods by showing that a generic, reusable, and validated library can be defined that allows collecting PMC information without the need to enter into the platform-specific details.

Performance Application Programming Interface. PAPI [13], is a cross-platform library with supporting utilities that represents the de-facto standard for the collection of *hardware events* on mainstream hardware devices, including heterogeneous and virtual platforms. A hardware event is capable of detecting the occurrence of a specific activity generated by the running software. Events can be tracked using specific registers called *performance monitoring counters* (PMCs). To that end event monitors – that are not visible from the system/user software – are mapped to software-visible PMCs via some PMC-related control registers. Interestingly, while the number of available PMCs is normally limited to few dozens at most, the number of traceable events depends on the specific platform support and can be in the order of hundreds or even thousands [16]. This requires several runs to capture the desired event monitors, with few event monitors read per run.

PAPI aims to provide a uniform environment across platforms and provides a unified interface across processors. This is achieved by resorting to a two-layer architecture:

- a *portable layer* implements the API and machine-independent functions,
- whereas a *machine-specific layer* defines machine-dependent (and operating system dependent) functions and data structures, using kernel extensions, operating system calls, or assembly language to access and configure monitoring counters.

In practice, however, different events may be supported, indirectly supported, or even unsupported depending on the processor in use. This apparent inconsistency is solved in PAPI by supporting two types of hardware events:

- *preset events*, also known as predefined events, are a common set of events deemed relevant and useful for performance characterization. These events are typically found in CPUs with debug support units and give access to several hardware events related to memory hierarchy, cache coherence protocol, cycle and instruction counts, functional units, pipeline status, etc.

- Some events, however, are only available on specific platforms. The *native events* set identifies the full set of events that can be tracked on a given CPU. Native events can be configured and traced directly, even if there is no corresponding preset event available.

The interface is implemented as an instrumentation library in C and needs to be compiled/linked together with the target application we want to characterize. The interface actually supports two different usage modes, which can be used side by side since they share the internal structures on which PAPI is built. A high-level API provides the basic ability to start, stop, and read the counters for a specified list of events. A low-level API, instead, manages hardware events in user-defined groups called *event sets* (preset or native) providing fine-grained measurement and control of the PAPI interface. The high-level interface offers a more lightweight semantics but at the cost of reduced flexibility.

3 Porting PAPI to an embedded platform

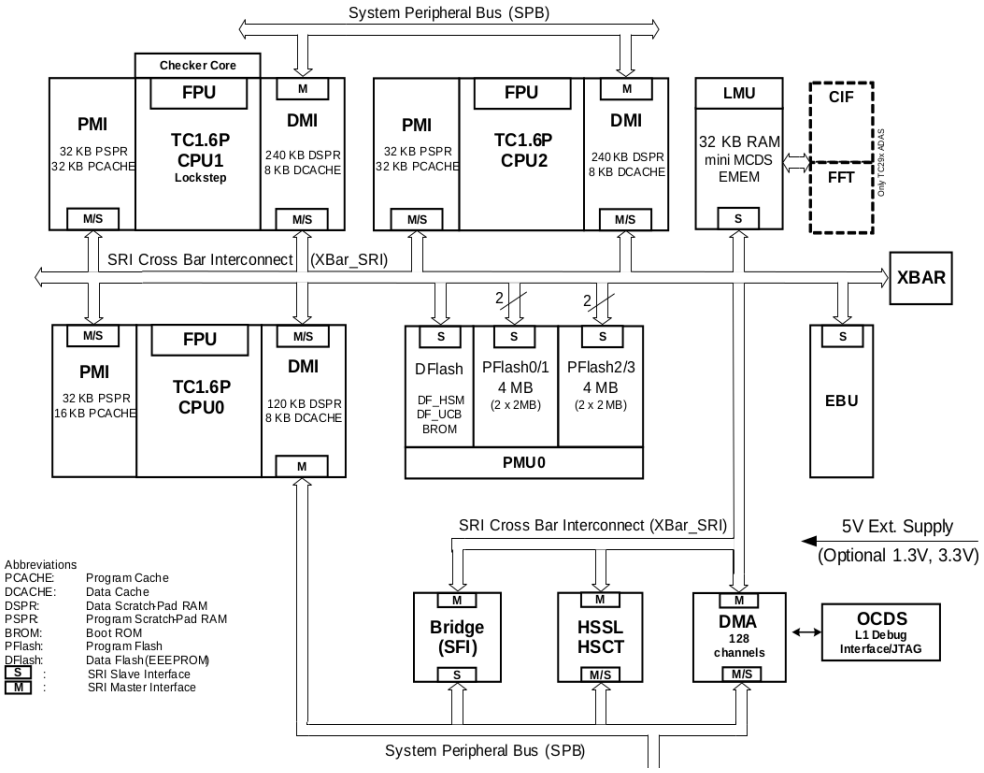

Figure 1 TC297 block diagram (from [11]).

In this work we assess the porting of the general-purpose PAPI library to an embedded target, to evaluate whether the same support for performance monitoring could be efficiently and effectively adopted in embedded systems. In our porting, we target the Infineon AURIX™ TriCore TC297 platform [11], a representative COTS platform in the automotive domain.

3.1 Reference platform

The TC297 block diagram is reported in Figure 1. The platform features three performance-efficient TriCore TC1.6P cores, all being equipped with separate core-local memories (scratchpads and caches) for instructions and data. Processors are then connected to each other and

to a shared "memory system" through the Shared Resource Interconnect (SRI) crossbar, which supports multiple requests to different slaves (memory regions or peripherals) to be served in parallel without incurring contention effects. The shared memory system comprises an SRAM device, accessed via the Local Memory Unit (LMU), and a FLASH device, accessed via the Program Memory Unit (PMU). The PMU offers 4 independent interfaces to the program flashes (2MB each) and 1 interface for the data flash. The LMU provides access to volatile memory resources whose primary purpose is to provide 32 kbytes of local memory for general purpose usage. Each of the TC1.6P cores implements three pipelines that allow the platform to support dual instruction issuing in parallel into an integer pipeline and load/store pipeline. The third pipeline is providing specialized support for zero-overhead loop instructions. Despite belonging to the same processor model, the three cores have slightly different memory specifications. In Core 1 and Core 2, first-level instruction and data caches and scratchpads are 32KB, 8KB, 32KB, and 240KB respectively, with slightly smaller capacities in the case of Core 0's, with 16KB, 8KB, 32KB, and 120KB. Finally, Core 1 is associated with a checker core, in lock-step mode.

Debug support and monitoring counters. A porting of PAPI to any target platform depends on the debug support and observability available. At the core level, the AURIX™ TC297 debug support features 5 dedicated PMC registers per core. Two registers are dedicated to count executed cycles (`CCNT`) and instructions (`ICNT`). The remaining three multiplexing registers (`M1CNT`, `M2CNT`, and `M3CNT`) can be configured to count the different hardware events supported by the platform by setting the proper bits in the Counter Control Register `CCTRL`.

The AURIX™ TC297 multiplexing registers can be enabled and configured to count 12 hardware events related to pipeline stalls (on the three different pipelines), cache behavior, and branch and SRI crossbar statistics:

- `IP_DISPATCH_STALL` : incremented on every cycle in which the Integer dispatch unit is stalled for whatever reason.
- `LS_DISPATCH_STALL` : incremented on every cycle in which the Load-Store dispatch unit is stalled for whatever reason.
- `LP_DISPATCH_STALL` : incremented on every cycle in which the Loop dispatch unit is stalled for whatever reason.
- `MULTI_ISSUE` : incremented in any cycle where more than one instruction is issued.
- `PCACHE_HIT` : incremented whenever the target of a cached fetch request from the fetch unit is found in the program cache.
- `PCACHE_MISS` : incremented whenever the target of a cached fetch request from the fetch unit is not found in the program cache and hence a bus fetch is initiated.
- `DCACHE_HIT` : incremented whenever the target of a cached request from the Load-Store unit is found in the data cache.
- `DCACHE_MISS_CLEAN` : incremented whenever the target of a cached request from the Load-Store unit is not found in the data cache and hence a bus fetch is initiated with no dirty cache line eviction.
- `DCACHE_MISS_DIRTY` : incremented whenever the target of a cached request from the Load-Store unit is not found in the data cache and hence a bus fetch is initiated with the write-back of a dirty cache line.
- `TOTAL_BRANCH` : incremented in any cycle in which a branch instruction is in a branch resolution stage of the pipeline.

- PMEM_STALL : incremented whenever the fetch unit is requesting an instruction and the instruction memory is stalled for whatever reason.
- DMEM_STALL : incremented whenever the Load-Store unit is requesting a data operation and the data memory is stalled for whatever reason.

Table 1 summarizes the events that can be traced through the (multiplexed) performance-monitoring counters with the respective configuration of the CCTRL register.

Table 1 Multi-Count Configuration (TC1.6P).

	Monitoring counters configuration registers		
CCRTL bits	M1CNT	M2CNT	M3CNT
000	IP_DISPATCH_STALL	LS_DISPATCH_STALL	LP_DISPATCH_STALL
001	PCACHE_HIT	PCACHE_MISS	MULTI_ISSUE
010	DCACHE_HIT	DCACHE_MISS_CLEAN	DCACHE_MISS_DIRTY
011	TOTAL_BRANCH	PMEM_STALL	DMEM_STALL

3.2 Selection and mapping of PAPI events

The ePAPI implementation only supports a selection of PAPI events. The selection was made according to two principles. First, while PAPI supports more than 100 preset events, actual implementations normally support only a subset of events. The TC297 is not an exception and several preset events are not supported. For instance, more than 30 events related to the L2 and L3 cache behavior, are inherently not supported in the target system as it exhibits a flat cache hierarchy. Second, the porting is tailored to the use of those performance monitors we consider more relevant for the embedded domain characterization, that is timing and/or energy analyses, multicore contention analysis, as well as average performance analysis.

Table 2 summarizes the results of our selection over PAPI event. Each selected event is associated with the available specification [11] and to the supporting counter in the TC297, with the associated PMC multiplexed register.

Mapping preset events is generally straightforward, with the exception of some PAPI events that could only be covered by combining the information from more than one PMC. In a few cases, the current support in the TC297 could only provide an over-approximation of the PAPI event. As an example, the PAPI_RES_STL event (cycles stalled on any resource) could be loosely upper-bounded by TC297 XMEM_STALL and XX_DISPATCH_STL events (as they partially overlap). Moreover, due to the inflexible configuration of the TC297 CCTRL (see Table 1), some events can only be covered by combining two values read from the same multiplexed PMC register. In this case, two executions are required to collect all necessary evidence from the PMCs. This last class of events is not directly implemented in ePAPI, thus the programmer is responsible for measuring and combining the different event counts as required. In addition, the last six entries in Table 2 report the native events supported by the TC297 that do not fall in PAPI preset event set.

4 ePAPI implementation and validation

The full PAPI specification includes more than 70 different functions. The ultimate objective of this study was not to provide a full implementation of the interface but, more precisely, to instantiate it to a representative embedded platform. The pruning over PAPI functions was mainly a consequence of the particular hardware-software configuration considered

Table 2 AURIX events to PAPI interface analysis.

PAPI event	Description	Counter source	Counter(s)
PAPI_BRU_IDL	Cycles branch units are idle	CCNT − TOTAL_BRANCH	CCNT − M1CNT
PAPI_L1_DCH	Level 1 data cache hit	DCACHE_HIT	M1CNT
PAPI_L1_ICH	Level 1 instruction cache hit	PCACHE_HIT	M1CNT
PAPI_L1_ICM	Level 1 instructions cache miss	PCACHE_MISS	M2CNT
PAPI_L1_DCA	Level 1 data cache accesses	DCACHE_HIT + DCACHE_M_C + DCACHE_M_D	M1CNT + M2CNT + M3CNT
PAPI_L1_DCM	Level 1 data cache misses	DCACHE_M_C + DCACHE_M_D +	M2CNT + M3CNT
PAPI_L1_ICA	Level 1 instruction cache access	PCACHE_HIT + PCACHE_MISS	M1CNT + M2CNT
PAPI_L1_ICR	Level 1 instruction cache reads	PCACHE_HIT + PCACHE_MISS	M1CNT + M2CNT
PAPI_MEM_SCY	Cycles stalled waiting for memory accesses	DMEM_STALL + PMEM_STALL	M2CNT + M3CNT
PAPI_L1_TCA	Level 1 total cache accesses	PCACHE_HIT+ PCACHE_MISS+ DCACHE_HIT+ DCACHE_M_C+ DCACHE_M_D	M1CNT(x2)+ M2CNT(x2)+ M3CNT
PAPI_L1_TCH	Level 1 total cache hits	PCACHE_HIT+ DCACHE_HIT	M1CNT(x2)
PAPI_L1_TCM	Level 1 total cache misses	PCACHE_MISS+ DCACHE_M_C+ DCACHE_M_D	M1CNT(x2)+ M3CNT
PAPI_TOT_CYC	Total cycles	CCNT	CCNT
PAPI_TOT_INS	Instructions completed	ICCNT	ICNT
PMEM_STALL	Cycles where the program memory is stalled.	PMEM_STALL	M2CNT
DMEM_STALL	Cycles where the data memory is stalled.	DMEM_STALL	M3CNT
MULTI_ISSUE	Cycles where more than one instruction is issued.	MULTI_ISSUE	M3CNT
IPDISP_STL	Cycles in which Integer Dispatch Unit is stalled.	IP_DISPATCH_STL	M1CNT
LSDISP_STL	Cycles in which Load-Store Dispatch Unit is stalled.	LS_DISPATCH_STL	M2CNT
LPDISP_STL	Cycles in which Loop Dispatch Unit is stalled.	LP_DISPATCH_STL	M3CNT

since ePAPI has been designed assuming a bare-metal environment and on top of the PMC support offered by the AURIX TC297. In particular, functions have not been selected for implementation in ePAPI for the following criteria:

1. Focus on CPU performance: while the latest versions of PAPI support diverse hardware components (e.g., network), we focused on CPU performance only as the central scope for timing characterization. We, therefore, discarded functions related to component selection.
2. Lack of platform support for Floating-Point Unit (FPU) events: despite the AURIX TC297 is equipped with a fully pipelined single-precision FPU [11], no support is provided to monitor hardware events related to the FPU module.
3. Lack of operating system support: this led us to discard all functions related to the collection of performance metrics on a thread or task basis, as well as those related to process virtualization.

4. Lack of standard output: we assume ePAPI to be deployed in scenarios where no visual output means is available, so that all the library outputs, including raw data and results, are not directly shown to the user but are stored into a predefined, configurable memory region. In reason of this less interactive use of ePAPI, we considered it unnecessary to implement those functions responsible for querying the system for hardware configurations or for printing any information on a screen.

Table 3 Subset of PAPI functions implemented in ePAPI.

High-level supported functions	
int PAPI_start_counters	Starts counting HW events
int PAPI_read_counters	Copies current PMCs to internal array and reset counters
int PAPI_accum_counters	Adds PMCs values to internal array and reset counters
int PAPI_stop_counters	Stops counting events and return current PMCs
int PAPI_num_counters	Gives the number of HW counters available in the system
int PAPI_epc	Gives events per cycle, real and processor time
int PAPI_ipc	Gives instructions per cycle, real and processor time
Low-level supported functions	
int PAPI_start	Starts counting HW events in an event set
int PAPI_read	Read HW events from an event set with no reset
int PAPI_accum	Accumulates and resets HW events from an event set
int PAPI_stop	Stops counting HW events in event set and return PMCs
int PAPI_reset	Resets the HW event counts in an event set
int PAPI_create_eventset	Creates a new empty event set
int PAPI_add_event	Adds a single HW event to an event set
int PAPI_add_events	Adds array of HW events to an event set
int PAPI_add_named_event	Adds an HW event by name to a PAPI event set
int PAPI_remove_event	Removes a HW event from a PAPI event set
int PAPI_remove_events	Removes an array of PAPI events from an event set
int PAPI_remove_named_event	Removes a named event from a PAPI event set
int PAPI_cleanup_eventset	Removes all PAPI events from an event set
int PAPI_event_name_to_code	Translates an PAPI event name into an integer event code
int PAPI_list_events	Returns a list of the HW events in an event set
int PAPI_num_events	Returns the number of HW events in an event set
int PAPI_state	Returns the counting state of an event set

As a result of a preliminary analysis of PAPI high-level and low-level interfaces, we restricted the number of functions to be included in ePAPI. This first ePAPI implementation supports almost all (7 out of 10) functions defined by the PAPI high-level interface but only a small subset (17 out of 66) of the low-level interface. Table 3 shows the subset of PAPI functions for which an implementation is provided in ePAPI on top of the AURIX TriCore. Full function signatures were omitted for the sake of clarity. The only high-level functions not implemented are *int PAPI_num_components*, related to component selection, and *int PAPI_flips, int PAPI_flops*, delivering high-level FPU statistics. Several low-level functions were not ported to ePAPI for the reasons above.

In the current ePAPI implementation, the results of the invocation of ePAPI functions, which includes PMCs and other relevant information, are mapped to a small area in the local Data Scratchpad (DSPR) of the monitored core, where the collected PMC values can be easily gathered. Alternative platform-specific output mechanisms can be defined (e.g., the EMEM module in the AURIX).

```
int Events[NUM_EVENTS] = {PAPI_TOT_CYC};
long long values[NUM_EVENTS];

/* Start counting events */
PAPI_start_counters(Events, NUM_EVENTS);

//Place code under analysis here

/* Read counters */
PAPI_read_counters(values, NUM_EVENTS);

//Place more code under analysis here

/* Add the counters */
PAPI_accum_counters(values, NUM_EVENTS);

//Place more code under analysis here

/* Stop counting events */
PAPI_stop_counters(values, NUM_EVENTS);
```

(a) PAPI use example.

```
int Events[NUM_EVENTS] = {PAPI_TOT_CYC};
long long values[NUM_EVENTS];

/* Start counting events */
ePAPI_start_counters(Events, NUM_EVENTS);

//Place code under analysis here

/* Read counters */
ePAPI_read_counters(values, NUM_EVENTS);

//Place more code under analysis here

/* Add the counters */
ePAPI_accum_counters(values, NUM_EVENTS);

//Place more code under analysis here

/* Stop counting events */
ePAPI_stop_counters(values, NUM_EVENTS);
```

(b) ePAPI use example.

Figure 2 Example uses of PAPI and ePAPI.

For the low-level aspects, ePAPI implementation builds on the adaptation to the TC297 of the `PMClib` library, a low-level, basic PMC interface developed as part of a previous study [6]. This library, entirely developed in assembly, allows direct control of the `CCTRL` register and manipulation of PMCs values.

ePAPI example. To evaluate the signature-level equivalence of our porting, Figure 2 compares two examples of the use of the high-level API of PAPI and ePAPI. In fact, the interface of ePAPI matches perfectly the one offered by PAPI. The only difference between the two implementations is found in the "e" prefix, which was used to make ePAPI implementation distinguishable but can be dropped for full compatibility. Sharing the same signature makes the use of ePAPI easier to users with PAPI background.

4.1 Validation of ePAPI

The AURIX™ TC297 implementation of ePAPI has been validated to check whether the functional behavior meets the software specification from PAPI documentation, by conducting a proper functional testing campaign. As a major requirement in ePAPI, and as a relevant difference with respect to mainstream PAPI, library calls are also required to incur low overhead, to avoid the performance monitoring process to interfere with the application being monitored, i.e., *probe effect* [18]. As a preliminary step, we validated correctness and accuracy of PMCs exploiting the low-level `PMClib` library (adapted from [6]) to measure small ad-hoc benchmarks that cause a known amount of events to be triggered for each traceable event. The functional correctness of ePAPI has been validated building on the same library and ad-hoc benchmarks for both the high-level and low-level APIs.

We assessed ePAPI consistency and accuracy against the `PMClib` library, which we know manipulates PMCs at a very low level with no unnecessary overhead. Clearly, the generic infrastructure of ePAPI (especially in its high-level) interface cannot have zero impact on the execution, as it necessarily requires more instructions to be executed. However, it is important that this impact is low and bounded. The validation of this specific requirement consisted of executing the same test programs using either the `PMClib` library or ePAPI interfaces (both high- and low-level) and compare the obtained PMC values.

Table 4 Results of ePAPI overheads validation.

PAPI Event	Bare-metal	ePAPI Read	ePAPI accum.	Bare-metal	ePAPI Read	ePAPI accum.	Bare-metal	ePAPI Read	ePAPI accum.
ePAPI_BRU_IDL	Cyc. Idle Branch Unit			Tot. Cycles			Tot. Instructions		
	70.017	+27	+28	100.019	+30	+31	80.011	+11	+11
ePAPI_L1_ICA	ICache accesses			Tot. Cycles			Tot. Instructions		
	499.002	+0	+0	1.029.597	+3	+3	1.017.010	+12	+12
ePAPI_L1_ICR	ICache reads			Tot. Cycles			Tot. Instructions		
	499.002	+0	+0	1.029.597	+23	+23	1.017.010	+12	+12
ePAPI_L1_ICH	ICache hits			Tot. Cycles			Tot. Instructions		
	498.890	+0	+0	1.029.595	+25	+26	1.017.010	+12	+12
ePAPI_L1_ICM	ICache misses			Tot. Cycles			Tot. Instructions		
	124.987	+0	+0	2.879.511	+23	+23	1.513.009	+11	+11
ePAPI_L1_DCA	DCache accesses			Tot. Cycles			Tot. Instructions		
	1.000.000	+0	+0	1.029.599	+20	+31	1.017.009	+11	+11
ePAPI_L1_DCH	DCache hits			Tot. Cycles			Tot. Instructions		
	999.900	+0	+0	1.029.597	+22	+22	1.017.009	+11	+11
ePAPI_L1_DCM	DCache misses			Tot. Cycles			Tot. Instructions		
	6.001	+0	+0	299.616	+16	+17	32.010	+12	+12
ePAPI_MEM_SCY	Stalls on Mem. accesses			Tot. Cycles			Tot. Instructions		
	290.336	+0	+0	430.579	+25	+25	65.011	+11	+11
PMEM_STALL	Stall on Program memory			Tot. Cycles			Tot. Instructions		
	127	+3	+3	160.061	+32	+33	143.011	+11	+11
DMEM_STALL	Stall cycles on Data memory			Tot. Cycles			Tot. Instructions		
	301.714	+0	+0	401.820	+28	+29	40.011	+11	+11
MULTI_ISSUE	Multiple issues			Tot. Cycles			Tot. Instructions		
	100.014	+3	+1	120.040	+33	+34	230.050	+12	+12
IP_DISPATCH_STALL	Instr. dispatch stalls			Tot. Cycles			Tot. Instructions		
	1.000	+0	+0	17.014	+33	+34	17.010	+12	+12
LS_DISPATCH_STALL	Load/store dispatch stalls			Tot. Cycles			Tot. Instructions		
	2.002	+10	+10	17.014	+33	+34	17.010	+12	+12
LP_DISPATCH_STALL	Load/store dispatch stalls			Tot. Cycles			Tot. Instructions		
	1.001	+0	+0	1.008.032	+25	+26	2.006.014	+11	+11

Table 4 compares the PMC values collected running specific benchmarks for all events supported in ePAPI. For a given event, the same benchmark is measured using `PMClib` and ePAPI high-level interfaces, considering "plain" and accumulated PMC reads. We focused on the high-level interface as it is the one potentially incurring more overhead. For each experiment, we report the counts over the relevant event, executed cycles, and instructions.

Results show that the overhead incurred by ePAPI is generally negligible. The only observed differences were on the number of instructions and cycles, while no overhead was observed on the specific event counts. The additional counts for `ePAPI_BRU_IDL` are to be interpreted as ePAPI only slightly affecting the branch unit. In fact, ePAPI is implemented in a way that counters are enabled and disabled to guarantee that the measured events belong to the program under analysis. ePAPI always executes around 11-12 instructions more than `PMClib`, due to the additional instructions and data used for generic PMC management. This is reflected in a difference in executed cycles varying from 25 to 28 cycles.

ePAPI limitations. The porting of ePAPI to the TC297 was motivated by the need for a common standardized interface for performance characterization of embedded systems, mainly to support timing and multicore contention analysis. As a result of our analysis on

the PAPI interface and on the support available in the specific target platform, we identified some limitations as well as some desirable characteristics that could not be matched owing to the limitations of PAPI, peculiarities of embedded targets or of the selected platform itself.

From the PAPI perspective, we observe that the subset of the interface defined by preset events is necessarily generic, and mostly designed to collect performance metrics for optimization purposes. Despite the use of native events can cover more relevant events from the embedded domain perspective (e.g., stalls events to characterize contention), not having those events in the cross-platform interface partially defeats the benefits of having a standardized interface across embedded targets.

From the AURIX™ TC297 perspective, we suffered from the limited PMC support available, which is pretty limited compared to the support available in conventional processors. This is not generally the case in more advanced embedded targets, and it was indeed the main cause for discarding PAPI functionalities from the porting. Considering the potential uses of the interface, the PMC support in the TC297 does not allow to characterize contention effects with satisfactory precision as stall events cannot be associated to the different target in the SRI. For instance, the `PMEM_STALL` event is counting stalls cycles suffered when fetching code from any PFlash interface, the LMU or even non-local scratchpads.

5 Conclusions and future directions

With the goal of supporting timing analysis of complex COTS multicore processors, in this paper we have reported our investigation on the adoption of a standardized performance monitoring interface for embedded targets. To that end, we considered the general-purpose PAPI specification and assessed it against the available PMC support in the AURIX™ TC297, a representative platform in the automotive domain. We identified a compatible subset of PAPI preset events and some platform-specific native events. We have developed and validated ePAPI, a functionally-equivalent and low-overhead implementation of PAPI for the AURIX™ TC297. As future directions, we are interested in the extension of ePAPI to support RTOS-based PAPI functions (e.g., supporting task- and thread-level PMCs) by considering the TC297-compatible Erika RTOS [7]. Further, since the first implementation of ePAPI is tailored to the TC297, we are also considering to extend ePAPI to different embedded platforms, preferably with wider PMC support.

References

1 perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/index.php/Main_Page`.

2 Jaume Abella, Carles Hernández, Eduardo Quiñones, Francisco J. Cazorla, Philippa Ryan Conmy, Mikel Azkarate-askasua, Jon Pérez, Enrico Mezzetti, and Tullio Vardanega. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems, SIES 2015, Siegen, Germany, June 8-10, 2015*, pages 39–48. IEEE, 2015. `doi:10.1109/SIES.2015.7185039`.

3 Airbus. Global Networks, Global Citizens. 2018-2037. Global Market Forecast. `https://www.airbus.com/aircraft/market/global-market-forecast.html`, 2018.

4 ARM. ARM Expects Vehicle Compute Performance to Increase 100x in Next Decade. `https://www.arm.com/company/news/2015/04/arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade`, 2015.

5 Dakshina Dasari, Vincent Nélis, and Benny Akesson. A framework for memory contention analysis in multi-core platforms. *Real-Time Systems*, 52(3):272–322, 2016. `doi:10.1007/s11241-015-9229-9`.

6 Enrique Díaz, Enrico Mezzetti, Leonidas Kosmidis, Jaume Abella, and Francisco J. Cazorla. Modelling multicore contention on the AURIX^{TM} TC27x. In *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*, pages 97:1–97:6. ACM, 2018. doi:10.1145/3195970.3196077.

7 Evidence. Erika Enterprise RTOS v3. http://www.erika-enterprise.com/, 2019.

8 Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Trans. Design Autom. Electr. Syst.*, 14(1):2:1–2:24, 2009. doi:10.1145/1455229.1455231.

9 Carles Hernández, Jaume Abella, Francisco J. Cazorla, Alen Bardizbanyan, Jan Andersson, Fabrice Cros, and Franck Wartel. Design and Implementation of a Time Predictable Processor: Evaluation With a Space Case Study. In Marko Bertogna, editor, *29th Euromicro Conference on Real-Time Systems, ECRTS 2017, June 27-30, 2017, Dubrovnik, Croatia*, volume 76 of *LIPIcs*, pages 16:1–16:23. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.ECRTS.2017.16.

10 Rafia Inam, Mikael Sjödin, and Marcus Jägemar. Bandwidth measurement using performance counters for predictable multicore software. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation, ETFA 2012, Krakow, Poland, September 17-21, 2012*, pages 1–4. IEEE, 2012. doi:10.1109/ETFA.2012.6489714.

11 Infineon. *AURIX™ TC29x B-Step 32-Bit Single-Chip Microcontroller - User's Manual V1.3 2014-12*. 2019.

12 Kyeong-Jae Lee and Kevin Skadron. Using Performance Counters for Runtime Temperature Sensing in High-Performance Processors. In *19th International Parallel and Distributed Processing Symposium (IPDPS 2005), CD-ROM / Abstracts Proceedings, 4-8 April 2005, Denver, CO, USA*. IEEE Computer Society, 2005. doi:10.1109/IPDPS.2005.448.

13 Kevin London, Shirley Moore, Phil Mucci, Keith Seymour, and Richard Luczak. The PAPI Cross-Platform Interface to Hardware Performance Counters. In *Department of Defense Users' Group Conference Proceedings*, pages 18–21, Biloxi, Mississippi, June 2001.

14 Enrico Mezzetti, Leonidas Kosmidis, Jaume Abella, and Francisco J. Cazorla. High-Integrity Performance Monitoring Units in Automotive Chips for Reliable Timing V&V. *IEEE Micro*, 38(1):56–65, 2018. doi:10.1109/MM.2018.112130235.

15 Jan Nowotsch, Michael Paulitsch, Daniel Buhler, Henrik Theiling, Simon Wegener, and Michael Schmidt. 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*, pages 109–118. IEEE Computer Society, 2014. doi:10.1109/ECRTS.2014.20.

16 NXP/Freescle. *e6500 Core Reference Manual - E6500RM Rev 0 06/2014*. 2014.

17 Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015. doi:10.1016/j.sysarc.2015.04.002.

18 Young Wn Song and Yann-Hang Lee. On the existence of probe effect in multi-threaded embedded programs. In Tulika Mitra and Jan Reineke, editors, *2014 International Conference on Embedded Software, EMSOFT 2014, New Delhi, India, October 12-17, 2014*, pages 18:1–18:9. ACM, 2014. doi:10.1145/2656045.2656062.

19 Theo Ungerer, Francisco J. Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quiñones, Mike Gerdes, Marco Paolieri, Julian Wolf, Hugues Cassé, Sascha Uhrig, Irakli Guliashvili, Michael Houston, Florian Kluge, Stefan Metzlaff, and Jörg Mische. Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability. *IEEE Micro*, 30(5):66–75, 2010. doi:10.1109/MM.2010.78.

20 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 *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*, pages 55–64. IEEE Computer Society, 2013. doi:10.1109/RTAS.2013.6531079.

21 Reza Zamani and Ahmad Afsahi. A study of hardware performance monitoring counter selection in power modeling of computing systems. In *2012 International Green Computing Conference, IGCC 2012, San Jose, CA, USA, June 4-8, 2012*, pages 1–10. IEEE Computer Society, 2012. doi:10.1109/IGCC.2012.6322289.

22 Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. FlexPRET: A processor platform for mixed-criticality systems. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*, pages 101–110. IEEE Computer Society, 2014. doi:10.1109/RTAS.2014.6925994.