Analysis of Performance Monitoring Counter support and Implementation of Performance Application Programming Interface (PAPI) on an Automotive chip

Author: Jeremy Jens Giesen León

Specialization in *Computer Engineering*

Degree in *Informatics Engineering*

Escuela de Ingeniería Informática (EII)

Universidad de Las Palmas de Gran Canaria (ULPGC)

Tutors:

Enrique Fernández García[1]

Enrico Mezzetti[2]

June, 2018

[1]Universidad de Las Palmas de Gran Canaria (ULPGC)

[2]Barcelona Supercomputing Center (BSC-CNS)

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

TFT04

SOLICITUD DE DEFENSA DE TRABAJO DE FIN DE TÍTULO

D/Dª Jeremy Jens Giesen León, autor del Trabajo de Fin de Título "*Analysis of Performance Monitoring Counter support and Implementation of Performance Application Programming Interface (PAPI) on an Automotive chip"*,
correspondiente a la titulación Grado en Ingeniería Informática,
en colaboración con la empresa/proyecto (indicar en su caso) Barcelona Supercomputing Center

S O L I C I T A

que se inicie el procedimiento de defensa del mismo, para lo que se adjunta la documentación requerida.

Asimismo, con respecto al registro de la propiedad intelectual/industrial del TFT, declara que:

[ ] Se ha iniciado o hay intención de iniciarlo (defensa no pública).
[X] No está previsto.

Y para que así conste firma la presente.

Las Palmas de Gran Canaria, a 29 de Mayo de 2018.

El estudiante

Firmado por GIESEN LEON

Fdo.: Jeremy Jens Giesen León

A rellenar y firmar **obligatoriamente** por el/los tutor/es

En relación a la presente solicitud, se informa:

[X] Positivamente

[ ] Negativamente
(la justificación en caso de informe negativo deberá incluirse en el TFT05)

Firmado por FERNANDEZ GARCIA ENRIQUE - 43253427H el día 31/05/2018 con un

Fdo.: Enrique Fernández García

Fdo.: Enrico Mezzetti

DIRECTOR DE LA ESCUELA DE INGENIERÍA INFORMÁTICA

Lista de comprobación de documentación adjunta (resumida, detalles en el Manual Operativo)

[X] Memoria (máximo 100 pág. ~30000 palabras) en formato pdf con logos e identificación del Trabajo en la portada y, tras ella, la primera página de este TFT04 insertada (el pdf resultante debe firmarse digitalmente, ubicando las firmas en el espacio previsto para ello en el TFT04).

[X] Resumen en formato txt en español e inglés (un único fichero txt con límite de 120 palabras por idioma).

[X] Código (en caso de que el TFT lo incluya, un directorio, o bien fichero comprimido, o bien un txt con instrucciones para acceder al repositorio).

- Sólo en el caso de que en este impreso TFT04 se haya indicado que NO está previsto iniciar trámite de registro de la propiedad intelectual/industrial …

[X] Impreso relativo a la Difusión en Abierto del TFT en Biblioteca ULPGC como pdf firmado de forma digital.

NOTA: si se marca el registro de la propiedad intelectual sólo los tutores y el tribunal pueden asistir a la defensa.

Procedimiento de entrega: se enviarán instrucciones concretas a través del Moodle.

Agradecimientos

Quiero en primer lugar agradecer al Barcelona Supercomputing Center y en especial al grupo CAOS por acogerme y darme tanto los medios, como soporte humano y financiero para hacer posible este Trabajo de Fin de Grado. Quiero mostrarle mi agradecimiento a Francisco Javier Cazorla Almeida, Enrico Mezzetti y a Enrique Fernández García por guiarme y darme la oportunidad de realizar este trabajo relacionado con la investigación en el campo de la arquitectura de computadores, así como por los recursos dedicados al desarrollo del mismo entre los que se encuentran tanto el material, los desplazamientos, así como su propio tiempo. También me gustaría citar a las personas que me ha dado soporte, en particular a Mikel Fernandez, sobre todo por su ayuda con los contratiempos de las fases más avanzadas del proyecto. No cabe duda de que el trabajo realizado me ha dado la oportunidad de definir y encauzar el rumbo al que quiero dirigir mi carrera profesional.

Por otro lado, me gustaría agradecer de forma general a todo el Personal Docente Investigador, miembros del Personal de Administración y Servicios y en definitiva a los compañeros que me han tendido la mano para ayudarme cuando lo he necesitado. De este último grupo me gustaría destacar a mi compañero y amigo Javier Enrique Barrera Herrera por su eterna y contagiosa ilusión, disposición y apoyo a lo largo de todos estos años.

Me gustaría hacer una especial mención al grupo de compañeros y amigos con los que he compartido tantos buenos momentos y viajes a lo largo de todo el Grado, a los que me gustaría recordarles que disponen de mi total disponibilidad y amistad de forma indefinida.

Por último y por ello no menos importante, quiero expresar mi más profundo agradecimiento por el apoyo incondicional y desinteresado que he recibido de mi familia y pareja a lo largo de todo el Grado, por haber sido capaces de entender mi limitada disponibilidad hacia ellos especialmente en los últimos momentos del mismo y por haber hecho de mi la persona que soy hoy.

Achnowledgments

First of all, I would like to thank the Barcelona Supercomputing Center and especially the CAOS group for welcoming me and giving me both the means and the human and financial support to make this project possible. I want to show my gratitude to Francisco Javier Cazorla Almeida, Enrico Mezzetti and Enrique Fernández García for guiding me and giving me the opportunity to do this work related to research in the field of computer architecture, as well as for the resources dedicated to the development of it, including the material, the trips, as well as their own time. I would also like to mention the people who have given me support, in particular to Mikel Fernandez, especially for his help to deal with the setbacks of the most advanced phases of the project. There is no doubt that the work done has given me the opportunity to define and channel the direction I want to give to my professional career.

On the other hand, I would like to thank in a general way all the Research Teaching Staff, members of the Administration and Services Staff and ultimately the colleagues who have reached me out when I needed it. From this last group I would like to highlight my colleague and friend Javier Enrique Barrera Herrera for his eternal and contagious illusion, willingness and support throughout all these years.

I would like to make a special mention to the group of colleagues and friends with whom I have shared so many good times and trips throughout the whole Degree, to whom I would like to remind that they have my full availability and friendship indefinitely.

Last but not least, I would like to express my deepest gratitude for the unconditional and unselfish support I have received from my family and partner throughout the Degree, for having been able to understand my limited availability to them especially in the last moments of it and for having made of me the person that I am today.

Index

Summary

Application domain and scope of the project

This project is framed in the fields of computer architecture, embedded real-time systems, and computer technology. It has been developed in collaboration with the Computer Architecture Operating System (CAOS) group at the Barcelona Supercomputing Center (BSC) as part of an existing research line of the group at issue, focused on the analysis of the effects of inter-core interference in multicore systems. Within this same research line, the group has been proposing different analytical models, for different hardware platforms, that relate Performance Monitoring Counters (PMCs) to the upper bound to the timing interference incurred by a program owing to conflicts in the use of shared hardware resources (e.g., main memory, bus, etc.). Devising an effective and industrially amenable approach for capturing timing interference is a major concern in critical embedded real-time systems where timely execution is as important as functional correctness. This project has been devised as a first step to lay the foundations for a methodology for the use of PMCs to support timing analysis of embedded processors, with particular interest on those used in automotive engineering, for which a representative platform (Infineon AURIX TC27x) was considered and analyzed. The PMC support available on the selected automotive board has been analyzed both theoretically and empirically in order to support the configuration and low-level manipulation of PMCs, creating a high-level application programming interface (API) that instantiates, for the first time, the high-performance Performance Application Programming Interface (PAPI) into the embedded domain.

Motivation and Objectives

Performance monitoring counters are extensively present in the debug support unit (DSU) of most modern hardware platforms. PMCs are normally used for early functional verification on the hardware design and for average performance analysis and profiling. PMCs have been normally exploited, for example, in the optimization of high-performance computing platforms and applications. In order to relieve the user from low-level, error-prone manipulation of PMC registers, several efforts have been put for the

definition of a high-level interface to the PMCs. The **P**erformance **A**pplication **P**rogramming **I**nterface (PAPI) is a project originally developed at the University of Tennessee's Innovative Computing Laboratory in the Computer Science Department that aimed at designing, standardizing, and implementing a portable and efficient API (Application Programming Interface) to access the hardware performance counters found on most modern microprocessors. However, the scope of PAPI (i.e., its support) is currently limited to a relatively short list of mainstream microprocessors and generally, from the high-performance domain. So far, no PAPI or equivalent implementation has been carried out on embedded processors, where instead we identify a strong need for a standardized interface to the PMC layer, to support the use of PMCs for timing analysis. This led us to consider the need to develop "ePAPI" (embedded Performance Application Programming Interface) which is the name that the library receives in reason of explicitly targeting the need of the embedded domain, and to suggest a new way of exploiting PMCs in the analysis of embedded real-time systems. In defining ePAPI we tried to understand and capture the requirements from the timing analysis of embedded systems. We also assess how the "generic" PAPI standpoint may or may not cover those requirements.

At the end of this project, we became familiar with a current and representative platform of the automotive domain. We have as well achieved a deep knowledge of the PMC support specially in relation with the PAPI organization. We were able to check the functioning of the PMC library and work scientifically to obtain verifiable results. In addition, the ability to present and explain the different techniques, evaluations and results with clarity and scientific rigor has been acquired.

Summary of the chapters

This section summarizes the how this work is structured in chapters, showing a preview of what is involved in each of them. Subsequently, we will highlight the justification of the specific competences, of the computer engineering specialization, that are covered in this project.

The first chapter introduces the reader to the problem of devising an effective and efficient analysis of the timing behavior of embedded real-time systems; some background information is given on the state of the art as far as timing analysis is concerned and the current methods of timing analysis. Following these points, the

limitations of the current methods are highlighted, and the PMCs are proposed as a means that assists and improve the existing methods. We introduce idea of a standardized API for PMCs for embedded systems, and hint at ePAPI as our original proposal. In addition, in the first chapter, we describe the project environment and set-up, which includes the software and hardware resources used to develop the project.

In the second chapter the Performance Application Programming Interface is introduced. Its architecture is described as well as the basic types on which it relies for its operation also known as Events and how to group them.

The chapter three, focuses in the PMC support of the AURIX TC275. In this chapter, the study related to the mapping of events between the board and the PAPI interface is carried out with a high level of detail.

In the fourth chapter, the design and implementation of ePAPI is disclosed. The design requirements are brought to discussion and the identification of a relevant subset of PAPI is described. Then, the events and counter interface including the high-level and low-level APIs are exposed together with the differences between ePAPI and the standard PAPI.

In the fifth chapter, the developed library is brought to verification and validation. First of all, we describe the methodology we followed, the test environment and the followed procedure. Then, we report on the verification of the library, followed by the validation.

Finally, the last chapter deals with the main contributions as well as the fulfilled objectives A critical assessment closes the document, providing a quick outlook on future directions.

Specific competences

This project covers, in addition to the common competences of the final project (TFG01), the following competences:

- IC03: Ability to analyze and evaluate computer architectures, including parallel and distributed platforms, as well as to develop and optimize software for them.
- IC05: Ability to analyze, evaluate and select the most suitable hardware and software platforms to support embedded and real-time applications.

- IC06: Ability to understand, apply and manage the guarantee and security of computer systems.
- IC07: Ability to analyze, evaluate, select and configure hardware platforms for the development and execution of applications and computer services.

Chapter 1

Introduction

1.1 Problem definition

The market of real-time embedded systems (CRTES, for its acronym: Critical Real-Time Embedded Systems), which includes aircraft systems/avionics, space, railways and automotive domains, is experimenting an unprecedented growth and it is expected to continue growing in the future. When we speak about CRTES it is mandatory to speak about the necessity to guarantee the execution times. This leads to the need of carrying out a study that gives us the worst-case execution time (WCET, for its acronym: Worst-Case Execution Time).

1.2 State of the art on timing analysis

In the state of art there are two well-differentiated approaches in what time analysis refers to: Static Timing Analysis and Measurement-Based Timing Analysis.

1.2.1 Static Timing Analysis (STA)

The static analysis consists of the construction of a system model and a mathematical representation of the application from which it is possible to derive the temporal behavior of the application, without the need to execute it. The mathematical representation is processed to determine the upper bound of the WCET, obtaining a formally proved results. The main limitation of this method is the large amount of information that must be taken into account to carry out the analysis, mainly information about the status or history of execution of the system. On top of this, the complexity added by certain architectures, in which processors under intellectual property restrictions with incomplete and / or reserved documentation, makes it more difficult, if at all possible, to build an accurate and sound model for static analysis. In these cases, the alternative is to perform an analysis based on measurements, which is still largely the most common method in industry.

1.2.2 Measurement-Based Timing Analysis (MBTA)

Measurement-based timing analysis consists in carrying out tests in an intensively way trying to cover the input data domain, with the aim of establishing a high-water mark (HWM). This high-water mark symbolizes the WCET, the longest observed execution, plus a margin of safety. Even so, it is quite difficult to ensure the behavior that a system

will have, especially in architectures with cache memories or other type of elements that introduce temporary complexity and unpredictability.

1.2.3 Limitations

As already highlighted, the validation of the execution time is a critical step when designing and using real time systems for control systems. As the time it takes to execute a task or a program depends on many factors, to be able to execute it under strong timing guarantees, it is necessary to know the most pessimistic case or Worst-Case Execution Time (WCET), which is usually obtained via a WCET analysis approach. One of the lines of research developed by the BSC focuses on the techniques and methodologies of WCET calculation, with special emphasis on multithreaded systems.

The growing demand in CRTES for increased performance impels us to use complex high-performance systems, with multiple levels of cache, and multiple cores in one chip. In this scenario, the WCET calculation approaches based on Static Timing Analysis (STA, for its acronym: Static Timing Analysis) are becoming obsolete. This is because they face significant challenges due to the excessive cost of getting the detailed knowledge of the internal operations and the state of the system, which makes it difficult to compute reliable and tight execution time bounds. This difficulty leads in most cases to the provision of extremely pessimistic bounds. All these concerns on the analyzability of the timing behavior of multicores are compromising their adoption in critical systems.

Measurement-Based Timing Analysis (hereinafter MBTA) have traditionally been considered as an industrially viable alternative for STA. However, end-to-end measurements, aimed at obtaining the High-Water Mark, hardly provide the necessary reliable guarantees required by the certification authorities of each critical application domain, especially in the increasingly complex hardware and software systems.

More advanced ways of carrying out MBTA can be devised by capturing additional information about the execution of the program. The Performance Monitoring Counters (PMC), that are usually available in the modern Debug Support Units, can be used to collect detailed information about the behavior of the software and hardware during execution, even in relation to time measurements. Unfortunately, PMC support varies greatly through hardware platforms, and tracked events may even vary between models of the same platform. This diversity complicates the definition of an analysis framework of generic time analysis (and process) based on PMCs.

1.3 Performance Monitoring Counters

All modern processors support some form of performance monitoring counters. Although originally implemented for debugging hardware designs during late hardware development pahses, they have come to be used extensively for performance analysis and

for validating tools and simulators. The types and numbers of events tracked and the methodologies for using these performance counters vary widely, not only across architectures, but also across systems sharing the same ISA. For example, the Pentium III tracks 80 different events, measuring only two at a time, but the Pentium 4 tracks 48 different events, measuring up to 18 at a time. Chips manufactured by different companies have even more divergent counter architectures: for instance, AMD and Intel implementations have little in common, despite the support the same ISA. Verifying that measurements generate meaningful results across arrays of implementations is essential to using counters for research.

1.3.1 PMCs support for Timing analysis

As already highlighted, hardware counters exist on every major processor today. These counters can be used to collect detailed information about the behavior of software and hardware during execution, even in relation to time measurements. Thus, they provide performance analysts with a basis for tool development, and application developers with valuable information about those sections of their code that can be improved to allow better performance. However, there are only a few APIs that allow accessing these counters, and many of them are poorly documented, unstable, or unavailable. In addition, performance metrics may have different definitions and different programming interfaces on different platforms.

1.4 Need for standardized API for PMCs

As described above, one of the objectives of this project is to investigate the possibility of implementing a common abstract PMC library to enable the collection, in a platform-independent way, of those hardware events that are relevant from the multicore contention analysis perspective. The PAPI interface and tools identify a de facto standard for the configuration and collection of hardware events on mainstream hardware devices. For this reason, we selected PAPI as a term of reference and investigate whether the generic (platform-independent) events singled out by the interface are relevant to multicore contention analysis. We were also interested in understanding whether the subset of relevant events can be effectively supported by the set of PMCs available on a typical platform from the real-time embedded domain. We focused on the support available on the AURIX TC-27x family of processor, a reference platform in the automotive domain.

1.5 Project environment

1.5.1 Software resources

This section describes the basic software components, such as the development environment, the debug engine, as well as the utilities and tools used in the project.

1.5.1.1 Aspects to consider

Starting from the fact that the AURIX TriCore platform was located in the BSC, it is concluded that remote access was necessary. To do so, the MobaXterm program was selected in order to connect from a terminal to a remote served called bsc-caos-gw-bsc.es from which it could be launched **rdesktop** instances to connect to the computer where the AURIX TriCore platform was connected.

1.5.1.2 Cygwin and Tricore-gcc

In the computer at issue there was a C/C++ development environment called CygWin, which is a collection of tools developed by Cygnus Solutions that provide a similar behavior to Unix systems on top of a Microsoft Windows system. Its goal is to port software that runs on POSIX systems to Windows by recompiling its sources.

The compiler that was used to generate the binaries that were going to be launched on top of the platform is the gcc porting to the AURIX TriCore architecture and ISA, called tricore-gcc provided by HighTec GmbH.

1.5.1.3 Universal Debug Engine UDE and Microcontroller Debugger for AURIX TriCore

The Universal Debug Engine (UDE) is the debug interface used in the project. The UDE was used to upload the binaries to the platform,, run them and obtain the results. It offers a wide range of development support solutions for software development of systems-on-silicon including debug support. Its features are:

- Microcontroller debug support.
- Multi-core Debugging (MCA) support
- Multi-core Debug Solution (MCDS) support by Universal Emulation Configurator (UEC)

- FLASH Memory Programming.
- Aurora Gigabit Trace (AGBT) support
- JTAG / MiniDAP / cJTAG / MiniJTAG / ETKS support
- Device Access Port (DAP / DAP2) with up to 160 MHz serial clock support
- Hardware Security Module (HSM) support
- Profiling support
- Code coverage ISO 26262 support
- Eclipse Plug-in
- UDE Starterkits
- PXROS-HR with Memory Protection
- CAN Loader, CAN Recorder and CANopen® Message Formatter
- CIF Video Trace support
- Generic Timer Module (GTM) support
- IP Snooping Trace support
- Triggered Transfer (TTF) support
- On-Chip Debug Support JTAG (OCDS L1/L2) for Core, PCP, PCP2, DMA

1.6 Hardware resources

Throughout the development of this project it has been guaranteed remote access to the selected hardware target, on which there has been the possibility of testing and analyzing the library that was being developed. The selected board on which the library has been developed was the AURIX TriCore TC275 which is a representative platform of the automotive domain.

1.6.1 The AURIX TriCore TC275

The TC275 combines three powerful technologies within one silicon die, achieving high levels of power, speed, and economy for embedded applications:

- Reduced Instruction Set Computing (RISC) processor architecture.
- Digital Signal Processing (DSP) operations and addressing modes.
- On-chip memories and peripherals.

DSP operations and addressing modes provide the computational power necessary to efficiently analyze complex real-world signals. The RISC load/store architecture provides high computational bandwidth with low system cost. On-chip memory and peripherals are designed to support even the most demanding high-bandwidth real-time embedded control-systems tasks.

Additional high-level features of the TC27x include:

- Efficient memory organization: instruction and data scratch memories, caches
- Serial communication interfaces – flexible synchronous and asynchronous modes
- Multiple channel DMA Controller – DMA operations and interrupt servicing
- Flexible interrupt system – configurable interrupt priorities and targets
- Hardware Security Module
- Flexible CRC Engine
- General-purpose timers
- High-performance on-chip buses
- On-chip debugging and emulation facilities
- Flexible interconnections to external components
- Flexible power-management

The TC27x is a high-performance microcontroller with three TriCore CPUs, program and data memories, buses, bus arbitration, interrupt system, DMA controller and a powerful set of on-chip peripherals. The TC27x is designed to meet the needs of the most demanding embedded control systems applications where the competing issues of price/performance, real-time responsiveness, computational power, data bandwidth, and power consumption are key design elements.

The TC27x offers several versatile on-chip peripheral units such as serial controllers, timer units, and analog-to-digital converters. Within the TC27x, all these peripheral units are connected to the TriCore CPUs / system via the System Peripheral Bus (SPB) and the Local Memory Bus (SRI). A number of I/O lines on the TC27x ports are reserved for these peripheral units to communicate with the external world.

1.6.1.1 TC275 Blocks Diagram

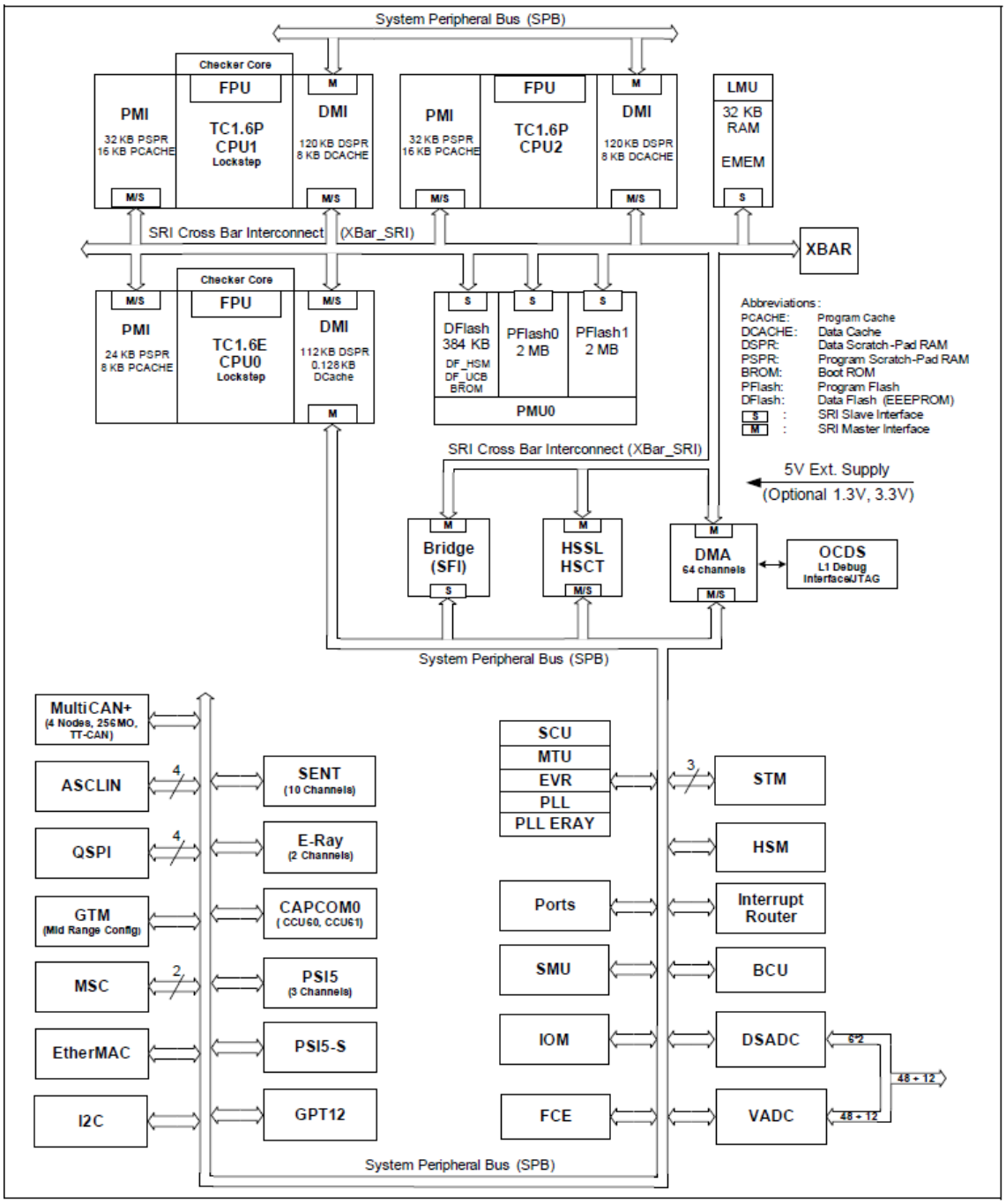

Figure 1: TC27x Blocks Diagram. (*taken from AURIX TC27x B-Step 32-Bit Single-chip Microcontroller User's Manual V 1.4.1 2014-02. Infineon Technologies AG. Page 78.)*

1.6.1.2 CPU Cores of the TC275 (CPU)

The TC275 includes two high performance TriCore TC1.6P cores and one high efficiency TriCore TC1.6E CPU Core. All processors implement the same version (V1.6) of the TriCore Processor Architecture.

The TriCore TC1.6P and TC1.6E CPUs provide the following features:

TriCore Architectural Highlights

- Unified RISC MCU/DSP
- 32-bit architecture with 4Gbytes unified data, program, and I/O address space
- 32 general purpose registers with fast automatic context-switching
- Multiply-accumulate unit able to sustain 2 MAC operations per cycle
- Fully pipelined Floating point unit
- Saturating integer arithmetic
- Bit handling
- Packed data operations
- Dedicated integer divide unit.
- Precise exceptions
- Flexible power management
- Flexible memory protection system

High-efficiency TriCore Instruction Set

- Powerful instruction set
- Binary compatibility between TC16P and TC16E
- Freely mixable 16-bit and 32-bit instructions for reduced code size
- Data types include: Boolean, array of bits, character, signed and unsigned integer, integer with saturation, signed fraction, double-word integers, and IEEE-754 single-precision floating point
- Data formats include: Bit, 8-bit byte, 16-bit half-word, 32-bit word, and 64-bit doubleword
- Flexible and efficient addressing mode for high performance and code density
- IEEE-754 compatible floating-point unit.
    - Single precision (denormalized numbers not supported for arithmetic operations).
    - Traps optionally generated on floating point exceptions.
    - Selectable rounding mode.

TriCore 1.6P CPU

- High performance superscalar CPU executing up to three instructions per cycle
- Optimized for high throughput computation
- Zero overhead loop

- Instruction memory:
    - 32Kbyte single cycle Program Scratch-Pad RAM (PSPR)
    - 16Kbyte single cycle Program Cache (PCACHE)
- Data memory:
    - 120Kbyte single cycle data Scratch-Pad RAM (DSPR)
    - 8Kbyte single cycle data Cache (DCACHE)
- All memories are ECC protected
- Cacheability of memory regions is programmable.
- FPU

TriCore 1.6E CPU

- High efficiency scalar CPU executing a maximum of one instruction per cycle
- Optimized for low power operation
- Optimized peripheral connectivity
- Instruction memory:
    - 24Kbyte single cycle Program Scratch-Pad RAM (PSPR)
    - 8Kbyte single cycle Program Cache (PCACHE)
- Data memory:
    - 112Kbyte single cycle data Scratch-Pad RAM (DSPR)
    - Data read buffer
- All memories are ECC protected
- Cacheability of all memory regions is programmable.
- FPU

Chapter 2

PAPI: Performance Application Programming Interface

2.1 Introduction to PAPI

2.1.1 What is PAPI?

PAPI is an acronym for **P**erformance **A**pplication **P**rogramming **I**nterface. The PAPI Project is being developed at the University of Tennessee's Innovative Computing Laboratory in the Computer Science Department. This project was created to design, standardize, and implement a portable and efficient API (Application Programming Interface) to access the hardware performance counters found on most modern microprocessors.

2.1.2 Architecture

The figure below shows the internal design of the PAPI architecture. In this figure, we can see the two layers of the architecture.

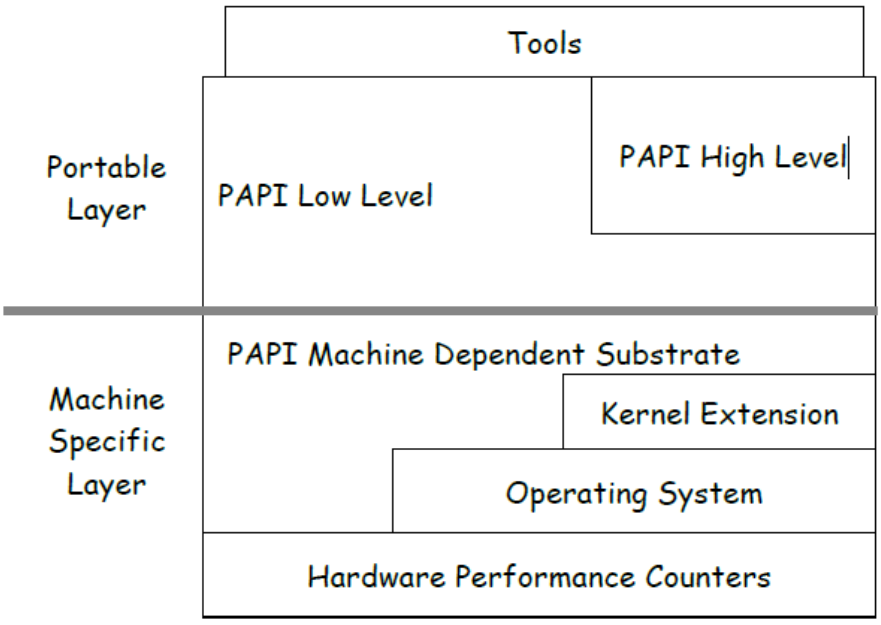

Figure 2: Layers of PAPI architecture. (*Taken from PAPI USER'S GUIDE Version 3.5.0*)

The **Portable Layer** consists of the API (low level and high level) and machine independent support functions.

The **Machine Specific Layer** defines and exports a machine independent interface to machine dependent functions and data structures. These functions are defined in the substrate layer, which uses kernel extensions, operating system calls, or assembly language to access the hardware performance counters. PAPI uses the most efficient and flexible of the three, depending on what is available.

PAPI strives to provide a uniform environment across platforms. However, this is not always possible. Where hardware support for features, such as overflows and multiplexing is not supported, PAPI implements the features in software where possible. Also, processors do not support the same metrics, thus you can monitor different events depending on the processor in use. Therefore, the interface remains constant, but how it is implemented can vary.

2.2 Events

2.2.1 What are events?

Events are occurrences of specific signals related to a processor's function. Hardware performance counters are typically implemented as a small set of registers that count events, such as cache misses and floating-point operations while the program executes on the processor. Monitoring these events facilitates correlation between the structure of source/object code and the efficiency of the mapping of events that are *native* to that architecture. PAPI provides a software abstraction of these architecture-dependent native events, as a collection of *preset* events that are accessible through the PAPI interface.

2.2.2 Native Events

Native events compromise the set of all events that are countable by the CPU. There are generally far more native events available than can be mapped onto PAPI preset events. Even if no preset event is available that exposes a given native event, native events can still be accessed directly. To use native events effectively you should be very familiar with the particular platform in use. PAPI provides access to native events on all supported platforms through the low-level interface. Native events use the same interface as used when setting up a preset event, but since a PAPI preset event definition is not available for native events, a native event name must often be translated into an event code before it can be used.

2.2.3 Preset Events

Preset events, also known as predefined events, are a common set of events deemed relevant and useful for application performance tuning. These events are typically found in many CPUs that provide performance counters and give access to the memory hierarchy, cache coherence protocol events, cycle and instruction counts, functional unit, and pipeline status. Furthermore, preset events are mappings from symbolic names (PAPI preset name) to machine specific definitions (native countable events) for a particular hardware resource. For example, Total Cycles (in user mode) is PAPI_TOT_CYC. Also, PAPI supports presets that may be derived from the underlying hardware metrics. For example, Total L1 Cache Misses (PAPI_L1_TCM) might be the sum of L1 Data Misses and L1 Instruction Misses on a given platform. A preset can be either directly available as a single counter, derived using a combination of counters, or unavailable on any particular platform.

The PAPI library names approximately 100 preset events, which are defined in the header file, **papiStfEventDefs.h**. For a given platform, a subset of these preset events can be counted through either a simple high-level programming interface or a more complete C or Fortran low-level interface. Note that processors and software are revised over time.

The exact semantics of an event counter are platform dependent. PAPI preset names are mapped onto available events so as to map as many countable events as possible on different platforms. Due to hardware implementation differences, it is not necessarily feasible to directly compare the counts of a particular PAPI preset events obtained on different hardware platforms.

2.3 Event Sets

Event Sets are user-defined groups of hardware events (preset or native), which are used in conjunction with one another to provide meaningful information. The user specifies the events to be added to an Event Set, and other attributes, such as: the counting domain (user or kernel), whether or not the events in the Event Set are to be multiplexed, and whether the Event Set is to be used for overflow or profiling. Other settings for the Event Set are maintained by PAPI, such as: what low-level hardware registers to use, the most recently read counter values, and the state of the Event Set (running/not running). Event Sets provide an effective abstraction for the organization of information associated with counting hardware events. The PAPI library manages the memory for Event Sets with a user interface through integer handles to simplify calling conventions. The user is free to allocate and use any number of them provided the substrate can provide the required resources. Only one Event Set can be in active use at any time in a given thread or process.

Chapter 3

AURIX TC275 Performance Monitoring Counters support

3.1 AURIX TC275 PMU

The AURIX TriCore TC275 includes a CPU Core Debug system that includes five Performance Monitor Counters. Of the five PMCs, three are multiplexed and two have a fixed configuration.

The performance counter sources from the AURIX TC275 hereafter, events (to simulate the PAPI vocabulary), are a total of fourteen and are described below:

CCNT CPU Clock Count Register.
ICNT Instruction Count Register.
IP_DISPATCH_STALL The counter is incremented on every cycle in which the Integer dispatch unit is stalled for whatever reason.
LS_DISPATCH_STALL The counter is incremented on every cycle in which the Load-Store dispatch unit is stalled for whatever reason.
LP_DISPATCH_STALL The counter is incremented on every cycle in which the Loop dispatch unit is stalled for whatever reason.
PCACHE_HIT The counter is incremented whenever the target of a cached fetch request from the fetch unit is found in the program cache.
PCACHE_MISS The counter is 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.
MULTI_ISSUE The counter is incremented in any cycle where more than one instruction is issued.
DCACHE_HIT

The counter is incremented whenever the target of a cached request from the Load-Store unit is found in the data cache.
DRB_HIT The counter is incremented whenever the target of a cached request from the Load-Store unit is found in the data read buffer. (**Only on energy efficient**).
DCACHE_MISS_CLEAN The counter is 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.
DRB_MISS The counter is incremented whenever the target of a cached request from the Load-Store unit is not found in the data read buffer and hence a bus fetch is initiated. (**Only on energy efficient**).
DCACHE_MISS_DIRTY The counter is 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 writeback of a dirty cache line.
TOTAL_BRANCH The counter is incremented in any cycle in which a branch instruction is in a branch resolution stage of the pipeline (IP_EX1, LS_DEC, LP_DEC).
PMEM_STALL The counter is incremented whenever the fetch unit is requesting an instruction and the Instruction memory is stalled for whatever reason.
DMEM_STALL The counter is incremented whenever the Load-Store unit is requesting a data operation and the data memory is stalled for whatever reason.

Except for the events identified by the names CCNT and ICNT, the events are obtained via the multiplexed counters as described below:

TC1.6P (Performance efficient)			
M1CNT COUNTER1	M2CNT COUNTER2	M3CNT COUNTER3	Not multiplexed Counters
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	

As the scope of this study only considers the *Performance Efficient* cores (TC1.6P), events related to the *Power Efficient cores* (TC1.6E) like the ones related to the DRB (Data Read Buffer) won't be considered.

3.2 Mapping AURIX events to PAPI interface

As highlighted since the beginning of the document, we have taken PAPI as a term of reference to investigate whether the generic (platform-independent) singled out by the interface are relevant to multicore contention analysis.

The following table summarizes the results of our analysis of whether and how current AURIX performance monitoring counters can support those PAPI events that are relevant from the standpoint of multicore contention analysis. In consideration of the fact that the AURIX exhibits a flat memory hierarchy (only one cache level), all PMCs tracking cache events (hit and misses) are considered to be relative to L1 Caches.

The cell colors follow a precise color code. A first color convention is used to identify the AURIX performance monitoring counter(s) that can be used to cover a specific PAPI event: blue stands for the M1CNT (COUNTER 1), yellow for the M2CNT (COUNTER 2), and green for the M3CNT (COUNTER 3). However, some PAPI events can only be covered by combining the information from more than one PMC. If the events are intercepted by activating more than one counter "at the same time", the cell is colored in purple. Finally, black will be used when the sub-events needed to cover a PAPI event are constrained to be collected by the same multiplexed counter in the AURIX. In this case more than one execution will be required to collect all necessary evidence from the PMCs.

	Events captured by the M1CNT (COUNTER 1)
	Events captured by the M2CNT (COUNTER 2)
	Events captured by the M3CNT (COUNTER 3)
	Events captured by more than one counter
	Events captured by more events tracked by the same counter (more than one run needed)

A second color convention is used to assess relevance of a PAPI event (multicore contention analysis) and implementability on the AURIX. Events that are not relevant for multicore contention analysis (they can be relevant for timing analysis in general, performance analysis, etc.) Rows of non-relevant events are made gray; relevant but not implementable on the AURIX are colored in red; possibly relevant events that are not so in the specific board, are colored in orange.

	Relevant but not captured in the AURIX
	Relevant in general but not in the case of the AURIX (e.g., L3 cache performance)
	Not supported but not relevant

The following analysis table uses a series of acronyms that are described below:

[1] Timing and/or Energy Consumption Analysis.

[2] Characterizing Multicore Contention

[3] Average Performance Analysis

[4] Contention analysis

PAPI EVENT	*DESCRIPTION*	*AURIX PMU support*	*Counter Source(s)*	*Counter Source Detail*	*Relevant for multicore contention analysis?*
PAPI_BR_CN	Conditional branch instructions	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_BR_INS	Branch instructions	**Y (single counter)**	**TOTAL_BRANCH**	The counter is incremented in any cycle in which a branch instruction is in a branch resolution stage of the pipeline (IP_EX1, LS_DEC, LP_DEC).	Relevant for TECA[1] but not directly for CMC[2].
PAPI_BR_MSP	Conditional branch instructions mispredicted	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_BR_NTK	Conditional branch instructions not taken	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_BR_PRC	Conditional branch instructions correctly predicted	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_BR_TKN	Conditional branch instructions taken	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].

PAPI EVENT	*DESCRIPTION*	*AURIX PMU support*	*Counter Source(s)*	*Counter Source Detail*	*Relevant for multicore contention analysis?*
PAPI_BR_UCN	Unconditional branch instructions	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_BRU_IDL	Cycles branch units are idle	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_BTAC_M	Branch target address cache misses	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_CA_CLN	Requests for exclusive access to clean cache line	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_CA_INV	Requests for cache line invalidation	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_CA_ITV	Requests for cache line intervention	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_CA_SHR	Requests for exclusive access to shared cache line	**N**		--	Relevant for TECA[1] but not directly for CMC[2].
PAPI_CA_SNP	Requests for a snoop	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_CSR_FAL	Failed store conditional instructions	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_CSR_SUC	Successful store conditional instructions	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_CSR_TOT	Total store conditional instructions	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_FAD_INS	Floating point add instructions	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_FDV_INS	Floating point divide instructions	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_FMA_INS	FMA instructions completed	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_FML_INS	Floating point multiply instructions	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_FNV_INS	Floating point inverse instructions	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].

PAPI EVENT	*DESCRIPTION*	*AURIX PMU support*	*Counter Source(s)*	*Counter Source Detail*	*Relevant for multicore contention analysis?*
PAPI_FP_INS	Floating point instructions	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_FP_OPS	Floating point operations	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].contention
PAPI_FP_STAL	Cycles the FP unit(s) are stalled	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_FPU_IDL	Cycles floating point units are idle	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_FSQ_INS	Floating point square root instructions	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_FUL_CCY	Cycles with maximum instructions completed	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_FUL_ICY	Cycles with maximum instruction issue	**N**	-	Note: The MULTI_ISSUE counter in the AURIX counts cycles with multiple instructions (not necessarily maximum number thereof)	Relevant for TECA[1] but not directly for CMC[2].
PAPI_FXU_IDL	Cycles integer units are idle	**N**	**-**	-	Relevant (if can be reconducted to activity on the interconnect)
PAPI_HW_INT	Hardware interrupts	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_INT_INS	Integer instructions	**N**	-	-	Relevant for APA[3]
PAPI_L1_DCA	Level 1 data cache accesses	**Y (compound)**	**DCACHE_HIT + DCACHE_MISS_CLEAN + DCACHE_MISS_DIRTY**	Descriptions found in the details box.	**Relevant**
PAPI_L1_DCH	Level 1 data cache hits	**Y (single counter)**	**DCACHE_HIT**	The counter is incremented whenever the target of a cached request from the Load- Store unit is found in the data cache.	**Relevant**

PAPI EVENT	*DESCRIPTION*	*AURIX PMU support*	*Counter Source(s)*	*Counter Source Detail*	*Relevant for multicore contention analysis?*
PAPI_L1_DCM	Level 1 data cache misses	**Y (compound)**	**DCACHE_MISS_DIRTY + DCACHE_MISS_CLEAN**	Descriptions found in the details box.	**Relevant**
PAPI_L1_DCR	Level 1 data cache reads	**N**	**Can be loosely upperbounded by the number of data cache accesses**		**Relevant**
PAPI_L1_DCW	Level 1 data cache writes	**N**	**Can be loosely upper-bounded by the number of data cache accesses**		**Relevant**
PAPI_L1_ICA	Level 1 instruction cache accesses	**Y (compound)**	**PCACHE_HIT + PCACHE_MISS**		**Relevant**
PAPI_L1_ICH	Level 1 instruction cache hits	**Y (single counter)**	**PCACHE_HIT**	The counter is incremented whenever the target of a cached fetch request from the fetch unit is found in the program cache.	**Relevant**
PAPI_L1_ICM	Level 1 instruction cache misses	**Y (single counter)**	**PCACHE_MISS**	The counter is 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.	**Relevant**
PAPI_L1_ICR	Level 1 instruction cache reads	**Y (compound)**	**PCACHE_HIT + PCACHE_MISS**	Descriptions found in the details box.	**Relevant**
PAPI_L1_ICW	Level 1 instruction cache writes	**N**	-	-	**Relevant (if writes allowed)**
PAPI_L1_LDM	Level 1 load misses	**N**	-	-	**Relevant**
PAPI_L1_STM	Level 1 store misses	**N**	-	-	**Relevant**
PAPI_L1_TCA	Level 1 total cache accesses	**Y (compound)**	**DCACHE_HIT + DCACHE_MISS_CLEAN + DCACHE_MISS_DIRTY + PCACHE_HIT + PCACHE_MISS**		**Relevant**

PAPI EVENT	*DESCRIPTION*	*AURIX PMU support*	*Counter Source(s)*	*Counter Source Detail*	*Relevant for multicore contention analysis?*
PAPI_L1_TCH	Level 1 total cache hits	**Y (compound)**	**DCACHE_HIT + PCACHE_HIT**		**Relevant**
PAPI_L1_TCM	Level 1 cache misses	**Y (compound)**	**DCACHE_MISS_DIRTY + DCACHE_MISS_CLEAN + PCACHE_MISS**		**Relevant**
PAPI_L1_TCR	Level 1 total cache reads	**N**	-	-	**Relevant**
PAPI_L1_TCW	Level 1 total cache writes	**N**	-	-	**Relevant**
PAPI_L2_DCA	Level 2 data cache accesses	**N**	-	-	Relevant (if L2 available)
PAPI_L2_DCH	Level 2 data cache hits	**N**	-	-	Relevant (if L2 available)
PAPI_L2_DCM	Level 2 data cache misses	**N**	-	-	Relevant (if L2 available)
PAPI_L2_DCR	Level 2 data cache reads	**N**	-	-	Relevant (if L2 available)
PAPI_L2_DCW	Level 2 data cache writes	**N**	-	-	Relevant (if L2 available)
PAPI_L2_ICA	Level 2 instruction cache accesses	**N**	-	-	Relevant (if L2 available)
PAPI_L2_ICH	Level 2 instruction cache hits	**N**	-	-	Relevant (if L2 available)
PAPI_L2_ICM	Level 2 instruction cache misses	**N**	-	-	Relevant (if L2 available)
PAPI_L2_ICR	Level 2 instruction cache reads	**N**	-	-	Relevant (if L2 available)
PAPI_L2_ICW	Level 2 instruction cache writes	**N**	-	-	Relevant (if L2 available)
PAPI_L2_LDM	Level 2 load misses	**N**	-	-	Relevant (if L2 available)
PAPI_L2_STM	Level 2 store misses	**N**	-	-	Relevant (if L2 available)
PAPI_L2_TCA	Level 2 total cache accesses	**N**	-	-	Relevant (if L2 available)
PAPI_L2_TCH	Level 2 total cache hits	**N**	-	-	Relevant (if L2 available)
PAPI_L2_TCM	Level 2 cache misses	**N**	-	-	Relevant (if L2 available)
PAPI_L2_TCR	Level 2 total cache reads	**N**	-	-	Relevant (if L2 available)

PAPI EVENT	*DESCRIPTION*	*AURIX PMU support*	*Counter Source(s)*	*Counter Source Detail*	*Relevant for multicore contention analysis?*
PAPI_L2_TCW	Level 2 total cache writes	**N**	-	-	Relevant (if L2 available)
PAPI_L3_DCA	Level 3 data cache accesses	**N**	-	-	Relevant (if L3 available)
PAPI_L3_DCH	Level 3 data cache hits	**N**	-	-	Relevant (if L3 available)
PAPI_L3_DCM	Level 3 data cache misses	**N**	-	-	Relevant (if L3 available)
PAPI_L3_DCR	Level 3 data cache reads	**N**	-	-	Relevant (if L3 available)
PAPI_L3_DCW	Level 3 data cache writes	**N**	-	-	Relevant (if L3 available)
PAPI_L3_ICA	Level 3 instruction cache accesses	**N**	-	-	Relevant (if L3 available)
PAPI_L3_ICH	Level 3 instruction cache hits	**N**	-	-	Relevant (if L3 available)
PAPI_L3_ICM	Level 3 instruction cache misses	**N**	-	-	Relevant (if L3 available)
PAPI_L3_ICR	Level 3 instruction cache reads	**N**	-	-	Relevant (if L3 available)
PAPI_L3_ICW	Level 3 instruction cache writes	**N**	-	-	Relevant (if L3 available)
PAPI_L3_LDM	Level 3 load misses	**N**	-	-	Relevant (if L3 available)
PAPI_L3_STM	Level 3 store misses	**N**	-	-	Relevant (if L3 available)
PAPI_L3_TCA	Level 3 total cache accesses	**N**	-	-	Relevant (if L3 available)
PAPI_L3_TCH	Level 3 total cache hits	**N**	-	-	Relevant (if L3 available)
PAPI_L3_TCM	Level 3 cache misses	**N**	-	-	Relevant (if L3 available)
PAPI_L3_TCR	Level 3 total cache reads	**N**	-	-	Relevant (if L3 available)
PAPI_L3_TCW	Level 3 total cache writes	**N**	-	-	Relevant (if L3 available)
PAPI_LD_INS	Load instructions	**N**	-	-	Relevant for APA[3]
PAPI_LST_INS	Load/store instructions completed	**N**	-	-	Relevant for APA[3]
PAPI_LSU_IDL	Cycles load/store units are idle	**N**			**Relevant**
PAPI_MEM_RCY	Cycles Stalled Waiting for memory Reads	**N**			**Relevant**

PAPI EVENT	*DESCRIPTION*	*AURIX PMU support*	*Counter Source(s)*	*Counter Source Detail*	*Relevant for multicore contention analysis?*
PAPI_MEM_SCY	Cycles Stalled Waiting for memory accesses	**Y (compound)**	**DMEM_STALL + PMEM_STALL**	Descriptions found in the details box.	**Relevant**
PAPI_MEM_WCY	Cycles Stalled Waiting for memory writes	**N**	Loosely upper-bounded by DMEM_STALL		**Relevant**
PAPI_PRF_DM	Data prefetch cache misses	**N**	-	-	Relevant for APA[3]
PAPI_RES_STL	Cycles stalled on any resource	**N**	**Can be loosely upper-bounded by summing up the following counters IP_DISPATCH_STALL + LS_DISPATCH_STALL + LP_DISPATCH_STALL + PMEM_STALL + DMEM_STALL (they are counting partially overlapping events)**	-	**Relevant**
PAPI_SR_INS	Store instructions	**N**	-	-	**Relevant**
PAPI_STL_CCY	Cycles with no instructions completed	**N**	-	-	Relevant for APA[3] Might be relevant for CA[4](if can be reconducted to activity on the interconnect)
PAPI_STL_ICY	Cycles with no instruction issue	**N**	-	-	Relevant for APA[3] Might be relevant for CA[4](if can be reconducted to activity on the interconnect)
PAPI_SYC_INS	Synchronization instructions completed	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_TLB_DM	Data translation lookaside buffer misses	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_TLB_IM	Instruction translation lookaside buffer misses	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_TLB_SD	Translation lookaside buffer shootdowns	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].

PAPI EVENT	*DESCRIPTION*	*AURIX PMU support*	*Counter Source(s)*	*Counter Source Detail*	*Relevant for multicore contention analysis?*
PAPI_TLB_TL	Total translation lookaside buffer misses	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].
PAPI_TOT_CYC	Total cycles	**Y (single counter)**	**CCNT**	CPU Clock Count Register	**Relevant** (fundamental for timing analysis in general)
PAPI_TOT_IIS	Instructions issued	**N**			Relevant for TECA[1] but not directly for CMC[2].
PAPI_TOT_INS	Instructions completed	**Y (single counter)**	**ICNT**	Instruction Count Register	Relevant for TECA[1] but not directly for CMC[2].
PAPI_VEC_INS	Vector/SIMD instructions	**N**	-	-	Relevant for TECA[1] but not directly for CMC[2].

Table 1: AURIX events to PAPI interface analysis.

As it can be concluded in the information found in the tables, an in-depth study has been carried out on the compatibility of the events that the AURIX is able to monitor and those collected in PAPI. While some events were easily mapped to the PAPI specification, there was a subset of events that were rated as candidates to be upper bounded. Finally, we decided not to add upperbounding events as we wanted to provide ePAPI only with exact accuracy in the counts.

On the other hand, keeping in mind the idea of building a standardized library for embedded systems, we made a study about the relevance of the events. In the study at issue an explanation about whether or not an event was relevant was made for each one. This helped us to divide the events in a series of groups. For example, the events related to the different cache memory levels were considered relevant but not in the case of the AURIX because of its topology that as we were able to see in the boards specification, only had L1 caches.

Regarding the events captured by more events tracked by the same counter (those that needed more than one run), the method designed to obtain them seeks to be the least invasive and consists of carrying out the different measurements that make up the event at issue separately to then treat them and obtain the result. For example, PAPI_L1_TCA would be obtained by measuring first, the data cache misses (DCACHE_MISS_CLEAN and DCACHE_MISS_DIRTY) together with DCACHE_HIT and then measuring both PCACHE_HIT and PCACHE_MISS to end up adding all the results to obtain PAPI_L1_TCA.

3.2.1 Detailed analysis of relevant events

In this section, it is disclosed the specification with a higher degree of detail, those events that are considered the most relevant separating them into three groups, those that are implementable, those that are not and those that are considered relevant but aren't defined in PAPI.

3.2.1.1 Implementable events

PAPI_L1_DCA

This event gives us the total number of L1 data cache accesses. It is relevant for both, execution time and power consumption. A data cache access may result in three different situations with different repercussion:

- A data cache hit: which is a separate-measurable event in this platform.
- A data cache dirty miss
- A data cache clean miss.

PAPI_L1_DCH

This event gives us the total number of L1 data cache hits. A cache hit is a state in which data requested for processing by a component or application is found in the cache memory. It is a faster means of delivering data to the processor, as the cache already contains the requested data. As a logical conclusion of what was pointed out in the event related to the data cache accesses, this event is relevant for both, execution time and power consumption. This event is implementable and can be used by a programmer to measure the performance of a program as far as cache memories are concerned.

PAPI_L1_DCM

This event gives us the total number of L1 data cache misses. A cache miss is a state where the data requested for processing by a component or application is not found in the cache memory. It causes execution delays by requiring the program or application to fetch the data from other cache levels or the main memory. In the AURIX case, cache misses may entail an access to the Flash or SRAM memories. If data caches implement a write-back policy (as opposed to a write-through one) the data miss count includes both clean and dirty misses although each one exhibits different behavior and latency. When the CPU requests an address from memory, a translation is made to get the correspondent

cache line that is supposed to contain the information. If the information that is being looked for isn't found in the specified line, the **dirty bit** is checked.

- Clean miss: if the **dirty bit** is set to zero, the information requested is loaded from the main memory to the cache together with the rest of the line.
- Dirty miss: if the **dirty bit** is set to one, the information contained in the line must be written back to memory before loading the requested information to the cache. Once the write back has finished, the information requested is loaded to the cache together with the rest of the line and the **dirty bit** is set to zero.

As we can see, the delays incurred by the different kind of misses are pretty different, as in the **dirty miss** an extra memory access is made. Despite the difference, this event groups both misses in only one.

PAPI_L1_ICA

This event gives us the total number of L1 instruction cache accesses. It is relevant for both, execution time and power consumption. An instruction cache access may result in two different situations: an instruction cache hit and a miss, which are both separate-implementable events on this platform.

PAPI_L1_ICH

This event gives us the total number of L1 instruction cache hits. A cache hit is a state in which an instruction requested for processing by a component or application is found in the cache memory. It is a faster means of delivering instructions to the processor, as the cache already contains the requested instructions. As a logical conclusion of what was pointed out in the event related to the instruction cache accesses, this event is relevant for both, execution time and power consumption. This event is typically provided by standard PMUs and can be used by a programmer to measure the performance of a program, as far as cache memories are concerned.

PAPI_L1_ICM

This event gives us the total number of L1 instruction cache misses. A cache miss is a state where the instruction requested for processing by a component or application is not found in the cache memory. It causes execution delays by requiring the program or application to fetch the instructions from other cache levels or the main memory as in this case.

PAPI_L1_ICR

This event gives us the total number of reads that have taken place in the L1 instruction cache. This event is relevant only in case the program code can be modified (which is typically not the case). This event when analyzed together with **PAPI_L1_ICH**, **PAPI_L1_ICM**, gives us information about how the cache is doing in terms of hit and misses taking into account the total number of reads. In this platform this event is

equivalent to the total cache accesses as the instruction cache. This event is relevant for both, execution time and power consumption.

PAPI_L1_TCA

This event is the sum of both, **PAPI_L1_ICA** and **PAPI_L1_DCA**, representing the total L1 cache accesses. This event is relevant for many reasons; some examples are the verification of **PAPI_L1_ICA** and **PAPI_L1_DCA** in the early stages of the implementation to check if the values separately are coherent and, once the implementation is finished, to get the total cache accesses if we don't want to perform a deep analysis of accesses.

PAPI_L1_TCH

This event is the sum of both, **PAPI_L1_DCH** and **PAPI_L1_ICH**, representing the total L1 cache hits. This event is relevant for many reasons, some examples are the verification of **PAPI_L1_DCH** and **PAPI_L1_ICH** in the early stages of the implementation to check if the values separately are coherent and, once the implementation is finished, to get the total cache hits if we don't want to perform a deep analysis of them.

PAPI_L1_TCM

This event is the sum of both, **PAPI_L1_DCM** and **PAPI_L1_ICM**, representing the total L1 cache misses. This event is relevant for many reasons; some examples are the verification of **PAPI_L1_DCH** and **PAPI_L1_ICH** in the early stages of the implementation to check if the values separately are coherent and, once the implementation is finished, to get the total cache misses if we don't want to perform a deep analysis of them.

PAPI_MEM_SCY

This event gives us the number of cycles stalled waiting for memory accesses. This value is the sum of both **PAPI_MEM_RCY** and **PAPI_MEM_WCY**. This event is strongly relevant when assessing the interference between cores, and hence, for timing analysis. When a core is accessing to a position, the other cores can't access to that position, leading to stalls and slowing down the program executions. On the downside, the event does not discriminate between reads/writes and the memory device (on which the stall occurs).

PAPI_TOT_CYC

This event gives us the number of total cycles. This measure is very relevant for the timing analysis, if not the most. If used in a correct way, it gives us information on the execution time of a program. This value can be also correlated to other aspects, like power consumption, and is critical in determining the worst-case response time in real-time systems.

3.2.1.2 Non-implementable events

PAPI_L1_DCR

This event gives us the total number of L1 data cache reads. This event when analyzed together with **PAPI_L1_DCH** and **PAPI_L1_DCM**, gives us information about how the cache is doing in terms of hit and misses taking into account the total number of readings. Having separate information on reads and writes is relevant in that each operation may exhibit different latency as well as support different communication mechanisms (e.g., buffers) on the interconnect. As all the events related to cache, this event is relevant for both, execution time and power consumption.

PAPI_L1_DCW

This event gives us the number of L1 data cache writes. This event when analyzed together with **PAPI_L1_DCM**, gives us information about how the cache is doing in terms of allocation and replacement taking into account the size of the cache. Similarly to PAPI_L1_DCR, having separate information on reads and writes is relevant in that each operation may exhibit different latency as well as support different communication mechanisms (e.g., buffers) on the interconnect. As all the events related to cache, this event is relevant for both, execution time and power consumption.

PAPI_L1_LDM

This event gives us the total number of L1 cache load misses. This is referring to when the processor needs to fetch data from main memory, but data does not exist in the cache. So whenever the processor wants some data from the main memory, it checks the cache, and if the data is already loaded you get load-hit and otherwise you get a load-miss. As all the events related to cache, this event is relevant for both, execution time and power consumption.

PAPI_L1_STM

This event gives us the total number of L1 cache store misses. A store-miss is related to when the processor wants to write back the newly calculated data to the main memory. When writing-back the data to the main memory, the processor has to make sure that the content of the cache and main memory are in sync with each other. It can happen with two different policies, *write-through* and *write-back*. So no matter what policy is implemented, you first need to check whether the data is already in the cache so you can store it to cache first (since it's faster), and if the data block you are looking for has been evicted from the cache, you get a store-miss related to that cache. As all the events related to cache, this event is relevant for both, execution time and power consumption.

PAPI_L1_TCR

This event gives us the total number of L1 cache reads. Making use of this event entails obtaining information with a lower level of granularity than measuring the readings to the L1 data cache (**PAPI_L1_ICR**) and instruction cache (**PAPI_L1_DCR**) separately, since this event is the sum from both. As all the events related to cache, this event is relevant for both, execution time and power consumption.

PAPI_L1_TCW

This event gives us the total number of L1 cache writes. Making use of this event entails obtaining information with a lower level of granularity than measuring the writings to the L1 data cache (**PAPI_L1_ICR**) and instruction cache (**PAPI_L1_DCR**) separately, since this event is the sum from both. As all the events related to cache, this event is relevant for both, execution time and power consumption.

PAPI_LSU_IDL

This event gives us the number of cycles when the load/store units are idle. In general, the degree of occupation of the load/store units is relevant, but when analyzing the degree of interference that occur in buses which are shared among several cores, the relevance of the degree of occupation becomes critical. If there are periods without load/store instructions, the only source of interference would be the instruction fetch.

PAPI_MEM_RCY

This event gives us the number of cycles stalled waiting for memory reads. This event is strongly relevant when assessing the interference between cores, and hence, for timing analysis. When a core is reading a position in memory, the other cores cannot write on it, leading to stalls and slowing down the program executions. On the downside, the event does not discriminate between memory devices/targets of the memory operation.

PAPI_MEM_WCY

This event gives us the number of cycles stalled waiting for memory writes. This event is strongly relevant when assessing the interference between cores, and hence, for timing analysis. When a core is writing into a position, the other cores can't write or read from it, leading to stalls and slowing down the program executions. On the downside, the event does not discriminate between memory devices/targets of the memory operation.

PAPI_RES_STL

This event gives us the number of cycles stalled on any resource. Although catching separately the cycles stalled on every individual resource would be more precise for an analysis, this event could be still useful in case we do not want to do a deep analysis of each one or if the platform's PMC's aren't able to catch them separately.

3.2.1.3 Relevant events not defined in PAPI

Since PAPI is limited to supporting a specific group of conventional high-performance processors, the PAPI selected events do not contemplate those events that are relevant as far as embedded systems are concerned. It is for this reason why this set of events that are considered relevant in this domain, have been added to the library.

ePAPI_PMEM_STL

This event gives us the number of cycles stalled waiting for program memory accesses. This event together with **ePAPI_DMEM_STL** is equivalent to **PAPI_MEM_SCY**. When a core is accessing to a position, the other cores can't access to that position, leading to stalls and slowing down the program executions. Unlike **PAPI_MEM_SCY**, the event does discriminate between reads/writes but not the memory device on which the stall occurs (as in the AURIX there are two PFlash interfaces).

ePAPI_DMEM_STL

This event gives us the number of cycles stalled waiting for program memory accesses. This event together with **ePAPI_PMEM_STL** is equivalent to **PAPI_MEM_SCY**. When a core is accessing to a position, the other cores can't access to that position, leading to stalls and slowing down the program executions. Unlike **PAPI_MEM_SCY**, the event does discriminate between reads/writes and the memory device on which the stall occurs.

ePAPI_MULTI_ISSUE

This event gives is the number of cycles where more than one instruction is issued. It differs from **PAPI_FUL_ICY** because the latter gives us the cycles with maximum instruction issue which in this board isn't the same amount of cycles. This event is strongly relevant when assessing the interference between cores, and hence, for timing analysis.

ePAPI_IPDISP_STL

This event gives us the number of cycles in which the Integer dispatch unit is stalled for whatever reason. This event is a clear indicator of the quality of the code in terms of structural hazards which are the result of a hardware resource that cannot be accessed in any given stage of the pipeline, by more than one instruction.

ePAPI_LSDISP_STL

This event gives us the number of cycles in which the Load/Store dispatch unit is stalled for whatever reason. This event is a clear indicator of the quality of the code in

terms of structural hazards which are the result of a hardware resource that cannot be accessed in any given stage of the pipeline, by more than one instruction.

ePAPI_LPDISP_STL

This event gives us the number of cycles in which the Loop dispatch unit is stalled for whatever reason. This event is a clear indicator of the quality of the code in terms of structural hazards which are the result of a hardware resource that cannot be accessed in any given stage of the pipeline, by more than one instruction.

Chapter 4

Embedded Performance Application Programming Interface (ePAPI): Design and implementation

4.1 Design requirements for an embedded PAPI

When studying the design requirements, we conclude that there are several points to consider.

First, we start from the idea that the library has been developed by mimicking a reference library and that seeks to give support to a type of processors that the standard version of the library does not contemplate. This leads to the need to channel the implementation of the new library in a way that would allow the end user to use it as the standard PAPI, preventing the users from going through a relearning process.

Secondly, in addition to develop the library considering that it is aimed at embedded systems, the avoidance of introducing event count errors to the counters when using the library has been stablished as a maxim.

Last of all, keeping always the functionality in mind, the library has been developed in a platform-independent way, leading to a portable library that can be used in other boards from the same family and others.

4.2 Analysis of the PAPI subset of functions to be included in ePAPI

4.2.1 Introduction to the analysis

In this section it is going to be exposed the analysis carried out in order to discern which functions are going to be part of the subset of functions that will compose the library that is going to be implemented and ran on top of the AURIX TriCore.

4.2.2 High level API functions analysis

Name of the function	*Description*	*Supported?*	*Motivation for supporting it or not*
int PAPI_accum_counters	add current counts to array and reset counters	**Y**	This function is supported since it gives the user the possibility of accumulating values to the array that contains the event results.
int PAPI_num_counters	get the number of hardware counters available on the system	**Y**	This function is supported since it is highly probable that the user requires to know the number of PMCs that the system has.
int PAPI_num_components	get the number of components available on the system	**N**	Reason[2]. Description found at the table caption.
int PAPI_read_counters	copy current counts to array and reset counters	**Y**	It is a basic function to be included in the library since it is the main way to read the PMC.
int PAPI_start_counters	start counting hardware events	**Y**	It is a basic function to be included in the library as it configures and starts the counters.
int PAPI_stop_counters	stop counters and return current counts	**Y**	It is a basic function to be included in the library since it is responsible of stopping the counters.
int PAPI_flips	simplified call to get Mflips/s (floating point instruction rate), real and processor time	**N**	Reason[3]. Description found at the table caption.
int PAPI_flops	simplified call to get Mflops/s (floating point operation rate), real and processor time	**N**	Reason[3]. Description found at the table caption.
int PAPI_ipc	gets instructions per cycle, real and processor time.	**Y**	This function is supported since the platform is able to measure both, total cycles and instructions completed. **Only instructions per cycle is obtained.**
int PAPI_epc	gets (named) events per cycle, real and processor time, reference and core cycles	**Y**	This function is supported since the platform is able to measure both, total cycles and total events (the last one is measured by adding all the values of the *values* array). **Only instructions per cycle is obtained.**

4.2.3 Low level API functions analysis

Name of the function	*Description*	*Supported?*	*Motivation for supporting it or not*
int PAPI_accum	accumulate and reset hardware events from an event set	**Y**	Reason[4]. Description found at the table caption.
int PAPI_add_event	add single PAPI preset or native hardware event to an event set	**Y**	This function is implemented since it is highly probable that the user requires to include new events to the eventSet.
int PAPI_add_named_event	add an event by name to a PAPI event set	**Y**	This function is implemented for the same reasons as *PAPI_add_event* and for the possibility it offers to add trough the name of the event.
int PAPI_add_events	add array of PAPI preset or native hardware events to an event set	**Y**	This function is implemented for the same reasons as *PAPI_add_event* and for the possibility it offers to add more than one event to the eventSet each time.
int PAPI_assign_eventset_component	assign a component index to an existing but empty eventset	**N**	Reason[2]. Description found at the table caption.
int PAPI_attach	attach specified event set to a specific process or thread id	**N**	Reason[5]. Description found at the table caption.
int PAPI_cleanup_eventset	remove all PAPI events from an event set	**Y**	This function is implemented since it is highly probable that the user requires to clean up the eventSet.
int PAPI_create_eventset	create a new empty PAPI event set	**Y**	This function is implemented since the user needs to create an eventset in order to store the events that he wants to measure.
int PAPI_detach	detach specified event set from a previously specified process or thread id	**N**	Reason[5]. Description found at the table caption.
int PAPI_destroy_eventset	deallocates memory associated with an empty PAPI event set	**N**	This functions ins't implemented since the the eventsets are created since the initialization of the library.
int PAPI_enum_event	return the event code for the next available preset or native event	**N**	This function isn't implemented since it isn't considered relevant in the scope of this project.
int PAPI_enum_cmp_event	return the event code for the next available component event	**N**	Reason[2]. Description found at the table caption.

Name of the function	*Description*	*Supported?*	*Motivation for supporting it or not*
int PAPI_event_code_to_name	translate an integer PAPI event code into an ASCII PAPI preset or native name	**N**	Reason[1]. Description found at the table caption.
int PAPI_event_name_to_code	translate an ASCII PAPI preset or native name into an integer PAPI event code	**Y**	This function is implemented since it allows the user to know the PAPI event code for further uses.
int PAPI_get_dmem_info	get dynamic memory usage information	**N**	This function isn't implemented since the hardware isn't able to monitor the dynamic memory usage.
int PAPI_get_event_info	get the name and descriptions for a given preset or native event code	**N**	Reason[1]. Description found at the table caption.
const PAPI_exe_info_t* PAPI_get_executable_info	get the executable's address space information	**N**	Reason[6]. Description found at the table caption.
const PAPI_hw_info_t* PAPI_get_hardware_info	get information about the system hardware	**N**	Reason[6]. Description found at the table caption.
const PAPI_get_multiplex	get the multiplexing status of specified event set	**N**	This function isn't implemented since multiplexing isn't supported in the AURIX the way PAPI needs it to be.
int PAPI_get_opt	query the option settings of the PAPI library or a specific event set	**N**	Reason[6]. Description found at the table caption.
int PAPI_get_cmp_opt	query the component specific option settings of a specific event set	**N**	Reason[2]. Description found at the table caption.
long long PAPI_get_real_cyc	return the total number of cycles since some arbitrary starting point	**N**	This function isn't implemented since getting the total number of cycles as an event, gives the same functionality.
long long PAPI_get_real_nsec	return the total number of nanoseconds since some arbitrary starting point	**N**	Reason[6]. Description found at the table caption.

Name of the function	*Description*	*Supported?*	*Motivation for supporting it or not*
long long PAPI_get_real_usec	return the total number of microseconds since some arbitrary starting point	**N**	Reason[6]. Description found at the table caption.
const PAPI_shlib_info_t* PAPI_get_shared_lib_info	get information about the shared libraries used by the process	**N**	Reason[5]. Description found at the table caption.
int PAPI_get_thr_specific	return a pointer to a thread specific stored data structure	**N**	Reason[5]. Description found at the table caption.
int PAPI_get_overflow_event_index	decomposes an overflow_vector into an event index array	**N**	This function isn't implemented since the size of the eventsets are already the maximum size possible.
long long PAPI_get_virt_cyc	return the process cycles since some arbitrary starting point	**N**	Reason[5]. Description found at the table caption.
long long PAPI_get_virt_nsec	return the process nanoseconds since some arbitrary starting point	**N**	Reason[5]. Description found at the table caption.
long long PAPI_get_virt_usec	return the process microseconds since some arbitrary starting point	**N**	Reason[5]. Description found at the table caption.
int PAPI_is_initialized	return the initialized state of the PAPI library	**N**	Reason[1]. Description found at the table caption.
int PAPI_library_init	initialize the PAPI library	**N**	This function isn't implemented since the PAPI library is initialized automatically.
int PAPI_list_events	list the events that are members of an event set	**Y**	This function is implemented since it is highly probable that the user requires to print the events found in an eventset and for the need of this functions during the development of the library.
int PAPI_list_threads	list the thread ids currently known to PAPI	**N**	Reason[5]. Description found at the table caption.
int PAPI_lock	lock one of two PAPI internal user mutex variables	**N**	Reason[5]. Description found at the table caption.

Name of the function	*Description*	*Supported?*	*Motivation for supporting it or not*
int PAPI_multiplex_init	lock one of two PAPI internal user mutex variables	**N**	Reason[5]. Description found at the table caption.
int PAPI_num_cmp_hwctrs	return the number of hardware counters for a specified component	**N**	Reason[2]. Description found at the table caption.
int PAPI_num_events	return the number of events in an event set	**Y**	This function is implemented since it is highly probable that the user requires to print the number of events found in an eventset and for the need of this functions during the development of the library.
int PAPI_overflow	set up an event set to begin registering overflows	**N**	Reason[6]. Description found at the table caption.
void PAPI_error	Print a PAPI error message	**N**	Reason[1]. Description found at the table caption.
int PAPI_profil	generate PC histogram data where hardware counter overflow occurs	**N**	Reason[6]. Description found at the table caption.
int PAPI_query_event	query if a PAPI event exists	**N**	Reason[1]. Description found at the table caption.
int PAPI_query_named_event	query if a named PAPI event exists	**N**	Reason[1]. Description found at the table caption.
int PAPI_read	read hardware events from an event set with no reset	**Y**	Reason[4]. Description found at the table caption.
int PAPI_read_ts	read from an eventset with a real-time cycle timestamp	**Y**	This function is implemented since it is relevant for timing analysis.
int PAPI_register_thread	inform PAPI of the existence of a new thread	**N**	Reason[5]. Description found at the table caption.
int PAPI_remove_event	remove a hardware event from a PAPI event set	**Y**	This function is implemented since it is highly probable that the user requires to remove events from the eventSet.
int PAPI_remove_named_event	remove a named event from a PAPI event set	**Y**	This function is implemented since it is highly probable that the user requires to remove events from the eventSet.
int PAPI_remove_events	remove an array of hardware events from a PAPI event set	**Y**	This function is implemented since it is highly probable that the user requires to remove events from the eventSet.

Name of the function	*Description*	*Supported?*	*Motivation for supporting it or not*
int PAPI_reset	reset the hardware event counts in an event set	**Y**	This function is implemented since it is considered a basic feature from the library.
int PAPI_set_debug	set the current debug level for PAPI	**N**	Reason[6]. Description found at the table caption.
int PAPI_set_cmp_domain	set the component specific default execution domain for new event sets	**N**	Reason[2]. Description found at the table caption.
int PAPI_set_domain	set the default execution domain for new event sets	**N**	Reason[6]. Description found at the table caption.
int PAPI_set_cmp_granularity	set the component specific default granularity for new event sets	**N**	Reason[2]. Description found at the table caption.
int PAPI_set_granularity	set the default granularity for new event sets	**N**	Reason[6]. Description found at the table caption.
int PAPI_set_multiplex	convert a standard event set to a multiplexed event set	**N**	This function isn't implemented since the library only works with standard eventsets.
int PAPI_set_opt	change the option settings of the PAPI library or a specific event set	**N**	Reason[6]. Description found at the table caption.
int PAPI_set_thr_specific	save a pointer as a thread specific stored data structure	**N**	Reason[5]. Description found at the table caption.
void PAPI_shutdown	finish using PAPI and free all related resources	**N**	
int PAPI_sprofil	generate hardware counter profiles from multiple code regions	**N**	Reason[6]. Description found at the table caption.
int PAPI_start	start counting hardware events in an event set	**Y**	Reason[4]. Description found at the table caption.
int PAPI_state	return the counting state of an event set	**Y**	This function is implemented since it is important from the user's point of view to know the state of an event set.

Name of the function	*Description*	*Supported?*	*Motivation for supporting it or not*
int PAPI_stop	stop counting hardware events in an event set and return current events	Y	Reason[4]. Description found at the table caption.
int PAPI_sterror	return a pointer to the error name corresponding to a specified error code	N	Reason[1]. Description found at the table caption.
int PAPI_thread_id	get the thread identifier of the current thread	N	Reason[5]. Description found at the table caption.
int PAPI_thread_init	initialize thread support in the PAPI library	N	Reason[5]. Description found at the table caption.
int PAPI_unlock	unlock one of two PAPI internal user mutex variables	N	Reason[5]. Description found at the table caption.
int PAPI_unregister_thread	inform PAPI that a previously registered thread is disappearing	N	Reason[5]. Description found at the table caption.
int PAPI_write	write counter values into counters	N	Reason[4]. Description found at the table caption.
int PAPI_get_event_component	return which component an EventCode belongs to	N	Reason[2]. Description found at the table caption.
int PAPI_get_eventset_component	return which component an EventSet is assigned to	N	Reason[2]. Description found at the table caption.
int PAPI_get_component_index	Return component index for component with matching name	N	Reason[2]. Description found at the table caption.
int PAPI_disable_component	Disables a component before init	N	Reason[2]. Description found at the table caption.
int PAPI_disable_component_by_name	Disable, before library init, a component by name.	N	Reason[2]. Description found at the table caption.

Table 2: PAPI functions supported in ePAPI. **Key**:

Reason[1]: This function isn't implemented since there is no screen to print the result. (Nor isn't mappable to memory).

Reason[2]: The function isn't supported due to the nonexistence of components to add/remove.

Reason[3]: The function isn't supported since the platform used doesn't include the possibility of counting floating point events.

Reason[4]: The function is implemented since it is one that is called by its highlevel equivalent and for its relevance when using PAPI at low level.

Reason[5]: The function isn't implemented because the scope of the project only includes one core of the AURIX TriCore and doesn't run an operating system to attach an execution to a process.

Reason[6]: The function isn't implemented since it isn't considered relevant in the scope of this project.

4.2.4 Functions analysis conclusions

Since the project focuses on the study of execution in a single core, the number of PAPI functions to be included in ePAPI is limited as can be seen in the section of reasons that lead to including a function or not. Another important reason that defines the inclusion of a function is the lack of an operating system. This fact leads us not to support functions related to processes or threads handling.

The data output which includes the results of the counts and other relevant information, uses a linker script label to map the information into the AURIX memory, more precisely, to the DSPR which is 120KB big. This aspect gives us a certain rigidity when it comes to data output as we only have the ability to map numeric results.

One relevant aspect that must be pointed out, is the lack of support for the monitoring of events related to floating point units, forcing us not to contemplate the high-level functions related to their measurement (PAPI_flips and PAPI_flops).

4.3 ePAPI implementation

In this section, the implementation of the library object of this project will be presented. For doing so, a series of aspects will be specified, such as the events supported by the library as well as the interface of the counters and the two APIs that use it.

Since ePAPI has been designed for a bare-metal environment, the results are being mapped into a specified direction in the memory because of the lack of ways to print out the information in a common way such as tty or console. The functionality in which the implementation relies in order to allow mapping the results to memory are the linker script labels.

4.3.1 Events and counter interface

ePAPI has a system of event codes that allows the user to refer to them when using the library. The event codes that ePAPI supports are the following;

1. ePAPI_BR_INS
2. ePAPI_L1_DCH
3. ePAPI_L1_ICH
4. ePAPI_L1_ICM
5. ePAPI_L1_DCA
6. ePAPI_L1_DCM
7. ePAPI_L1_ICA
8. ePAPI_L1_ICR
9. ePAPI_MEM_SCY
10. ePAPI_TOT_CYC
11. ePAPI_TOT_INS
12. ePAPI_PMEM_STL
13. ePAPI_DMEM_STL
14. ePAPI_MULTI_ISSUE
15. ePAPI_IPDISP_STL
16. ePAPI_LSDISP_STL
17. ePAPI_LPDISP_STL

The exact meaning of the events, as well as a detailed description, is provided in Chapter 3: AURIX TC275 Performance Monitoring Counters support, section 3.2.

The listed events can be used in conjunction by using Event Sets, which are user-defined groups of hardware events. This means that the user can specify the events to be added to an Event Set which makes the library much easier to use. It is the user's responsibility to choose events that can be counted simultaneously by reading the processors documentation.

In addition to the described advantage, the event set provide us with information related to the set at issue, such its state and the number of events it contains among others. As you can see in the previous list, ePAPI only supports events known as preset events, also known as predefined events, which are a common set of events deemed relevant and useful for application performance tuning. The ePAPI library names 17 preset events, which are defined in the header file, ePAPI.h.

When it comes to the counter interface, it must be clarified that ePAPI is written in C. The function calls are defined in the header file, papi.h and consist of the following form:

```
<returned data type> ePAPI_function_name (arg1, arg2, …)
```

The functions implemented in ePAPI are divided into two distinct groups, the high-level API and the low-level API. As it is normal in this type of topologies, the high-level API makes use of the low-level API for its operation.

When it comes to describe the differences between the standard PAPI and ePAPI, it is imperative to talk about the differences as far as events are concerned. The standard PAPI includes two types of events, the preset and the native ones, while ePAPI only considers the preset. When we go deeper into the study of the supported preset events, the standard PAPI names approximately 100 preset events, while ePAPI only 17. This is due to the limited PMC support that the AURIX offers for being an embedded system.

Both libraries support the use of Event Sets. The difference that exists between both implementations is that in the standard PAPI the user has the ability to configure attributes, such as: the counting domain, whether or not the events are to be used for overflow or profiling while in ePAPI that kind of attributes aren't supported because in the scope of the project only one core is brought to study.

On the other hand, when it comes to describe the differences between the counter interfaces, the first point to clarify is that although written in C, the standard PAPI offers the user the ability to use the library in Fortran as well as in C.

4.3.2 High-level API

The high-level API (**A**pplication **P**rogramming **I**nterface) provides the ability to start, stop, and read the counters for a specified list of events. Some of the benefits of using the high-level API rather than the low-level API are that it is easier to use and requires less setup (additional calls). This ease of use comes with somewhat loss of flexibility.

It should also be noted that the high-level API can be used in conjunction with the low-level API and in fact does call the low-level API. However, the high-level API by itself is only able to access those events countable simultaneously by the underlying hardware.

There are seven functions that represent the high-level API that allow the user to access and count specific hardware events.

- void ePAPI_accum_counters
- void ePAPI_num_counters
- void ePAPI_read_counters
- void ePAPI_start_counters
- void ePAPI_stop_counters
- void ePAPI_ipc
- void ePAPI_epc

4.3.2.1 Execution rate calls

Two ePAPI high-level functions are available to measure total instruction rates. These two calls are shown below:

```
void ePAPI_ipc ()
```

```
void ePAPI_epc(int *EventSet, long long *values, int
NUM_EVENTS)
```

Arguments

int *EventSet	It is an array of codes for events such as ePAPI_TOT_CYC.
int NUM_EVENTS	It is the number of items in the events array.
long long *values	It is an array where to put the counter values.

The first execution rate call sets up the counters to monitor ePAPI_TOT_INS and ePAPI_TOT_CYC (depending on the call) as well as the events found in the specified Event Set (in the case of epc), and starts the counters. Subsequent calls to the same rate function will read the counters and return the instructions per cycle together with the real time, when calling ipc, and the appropriate rate of execution together with the real time when calling epc. A call to ePAPI_stop_counters will reinitialize all values to 0 and stop the counters.

The simultaneous use of both rate calls is incompatible with each other as well as with the rest of the functions described below.

4.3.2.2 Starting, numbering, reading, accumulating and stopping counters

In ePAPI counters can be started, numerated, read, accumulated and stopped by calling the following high-level functions, respectively:

```
void ePAPI_start_counters (int *Events, int NUM_EVENTS)
```

```
void ePAPI_num_counters()
```

```
void ePAPI_read_counters(long long *values, int NUM_EVENTS)
```

```
void ePAPI_accum_counters(long long *values, int
NUM_EVENTS)
```

```
void ePAPI_stop_counters(long long *values, int array_len)
```

Arguments

int *Events	It is an array of codes for events such as ePAPI_TOT_CYC.
int NUM_EVENTS	It is the number of items in the events array.
long long *values	It is an array where to put the counter values.

`ePAPI_num_counters` returns the number of hardware counters available on the system. On the other hand, `ePAPI_start_counters` starts counting the events named in the `*Events` array. This function implicitly stops and initializes any counters running as a result of a previous call to `ePAPI_start_counters`. It is the user's responsibility to choose events that can be counted simultaneously by reading the processors documentation. The size of `NUM_EVENTS` shouldn't be larger than the value returned by `ePAPI_num_counters`.

`ePAPI_read_counters`, `ePAPI_accum_counters` and `ePAPI_stop_counters` all capture the values of the currently running counters into the array `*values`, although each of them behaves somewhat differently.

`ePAPI_read_counters` copies the current counts into the elements of the `*values` array, resets the counters to zero, and leaves the counters running.

`ePAPI_accum_counters` adds the current counts into the elements of the `*values` array, resets the counters to zero, and leaves the counters running.

`ePAPI_stop_counters` stops the current counts and writes the current counts into the elements of the `*values` array.

4.3.2.3 Differences between the high-level APIs of PAPI and ePAPI

In the following section the differences between PAPI and ePAPI are described as far as the high-level API is concerned.

Execution rate calls

When it comes to the execution rate calls, the standard PAPI includes two more functions that aren't supported in ePAPI. The functions at issue are called `PAPI_flips` and `PAPI_flops`. The reason why they haven't been implemented in ePAPI is because of the lack of support on floating point operations in the AURIX. As their names indicate,

these functions are able to return the Mflips/s (floating point instruction rate) and the Mflops (floating point operation rate) and worked similar to `ePAPI_ipc` and `ePAPI_epc` functions.

Starting, numbering, reading, accumulating and stopping counters

On the other hand, regarding the starting and numbering functions, known as `ePAPI_num_counters` and `ePAPI_start_counters`, it should be noted that when used in the standard PAPI, they have an additional function which is initializing the library. This functionality hasn't been added in ePAPI because it is unnecessary to initialize the library because of the way it has been implemented.

4.3.3 Low-level API

The low-level API (**A**pplication **P**rogramming **I**nterface) manages hardware events in user-defined groups called Event Sets. It is meant for experienced application programmers and tool developers wanting fine-grained measurement and control of the PAPI interface. Other feature of the low-level API is the ability to obtain information about the executable. Some of the benefits of using the low-level API rather than the high-level API are that it increases efficiency and functionality.

4.3.3.1 Creating, adding, removing, and emptying (to) an event set

As described above, the low-level API manages hardware events in groups called Event Sets. These structures can be created, emptied and can be added and removed events. This operations give the user the ability to split hairs as far as event sets are concerned.

Creating an Event Set

An event set can be created by calling the following low-level function:

```
void ePAPI_create_eventset (int *EventSet)
```

Argument

int *EventSet	Address of the location where the EventSet identifier is.

Once it is created, the user may add hardware events to the EventSet by calling ePAPI_add_event or ePAPI_add_events.

Adding events to an Event Set

Hardware events can be added to an event set by calling the following low-level functions:

```
void ePAPI_add_event (int EventSet, int EventCode)
```

```
void ePAPI_add_events (int EventSet, int *EventCode, int
size)
```

Arguments

int EventSet	An integer handle for an Event Set created by `ePAPI_create_eventset`.
int EventCode	A defined event such as ePAPI_TOT_CYC
int *EventCode	Addres of an array of defined events.
size	An integer that indicates the number of events in the array *EventCode

`ePAPI_add_event` adds a single hardware event to an Event Set, while `PAPI_add_events` does the same as `ePAPI_add_event`, but for an array of hardware event codes. Implicitly, `ePAPI_add_event` calls `ePAPI_add_event` to do the addition.

Removing and emptying events in an Event Set

A hardware event and an array of hardware events can be removed from an Event Set by calling the following low-level functions, respectively:

```
void ePAPI_remove_event (int EventSet, int EventCode)
```

```
void ePAPI_remove_events(int EventSet, int *EventCode, int
size)
```

Arguments

int EventSet	An integer handle for an Event Set created by `ePAPI_create_eventset`.
int EventCode	A defined event such as ePAPI_TOT_CYC
int *EventCode	Addres of an array of defined events.
size	An integer that indicates the number of events in the array *EventCode

ePAPI_remove_event removes a single hardware event to an Event Set, while PAPI_remove_events does the same as ePAPI_remove_event, but for an array of hardware event codes. Implicitly, ePAPI_remove_event calls ePAPI_remove_event to do the removal.

In addition, all the events in an event set can be emptied by calling the following low-level function:

```
void ePAPI_cleanup_eventset (int EventSet)
```

Argument

int EventSet	An integer handle for an Event Set created by ePAPI_create_eventset.

Starting, reading, adding and stopping events in an Event Set

Hardware events in an Event Set can be started, read, added, and stopped by calling the following low-level functions, respectively:

```
void ePAPI_start (int EventSet)
void ePAPI_read (int EventSet, long long *values)
void ePAPI_accum (int EventSet, long long *values)
void ePAPI_stop (int EventSet, long long *values)
```

Arguments

int EventSet	An integer handle for an Event Set created by ePAPI_create_eventset.
long long *values	An array to hold the counter values of the counting events.

ePAPI_start starts counting the events of a previously defined Event Set.

ePAPI_read copies the current counts into the elements of the *values array. The counters are left counting after the read without resetting.

ePAPI_accum adds the current counts into the elements of the *values array. The counters are left counting after the read without resetting.

ePAPI_stop stops the current counts and writes the current counts into the elements of the *values array.

Resetting events in an event set

The hardware event counts in an event set can be reset to zero by calling the following low-level function:

```
void ePAPI_reset (int EventSet)
```

Argument

int EventSet	An integer handle for an Event Set created by `ePAPI_create_eventset`.

4.3.3.2 State of an Event Set

The counting state of an Event Set can be obtained by calling the following low-level function:

```
void ePAPI_state (int EventSet)
```

Argument

int EventSet	An integer handle for an Event Set created by `ePAPI_create_eventset`.

The functions maps to memory a 1 if the Event Set is running or a 0 if the Event Set isn't running.

4.3.3.3 Differences between the low-level APIs of PAPI and ePAPI

In the following section the differences between PAPI and ePAPI are described as far as the low-level API is concerned.

Initialization of the low-level API

In the same way as in the high-level API, in the low-level API the standard PAPI needs to be initialized. For doing so, PAPI uses a function named `PAPI_library_init`. This previous step is not required by ePAPI for its operation.

Destroying an Event Set

The standard PAPI gives the user the ability to destroy an Event Set by calling the function named `PAPI_destroy_eventset`. This function is not contemplated in ePAPI because they are automatically destroyed by the end of the execution.

State of an Event Set

The standard PAPI uses a different prototype function to get the state of an Event Set:

```
PAPI_state (int EventSet, *status)
```

This is because in PAPI the state of an Event Set can be more than two. While in ePAPI an Event Set can be either running or not running, in the standard PAPI are considered all the states described below which aren't contemplated in ePAPI because of the scope of the project.

`PAPI_STOPPED`	*EventSet* is stopped
`PAPI_RUNNING`	*EventSet* is running
`PAPI_PAUSED`	*EventSet* temporarily disabled by the library
`PAPI_NOT_INIT`	*EventSet* defined, but not initialized
`PAPI_OVERFLOWING`	*EventSet* has overflow enabled
`PAPI_PROFILING`	*EventSet* has profiling enabled
`PAPI_MULTIPLEXING`	*EventSet* has multiplexing enabled
`PAPI_ATTACHED`	*EventSet* is attached to another thread/process

Chapter 5

ePAPI verification and validation

5.1 Methodology

When it comes to the methodology followed, two aspects of it must be highlighted. The test environment and the followed procedure.

5.1.1 Test environment

The test environment of the verification and validation of the library includes all the resources of the project environment. A brief summary of them is found below:

- Cygwin and Tricore-gcc: that allowed us to compile the sources that contained the library and the test programs.
- Universal Debug Engine UDE: that allowed us to upload the binaries and execute them on the selected cores as well as check the results obtained after the execution.
- A translation script: that let us translate the mapped-to-memory results into readable text.
- Remote access to a PC found at the BSC to which the AURIX TriCore platform is connected.

5.1.2 Followed procedure

The procedure followed has been focused in both, verifying and validating the developed library. On the one hand, it consisted in the design and implementation of test programs that seeked to stress certain components of the AURIX that are directly related to the events of the library to be validated. On the other hand, regarding the verification, a selection and test of possible use scenarios of the library was studied.

The implemented stress programs were developed to prove the ePAPIs consistency and accuracy with respect to bare-metal PMC readings. So the validation consisted in executing the stress programs both, in bare-metal and with ePAPI separately to check if ePAPI introduced count errors to the counters.

5.2 Verification of ePAPI functions

In the following section we will describe the verification process carried out in order to verify the correct functioning of the resulting library of this project. To do this, a method that would allow us to study the possible use cases was selected in order to verify the correct functioning of the library and its uses.

The selected method was the recreation of the most common use cases from the library. In order to get those common use cases, we looked for examples in the user's guide of the standard PAPI (found in the ePAPI user's guide as well). In addition to these tests, we performed independent verifications of each of the functions that compose ePAPI.

Both, the high-level API and the low-level API were verified together and separately. When verifying the high-level API, some of the examples included calling to a big set of the functions that ePAPI supports. —i.e, numbering the events together with the call that list them, starting the counters followed by several reads that were interleaved by pieces of benchmarks or stress programs, alternating resets and accumulations of the counters and finally stopping the counters.

On the other hand, when it comes to the low-level API, similar examples were executed but adding Event Set manipulation functions as well as information obtaining functions about the executable.

5.3 Validation of ePAPI

As described above, the validation of ePAPI consisted in the design of specific stress programs that were executed separately in bare-metal and using the library, with the objective of checking if the library introduced count errors to the counters whilst running the programs and counting the events.

5.3.1 Experiment set-up

As it has been described in chapter 4, ePAPI supports 17 monitoring events. Checking if ePAPI introduces count errors to the counters is mandatory for each of them, so this section will describe the experiment set-up that includes tests for each event.

For each event a stress program was designed in assembly. The stress programs were embedded into two programs per event, in one of them the counters were configured and used in bare-metal. In the other one, ePAPI was used to perform the count. The used programs are found in the annex to this document.

5.3.2 Results

In the following table, we can find all the results obtained from the tests carried out to verify that the developed library does not introduce count errors to the counters in order to be qualified as a useful dependable performance hardware counter measurement tool.

ePAPI_BR_INS	**Total branches**	**Total cycles**	**Total instructions**
Bare-metal	4.001.001	6.355.019	6.009.010
ePAPI	4.001.004	6.019.042	6.009.021
ePAPI_L1_DCH	**Data cache hits**	**Total cycles**	**Total instructions**
Bare-metal	5.999	187.028	32.010
ePAPI	5.999	187.051	32.021
ePAPI_L1_DCM	**Data cache misses**	**Total cycles**	**Total instructions**
Bare-metal	6.001	187.029	32.010
ePAPI	6.001	187.051	32.021
ePAPI_L1_DCA	**Data cache accesses**	**Total cycles**	**Total instructions**
Bare-metal	12.000	187.028	32.010
ePAPI	12.000	187.051	32.021
ePAPI_L1_ICH	**Instr. cache hits**	**Total cycles**	**Total instructions**
Bare-metal	499.000	11.028.024	1.017.010
ePAPI	499.000	11.028.048	1.017.021
ePAPI_L1_ICM	**Instr. cache miss**	**Total cycles**	**Total instructions**
Bare-metal	-	-	-
ePAPI	-	-	-
ePAPI_L1_ICA	**Data cache misses**	**Total cycles**	**Total instructions**
Bare-metal	499.002	11.028.021	1.017.010
ePAPI	499.002	11.028.052	1.017.021
ePAPI_L1_ICR	**Data cache misses**	**Total cycles**	**Total instructions**
Bare-metal	499.002	11.028.021	1.017.010
ePAPI	499.002	11.028.054	1.017.021
ePAPI_MEM_SCY	**Memories stall**	**Total cycles**	**Total instructions**
Bare-metal	197.007	289.014	156.010
ePAPI	197.007	289.040	156.021
ePAPI_DMEM_STL	**Data mem. stall**	**Total cycles**	**Total instructions**
Bare-metal	126.001	181.015	47.010
ePAPI	126.001	181.037	47.021
ePAPI_PMEM_STL	**Progr. mem. stall**	**Total cycles**	**Total instructions**
Bare-metal	17.005	53.015	31.010
ePAPI	17.006	53.040	31.021
ePAPI_MULTI_ISSUE	**Multi issues**	**Total cycles**	**Total instructions**
Bare-metal	100.013	120.039	230.050
ePAPI	100.016	120.062	230.061

ePAPI_IPDISP_STL	Instr. Disp. stall	Total cycles	Total instructions
Bare-metal	1.000	17.015	17.010
ePAPI	1.000	17.037	17.021
ePAPI_LSDISP_STL	L/S Disp. stall	Total cycles	Total instructions
Bare-metal	2.002	17.015	17.010
ePAPI	2.006	17.037	17.021
ePAPI_LPDISP_STL	Loop Disp. stall	Total cycles	Total instructions
Bare-metal	-	-	-
ePAPI	-	-	-

As we can see in the results contained in the table, ePAPI allows the users to use the PMCs of the AURIX to count events without adding count errors to them.

In addition to the values corresponding to the events that have been sought to be measured while stressing certain parts of the AURIX, in each comparison the values of total cycles as well as of total instructions have been obtained. This allows us to have a deeper insight as far as the comparative between bare-metal counting and the count done with ePAPI is concerned. By using this information, we get the possibility to check further relevant information — e.g. the striking fact that ePAPI always executes 11 instructions more than its equivalent in bare-metal, regardless of the total number of instructions. This is logical due to the differences in the transmission, start and end of both systems. In any case, the constancy of these 11 instructions is a negligible, assumable, and if critical, correctable difference.

As a direct consequence of the increase in the number of instructions, the number of cycles is affected by their duration. This is reflected in turn in a constant difference of around 25 cycles.

As we can see for the values corresponding the ePAPI_LPDSIP_STL and ePAPI_L1_ICM events, the fields are null. This is due to the impossibility of obtaining the results, although favorable, of the experimental system found at the BSC due to a connectivity problem that had place in the final phases of the project. Once the issue is solved, the results will be collected and sent to the academic tribunal as an *erratum addendum* if allowed.

Chapter 6

Main contributions, achieved objectives, and future directions

6.1 Main contributions and achieved objectives

One of the main investigation lines of the Barcelona Supercomputing Center works in the context of several industrial and research projects in collaboration with some of the main tools, components and OEM industries in the automotive, avionics and space domains. As part of those projects, a number of ARM, NVIDIA, Zynq and Infineon boards, among others, need to be set up and interfaced through debug interfaces and/or low-level software to monitor the execution of programs, either with or without an OS layer. Porting applications from the critical real-time systems domains (namely automotive, avionics and space) for their evaluation on top of the identified target platforms, are a necessary step to provide industry with information on how to use those platforms reliably for the execution of their most critical software, such as that responsible of navigation in planes/satellites, and autonomous driving in cars.

This project seeks to interface one of the boards supported by the CAOS group to make possible the monitoring of the execution of programs ran on top of it in a simpler way, since the modus operandi so far consisted in making measurements in bare-metal.

One of the premises of the project has been the implementation of the functionality at issue in a platform-independent way, which has led to a synergy that has allowed us to obtain a library that can be exported to other platforms, such as one of those described at the beginning of this section.

By the end of this project, I have become familiar with a current and representative platform of the automotive domain such as the AURIX TriCore TC275, as well as have achieved a deep knowledge of the PMC support specially in relation with the PAPI organization. I have worked scientifically to obtain verifiable results and have obtained the ability to present and explain the different techniques, evaluations and results with clarity and scientific rigor.

6.2 Critical assessment

When a project of these characteristics is developed, we find that some desirable characteristics are not covered by certain limitations added by the selected hardware as well as by the scope of the project itself. In this section, we will analyze and discuss the limitations at issue.

When we study the limitations introduced by the board, we can highlight some that are typical of the embedded systems domain and that include premises on which their design is based, such as energy saving among others. e.g., the limited number of PMCs and the lack of support that these give to monitor FPU events. Regarding the number of PMCs, these are considered to be scarce, mainly when differentiated with the amount found in conventional processors which is the common amount with which PAPI works with. This idea is strengthened when we remember that three of the five PMCs that the system has, are multiplexed, what introduces certain rigidity to its use. Following the same line, it is worth highlighting the need to make use of the high efficient core (core 0) to govern the high-performance ones (cores 1 and 2), which is the type of core taken to study, what again introduces rigidity to the use of the system.

On the other hand, when we study the limitations introduced by the scope of the project, since it has been carried out around a development board, it has lacked a way of displaying the results of the count apart from the UDE. What led us to have to map the result to a certain memory address to be able to consult the obtained values. In addition, the lack of operating system led us to develop the library in bare-metal which among other aspects, prevented us from having a user interface.

6.3 Future directions

As it happens in the closure of most projects, we can list a series of possible extensions or related future studies. Next, we highlight some that may be interesting:

- Since the scope of this project has only considered the performance efficient cores, it could be interesting to extend the present study taking into account the energy efficient one;
- There are certain aspects that have not been appreciated in the validation process, such as the in-depth study of the different ways that ePAPI offers to read the counters such as ePAPI_accum/ePAPI_accum_counters, ePAPI_read_ts, etc., although being all based originally on the same method.;
- The real-time operating system (RTOS) layer was not considered in the scope of the project. A subset of PAPI only makes sense in the presence of an RTOS (e.g., per task counters support). It could be interesting to extend this study by considering an automotive RTOS (e.g., Erika Enterprise RTOS).

Bibliography

[1] PAPI USER'S GUIDE Version 3.5.0

[2] PAPI 5.6.0.0: The High-Level API. http://icl.cs.utk.edu/papi

[3] PAPI 5.6.0.0: The Low-Level API. http://icl.cs.utk.edu/papi

[4] AURIX TC27x B-Step 32-Bit Single-chip Microcontroller User's Manual V 1.4.1 2014-02. Infineon Technologies AG

[5] TriCore V1.6 Core Architecture 32-bit Unified Processor Core User Manual (Volume 1) V1.0, 2012-05. Infineon Technologies AG 81726 Munich

[6] TriCore V1.6 Core Architecture 32-bit Unified Processor Core User Manual (Volume 2) V1.0, 2012-05. Infineon Technologies AG 81726 Munich

[7] TriCore ™ Compiler Writer's Guide 32-bit Unified Processor Edition 2003-12. Infineon Technologies AG D-81541 München, Germany

[8] TriCore ™ 1 Pipeline Behavior & Instruction Execution Timing TriCore ™ 1 Modular (TC1M). Infineon Technologies AG 81726 Munich

[9] Can Hardware Performance Counters be Trusted? Vincent M. Weaver and Sally A. McKee. Computer Systems Laboratory Cornel University.