Ref. Ares(2024)2277928 - 26/03/2024

D4.1 Interim Platform Technologies Report

Version 1.0

Documentation Information

Contract Number	101069595
Project Website	www.safexplain.eu
Contratual Deadline	31.03.2024
Dissemination Level	PU
Nature	R
Authors	Enrico Mezzetti (BSC), William Guarienti (EXI)
Contributors	Mikel Fernandez (BSC), Sergi Vilardell (BSC), Francisco Cazorla (BSC)
Reviewer	Gabriele Giordana (AIKO)
Keywords	Platform support, Hardware and Software stack, Timing characterization

This project has received funding from the European Union's Horizon Europe programme under grant agreement number 101069595.

SAFEXPLAIN

Change Log

Version	Description Change
V0.1	First draft
V0.2	Reviewed version
V1.0	Final version

Table of Contents

Executive Summary 3
1 Introduction 4
1.1 Scope 4
1.2 Structure of the Document 5
2 NVIDIA ORIN MPSoC 6
2.1 Orin overview 6
2.2 Default software stack 7
3 Timing Interference Control (T4.1) 9
3.1 Software Sources of Timing Interference 9
3.2 Hardware Sources of Timing Interference 12
3.3 Technological assessment 22
4 Observability Channels (T4.2) 24
4.1 PMU and HEM analysis 24
4.2 HEMs identification 25
4.3 Hardware Event Monitors PMULib 29
4.4 SCF HEMs overhead 31
4.5 Technological assessment 33
5 Timing Prediction Methods and Tools (T4.3) 35
5.1 Timing characterization strategy 35
5.2 Inter-Run Variability 35
5.3 Statistical Analysis based on the Markov Inequality 40
5.4 Interference monitoring mechanism and Templates 42
5.5 Technological assessment 44
6 Platform- and System-level V&V support (T4.4) 48
6.1 SAFEXPLAIN Middleware concept 48
6.2 SAFEXPLAIN Middleware support 49
6.3 Technological assessment 53
7 Acronyms and Abbreviations 55
8 References 56
9 Annex 1 – PMULib interface 58
9.1 Function Documentation 58
9.2 Macro Documentation 60
9.3 Usage Example 63

Executive Summary

This deliverable reports on the technical and technological progresses achieved in WP4 during the first 18 months (MS2) of the project. In particular, this report captures the advancements done in all WP4 tasks (T4.1-T4.4) by MS2 hence covering aspects related to hardware and timing characterization as well as specific solutions adopted at platform level to support the activities of other work packages. In the following we provide an assessment of the degree of completion in each of the WP4 tasks and the respective outcomes (technologies and tools). Technologies and tools will be also assessed with respect to their readiness for integration and next steps.

1 Introduction

This report reports on the progress achieved in the scope of WP4. This work package brings together all platform-level aspects that are relevant for the supporting both performance and FUSA requirements on top of the platform. The overarching goal of WP4 is to support the development, execution, and analysis of the solutions proposed by other technical work packages and deployed through SAFEXPLAIN case studies.

1.1 Scope

The Platform WP comprises 4 tasks and a higher-level meta-task to support the integration of WP2 and WP3 solutions in the case studies. As such, WP4 has strict relations with all SAFEXPLAIN work packages and, in fact, facilitates their alignment. Figure 1 below depicts the main tasks in WP4 and how they support SAFEXPLAIN technologies and integration by capturing explicit and implicit requirements from other WPs.

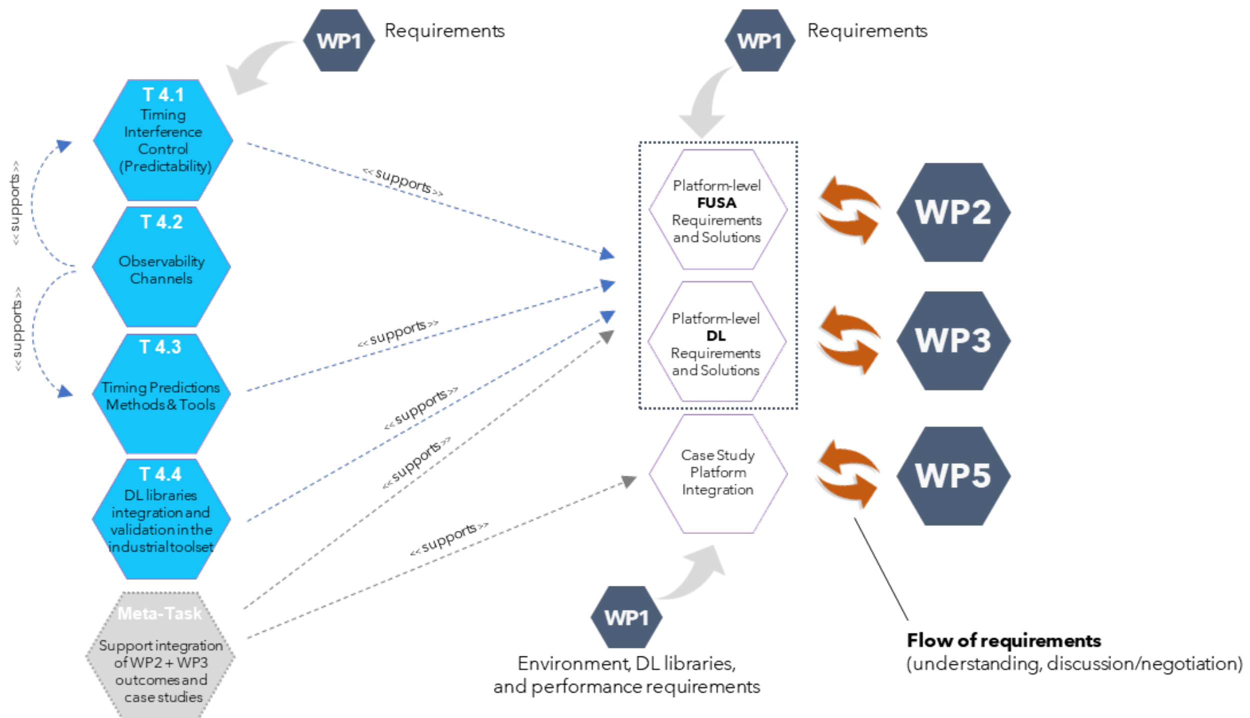


Figure 1 - WP4 role and relation with other WPs.

The main technological tasks in the WP are:

- ***T4.1 Timing interference control***, covering the hardware analysis of the target platform to identify the sources of interference and the available support for segregation and partitioning. This task is critical to support FUSA aspects, and particularly the deployment, under the supervision of WP2, of FUSA architecture and patterns presented in [1].
- ***T4.2 Observability channels***, dealing with available means of collecting hardware-level information on program execution on top of the target platform, and providing an

integrated tool to configure those means and access extract the relevant information at both run and analysis time.

- ***T4.3 Timing prediction methods and Tools***, providing support for the analysis of the timing behavior of the deployed functionalities, building on timing interference mitigations enabled by T4.1 analysis and SAFEXPLAIN FUSA solutions (WP2) and exploiting timing information gathered on top of T4.2 outcomes.
- ***T4.4 DL libraries integration and validation in the industrial toolset***, facilitating the integration of SAFEXPLAIN DL libraries and solutions in a partially automated setup supporting FUSA task through offline V&V activities and run-time monitoring.

1.2 Structure of the Document

In the following sections we provide a review of WP4 activities and progresses up to MS2.

- We will start with a section devoted to the introduction to the NVIDIA AGX Orin [2], the reference platform adopted in SAFEXPLAN. This covers both hardware and software aspects.
- We will then continue following the task structure of the WP.

Each section will include a description of the task objectives, the strategy followed, the obtained results, and an assessment over maturity of the provided solutions from the standpoint of the integration on the case studies.

2 NVIDIA ORIN MPSoC

One of the main objectives of WP1 consisted in the selection of the relevant target boards for the project. The selection process aimed to identify target platforms that were both representative of the target domains (critical embedded systems) and therefore interesting from the FUSA perspective, and apt to sustain the execution of performance intensive AI-based applications, hence providing support for general-purpose and AI-specific hardware accelerators.

Based on the technological and performance requirements emerging form the use cases, the project partners reached a consensus on the adoption of the NVIDIA AGX Orin [2] as target platform.

In the following we summarize the main hardware features relevant for the project. We also report on the adopted software stack, which is equally relevant to provide a homogeneous development and execution environment across project partners and tools.

2.1 Orin overview

The NVIDIA Jetson AGX Orin is a family of heterogenous MPSoC (Orin 32/64 Nano) developed by NVIDIA to cover the emerging requirements form diverse markets, all sharing the need for high-performance to support AI-based functionalities at reduced SWaP (Size, Weight, and Power). In SAFEXPLAIN, the AGX Orin Dev Kit has been selected.

The Orin comprises 3 clusters of 4 Arm Cortex-A78AE CPUs [3], a NVIDIA Ampere GPU, ad-hoc AI-oriented accelerator such as NVDLA and PVA, as well as a video encoder and a video decoder (see Figure 2). The system also exploits a high-speed IO, with 204 GB/s of memory bandwidth, and 32GB of DRAM (in the Dev Kit version). The Orin can deliver up to 275 TOPS which are enabling the execution of multiple concurrent AI application.

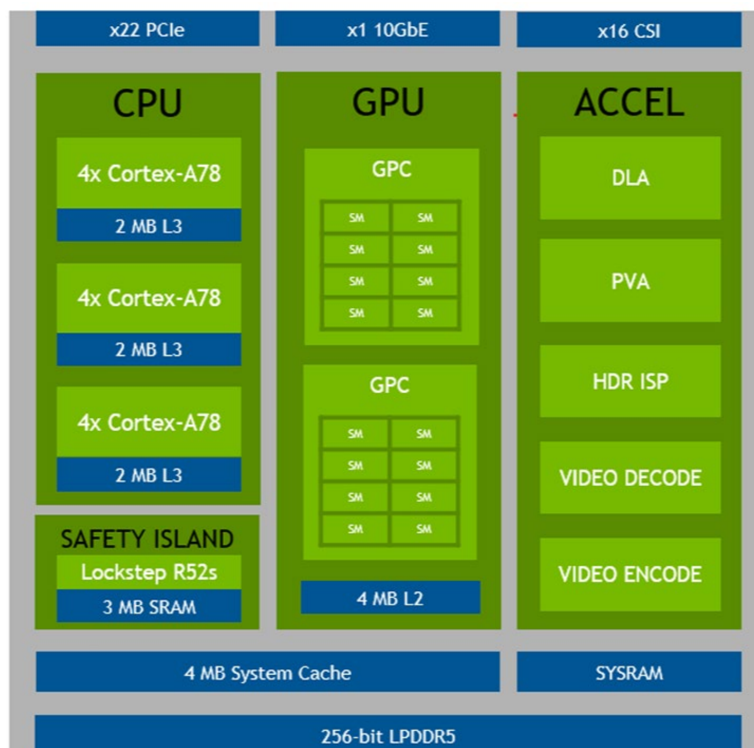


Figure 2 - Block Diagram of our target platform (from [4]).

2.2 Default software stack

The NVIDIA AGX Orin [2] comes with tailored OS support and libraries. The software stack includes a specific version of a Linux-based Operating System as well as a score of dedicated libraries to support the development and execution of AI applications. To favour homogenization and coordination across development environments in the different WPs, WP4 promoted the early identification of a shared software stack configuration to guarantee inter-compatibility of tools. Figure 3 illustrates the positioning of the low-level software layer in the SAFEXPLAIN stack.

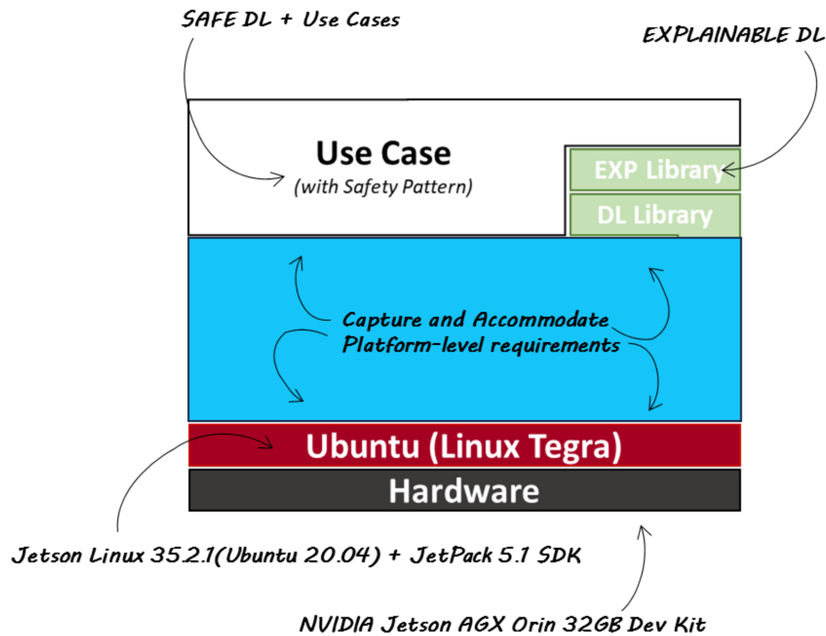


Figure 3 - Baseline HW and SW stack.

As anticipated the low-level software stack includes a tailored instance of an Ubuntu release also called Linux Tegra version. The set of tools normally available on Linux systems is complemented by the NVIDIA proprietary SDK JetPack, providing the necessary support to exploit the programming of the GPU and accelerators and the exploitation of standard deep learning libraries. Technological solutions developed in WP2 and WP3 are meant to rely on this software substrate. The setup is also the baseline for the development and porting of the different case studies.

Current SAFEXPLAIN setup consists in the following elements and versions:

- Jetpack 5.1
- Jetson Linux 35.2.1
- Ubuntu Version 20.04
- Kernel Version 5.10.65-tegra
- Tensor RT 8.5.2
- cuDNN 8.6.0
- CUDA 11.4.19
- OpenCV 4.5.4
- Python 3.8.10
- PyTorch 1.14
- Vulcan 1.3.203
- Vulcan SC 1.0

Changes to the setup are expected to happen during the project, only after checking the compatibility of the updates with the partners assumptions. We expect to move to the new release of the Jetson Linux and JetPack later this year (the new version is still under beta release).

3 Timing Interference Control (T4.1)

This task analyses and classifies the sources of timing interferences, and the hardware/software mechanisms available to control such interference (e.g., cache partitioning). Then, it uses those mechanisms to limit timing interference by construction, especially for DL-based tasks, as part of the SAFEXPLAIN software stack, so they can be used in the case studies, and allow implementing the 'containers' described in T2.4. Timing interference analysis strategy, including analysis time and run-time interference mitigation and control approaches, are designed, and developed in alignment with WP2 strategy [1]. At the top level we classify the sources of timing interference between software (Section 3.1) and hardware (Section 3.2).

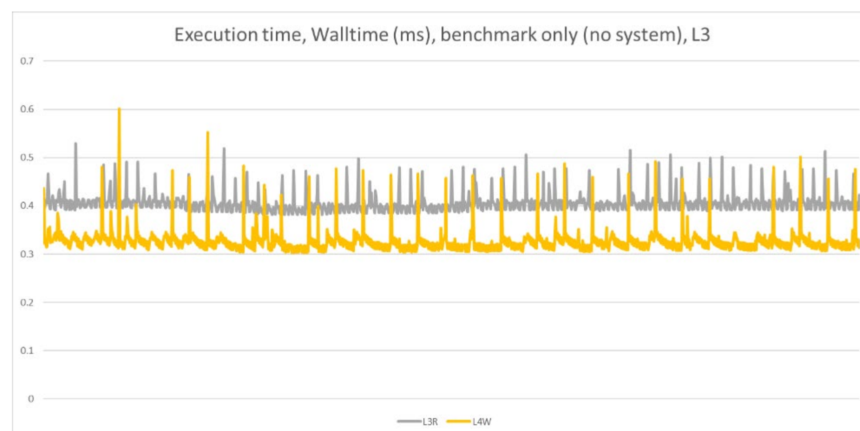
3.1 Software Sources of Timing Interference

The Orin software stack builds on Linux Tegra, which consists in the tailoring of a full Ubuntu distribution. The use of a Linux-based, general-purpose operating system introduces some interference or jitter in the execution of tasks stemming from the many background activities the OS is undergoing. This would not be the case in real-time operating systems where OS services are limited, and timing of user applications is preserved. Real-time OSes, however, are not available on the target platform.

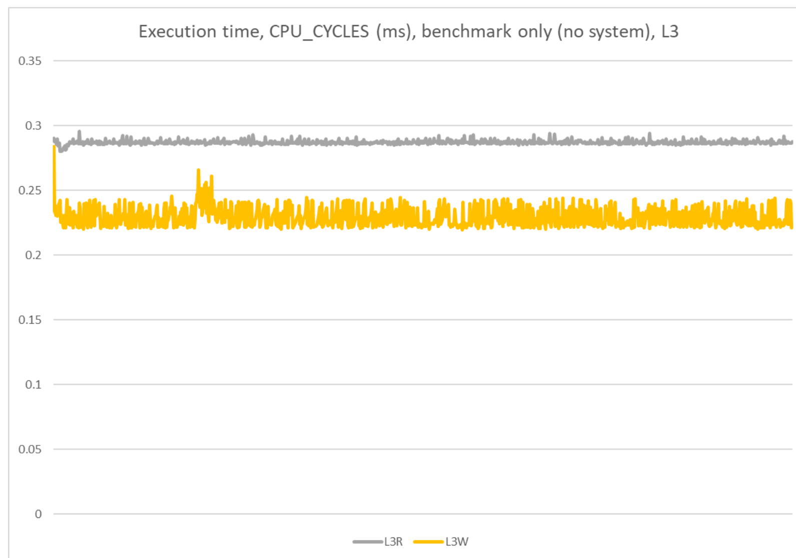
3.1.1 Linux Ubuntu setup

We start our analysis by running several reference benchmarks on the default Orin software-stack, only including the Linux layer. Selected benchmarks consist in carefully designed pieces of code exercising configurable large amount of a specific type of data accesses (read or write operations) to a specific layer in the cache and memory hierarchy (L2, L3 L4, or main memory). We tracked the wall-clock time and the cycle count. Figure 4 shows the results for a benchmark performing reads to the L3 (L3R) and the benchmark performing writes to the L3 (L3W).

Figure 4a shows that every 15 to 30ms approximately the wall clock time suffers a spike. This behavior can be most clearly observed for the L4W benchmark. This relates to activities of the Linux OS that are triggered periodically. Figure 4b shows that those peaks do not arise when we measure the actual execution time of the task, i.e. the time it was actually running in the CPU: as it can be seen the peaks disappear. This confirms that the peaks are due to the OS activity.



(a) Execution time measured as the Wall clock time using clock_gettime().



(b) Execution time measured with the hardware event monitor CPU_CYCLES

Figure 4 - Execution time of the L3R and L3W benchmarks.

We extended the analysis to other benchmarks performing reads and writes to the different levels of the memory hierarchy (L2, L3, and memory). For each of them, we perform a comparison between the wall time and the clock time as reported by the CPU_CYCLES counter. We identify that the wall clock time is approximately 40% higher than the time reported by CPU_CYCLES, due to activities performed by the OS.

	Ratio (avg)
L2R	1.45
L2W	1.48
L3R	1.41
L3W	1.46
L4R	1.38
L4W	1.38
MEMR	1.34
MEMW	1.36

These results confirm the existence of OS activities that can affect the execution time of running programs and hence need to be removed.

3.1.2 ROS2 setup

The Robotic Operating System (ROS) is the de-facto standard for developing complex autonomous systems characterized by a strong interaction with the physical environment. ROS, which underwent through a major release and is now available in version 2 (aka ROS2), is also typically used as a middleware layer in AI-based applications in several domains. ROS2 [5] provides a relatively simple and scalable application semantics based on communication among functional nodes that cooperates to deploy a given functionality, from sensers to actuators. The communication semantics is based on the publisher-subscriber paradigm.

In the scope of SAFEXPLAIN, ROS2 has been selected as a baseline element in the software stack as all use cases clearly fit its publisher-subscriber semantics. More details on SAFEXPLAIN software stack, and how ROS2 fits in it, are provide and discussed in Section 6.1.

ROS2 contributes an additional software layer that can introduce further jitter on top of the OS-induced one.

We assessed the overheads introduced by ROS2 on top of those introduced by Linux. To that end we ported the benchmarks as ROS2 nodes and executed them, measuring CPU_CYCLES. We Compared the resulting distributions of two benchmarks, L2R and L3R.

In Figure 5 and Figure 6 we see that the shape of the distribution of execution for both experiments done with Linux and with Linux + ROS2 are very similar. ROS2 seems to add some overhead to the overall execution time, which is relatively small, about a 0.5% increase in the distribution mean in both cases. Nonetheless, it does not provide additional noise to the execution time.

We can test that by compensating the overhead induced by ROS2 and comparing the distributions with a Kolmogorov-Smirnov (KS) test. The KS test uses as statistic the maximum difference in the Empirical Cumulative Distribution Function (ECDF) in this case. When performing the KS test with bootstrap resampling for a more robust estimation, the p-values are 0.40 and 0.47 for L2R and L3R respectively, therefore we cannot reject that the distributions are different.

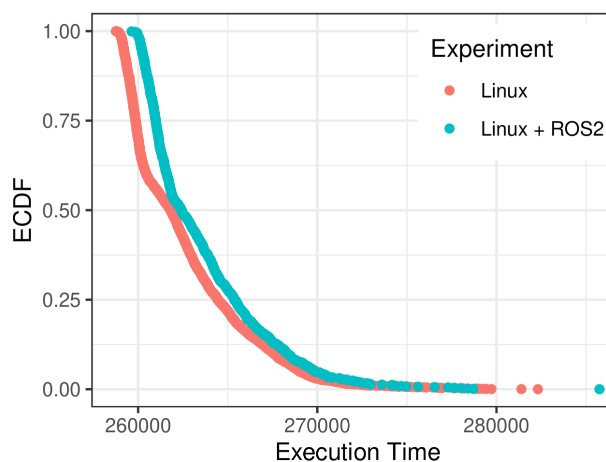


Figure 5: Empirical Cumulative Distribution Functions for the execution time of L2R with Linux and Linux + ROS2

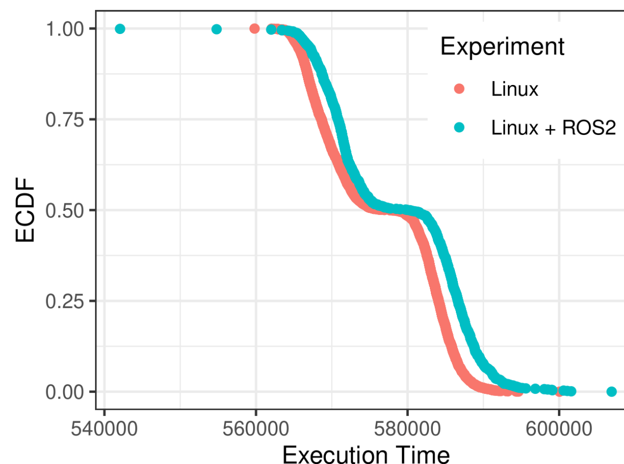


Figure 6: Empirical Cumulative Distribution Functions for the execution time of L3R with Linux and Linux + ROS2

Overall we conclude that not additional means are needed to control ROS2 impact on the execution time of applications.

3.1.3 Custom middleware setup

For the sake of completeness, the software-level interference analysis will be completed by considering the additional layer contributed by the SAFEXPLAIN Middleware, which is an abstraction layer we developed to accommodate various project level aspects, as detailed in Section 6.1. A thorough analysis is postponed to the final release of the Middleware and, hence, results will be included in the next release of this deliverable. Nonetheless, we performed some preliminary experiments on an intermediate version which seems to confirm the trend observed with ROS2. This is indeed not surprising in reason of the limited intrusiveness of the Middleware on the functional behavior, which is preserved and contained within the ROS2 entities.

3.2 Hardware Sources of Timing Interference

A high-level view of the Orin block diagram, showing the main clusters and devices, is provided in Figure 7) from the official documentation [2].

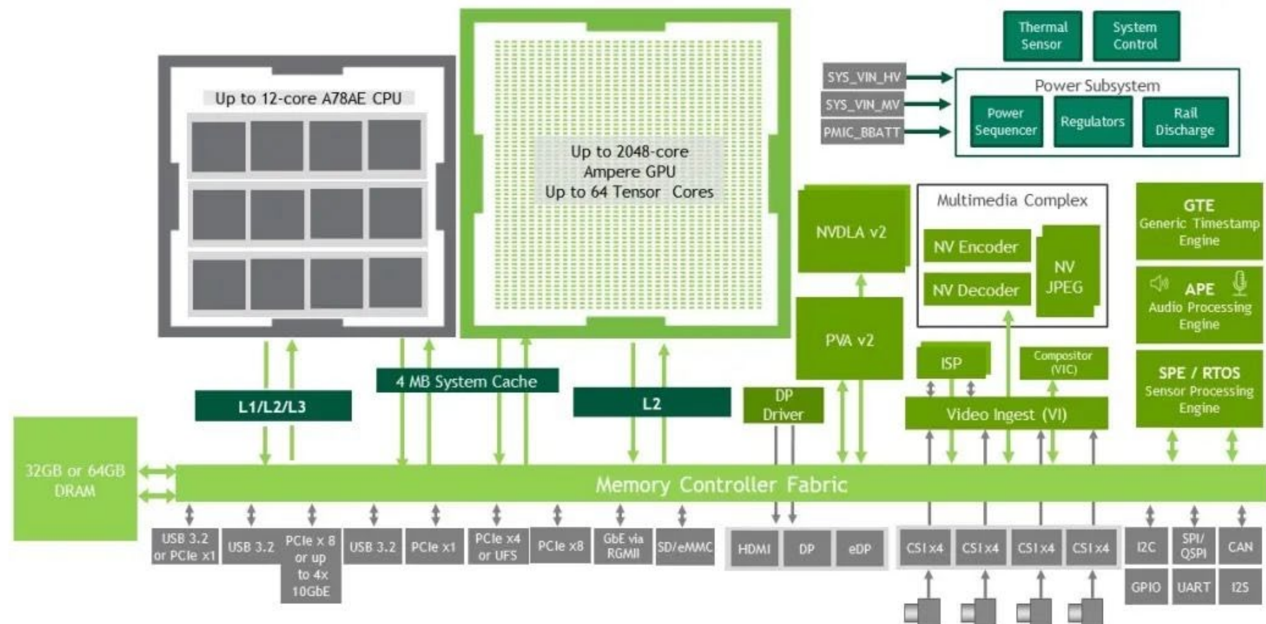


Figure 7 - Orin high-level block diagram (from [2]).

3.2.1 CPU Complex

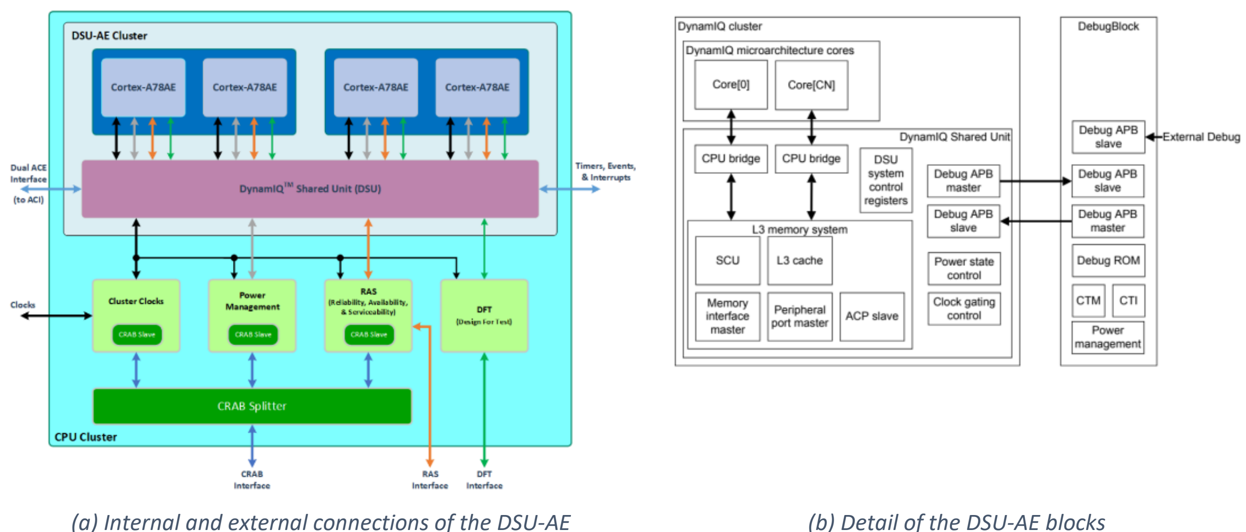
The high-level overview of the CPU Complex showing its CPU cores is presented in **Error! Reference source not found.Error! Reference source not found.**. We can see three different CPU clusters, each one implementing four Cortex-A78AE cores and a cluster-private L3 cache that is shared among the cores of each cluster.

3.2.1.1.1 DynamIQ™ Shared Unit (DSU-AE)

Each cluster in the CPU Complex comprises the DynamIQ Shared Unit (DSU-AE) that embeds a 2-MiB L3 Cache and the Snoop Control Unit (SCU), as seen in Figure 9b. The L3 and SCU are in charge of maintaining coherency between caches in the cores and L3. So, while the caches in the cores are private, the L3 is shared among the 4 cores in the cluster.

The DSU-AE provides the internal interfaces to the cores, as presented in Figure 9a where the connection to the four cores can be observed.

Figure 8 - Block diagram of the CPU Complex (from [2]).



(a) Internal and external connections of the DSU-AE *(b) Detail of the DSU-AE blocks*

Figure 9 – Block diagrams of the DSU-AE [6]

Comparators

Comparator logic is only enabled when the Cortex®-A78AE is operating in Lock-mode. There are two instances of each comparator, reporting on separate outputs. Delay flops are also associated with Lock-mode. The delay flops create the temporal diversity between the primary and redundant logic and align the comparison logic.

Snoop Control Unit (SCU)

The SCU contains buffers that can handle direct cache-to-cache transfers between cores without having to read or write data to the L3 cache. Cache line migration enables dirty cache lines to be

moved between cores, and there is no requirement to write back transferred cache line data to the L3 cache.

L3 cache

The cache is 16-way set associative with a 64-byte line length and a total size of 2MB. It is shared by all the cores in the cluster and supports stashing request from the ACE/CHI interface. However, if it is heavily loaded and does not have any free buffers, it drops the stash request.

The L3 cache data allocation policy changes depending on the pattern of data usage. Exclusive allocation is used when data is allocated in only one core. Inclusive allocation is used when data is shared between cores. The L3 cache implements two slices, each with a set of tag and data RAMs. Requests are allocated to a particular slice based on the address of the request. Splitting the cache into slices improves the bandwidth because the two slices can be accessed in parallel.

Cache capacity	Tag RAM				Data RAM	
	0	1	2	3	0	1
None	Off				Off	
¼	On	Off			On	Off
½	On		Off		On	Off
¾	On			Off	On	
All on	On				On	

The L3 supports the creation of groups of cache ways to partition and assign to individual processes. Cache partitioning ensures that processes do not dominate the use of the cache to disadvantage other processes.

L3 cache partitioning is achieved by partition scheme IDs and groups of cache ways, where:

- Each group contains four ways.
- Each group can either be assigned as private to one or more partition scheme IDs, or be left unassigned.
- Each unassigned group can be shared between all eight partition scheme IDs.

Each core in the cluster must be assigned to at least one of the eight partition scheme IDs. L3 cache accesses from a given core can allocate into:

- Any cache way that belongs to a group that is assigned as private to the partition scheme ID of this core.
- Any cache way that belongs to an unassigned group that is shared by the entire cluster.

L3 cache placement

The placement algorithm used for the L3 is undocumented. BSC investigated to empirically determine how data is placed into L3 sets. There are two main assumptions that were investigated:

- Modulo placement
- XOR placement, similar to the one used in the Cortex-A78AE L2 cache.

BSC prepared an experimental setup where, based on the initial assumption, more than 16 pieces of data would be placed into a single L3 set. If the assumption was correct, we would observe L3 cache misses, as the L3 only provides 16-ways per cache set.

The result showed that the formula used for determining the L3 set for placement is ([26:17] XOR [16:7]). Bits [5:0] are within the cache line offset (64 bytes) while bit 6 is used to select L3 cache slice. Figure 10 shows how memory address bits are used to access a cache lines in L2 and L3.

<table>
<tr><td>address</td><td>32</td><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="20">L3 placement</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="10">L3XORH</td><td colspan="10">L3XORL</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="16">L2 placement</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="8">XOR high</td><td colspan="8">XOR low</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="12">OS page</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="7">slice+line</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="6">line</td></tr>
</table>

Figure 10 – Address bits used for L2 and L3 cache placement.

3.2.2 GPU Cluster

The Orin SoC has an NVIDIA Ampere GPU with two Graphics Processing Clusters (GPCs). A GPC is the high-level hardware block with all the compute/graphics processing units for graphics-related computation, rasterization, rendering, Ray Tracing, pixel generation, etc. A high-level block diagram view is presented in Figure 11.

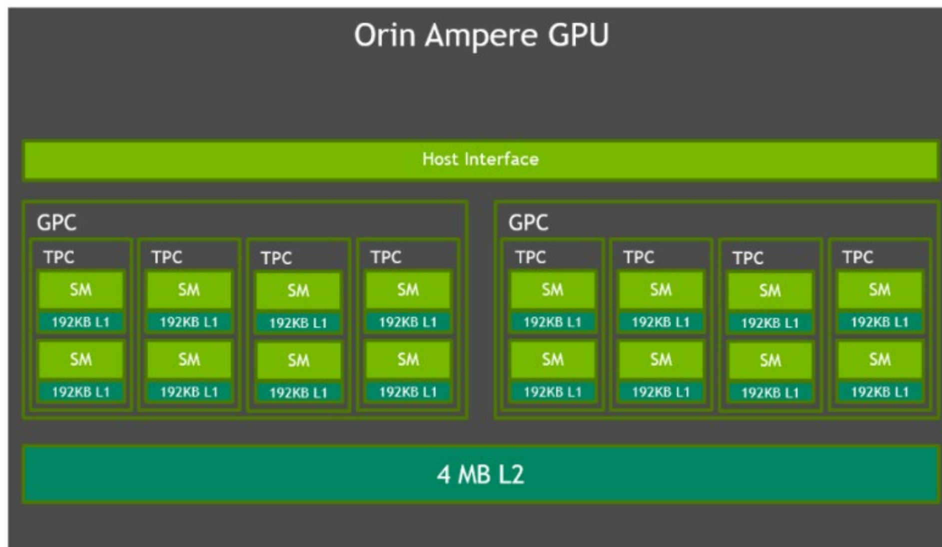


Figure 11 – Orin Ampere GPU high level block diagram (from [2])

The Ampere GPC contains the following components: A Raster Engine, four Texture Processing Clusters (TPCs), each consisting of 2 Streaming Multiprocessors (SMs), each with its own Ray Tracing (RT) core, and 1 PolyMorph Engine (PE). These elements can be seen in Figure 12.

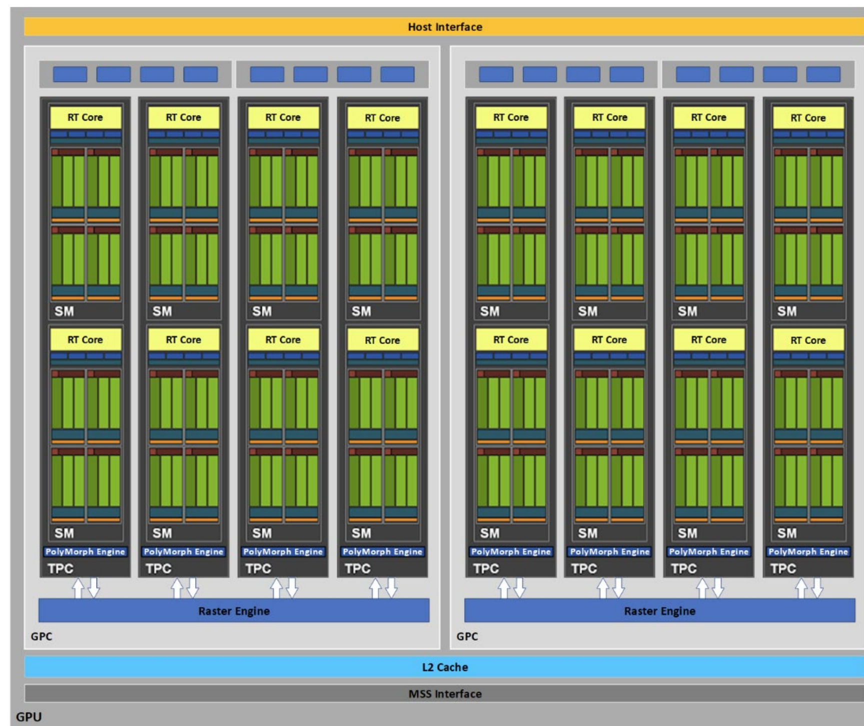


Figure 12 – Orin Ampere GPU detailed block diagram (from [2]).

Streaming Multiprocessor

The Ampere Streaming Multiprocessor has 128 CUDA cores. As shown in Figure 13, the SM is partitioned in four processing blocks, with each containing:

- 1 3rd-generation Tensor core,
- 1 64-KiB Register Files (in 16,384 x 32 organization),
- 1 Texture (TEX) unit,
- 1 L0 I-Cache, 1 Warp Scheduler, and
- 1 Dispatch (32 threads/clk) unit,

and all four sharing:

- 1 192-KiB for L1 Data Cache and Shared Memory, and
- 1 2nd-generation Ray Tracing (RT) core.

3.2.3 Interconnect

The interconnect is based on 2 technologies based on the AMBA protocol of ARM: AXI3 and AXI4. The system control fabric is based on AXI3, this means no QOS protocol is applied on them. On the other hand, the data fabric to access main memory (DDR) is based on AXI4 protocol.

3.2.4 Other features

The board is also equipped with a series of clusters focused on special tasks that are configured by Nvidia on their firmware. Those modules are: SPE Sensor processing engine; APE Audio processing engine; Safety Island in charge of the power control of the board; Boot power management; Real time cameras processing.

Figure 13 – Ampere Streaming Multiprocessor block diagram (from [2])

3.2.4.1 Input / Output (I/O)

GPC-DMA

There are 32 channels in GPC-DMA. A DMA channel can transfer a specified range of data between a memory address space (SysRAM or external memory) and an MMIO address space. A DMA channel can also transfer data between a memory address space and another memory address space (Mem-to-Mem copy). The DMA controller follows a simple round robin arbitration scheme between the channels, starting with channel 0. Each channel can have an independent burst transfer size programmed to one word, two words, four words, eight words, or 16 words. For Memory transfers, we only support two-words and 16-words bursts. There is a corresponding read/write buffer in the memory buffer manager for each channel. There is also a corresponding buffer for each channel on the peripheral side.

Continuous Mode

In continuous mode single buffer mode, software has two separate buffers that are maintained by software to emulate the hardware ping pong buffer. In this mode, software enables the DMA with the ping-buffer address and then reprograms the DMA with pong buffer after enabling the DMA. The DMA registers are shadowed (latched) every time upon entering the continuous cycle. The register programming can be done for the pong buffer either after enabling the channel (for the first reprogramming) or receiving an interrupt (for any subsequent reprogramming).

3.2.4.2 SCF and L4

The System Coherency Interconnect (SCF) connects the CPU Complex (CCPLEX) to the DRAM. It is in charge of maintaining coherency between the clusters and connecting them to the rest of the SoC. The SCF embeds a shared L4 cache that is shared among the three CPU clusters. The cache is also shared with the GPU, but it’s not fully coherent; coherency is one-way [7], so the GPU is able to read CPU cache, but not the other way around. In Figure 14 a view of how the SCF is connected to the clusters and to the interfaces to the rest of the SoC is shown.

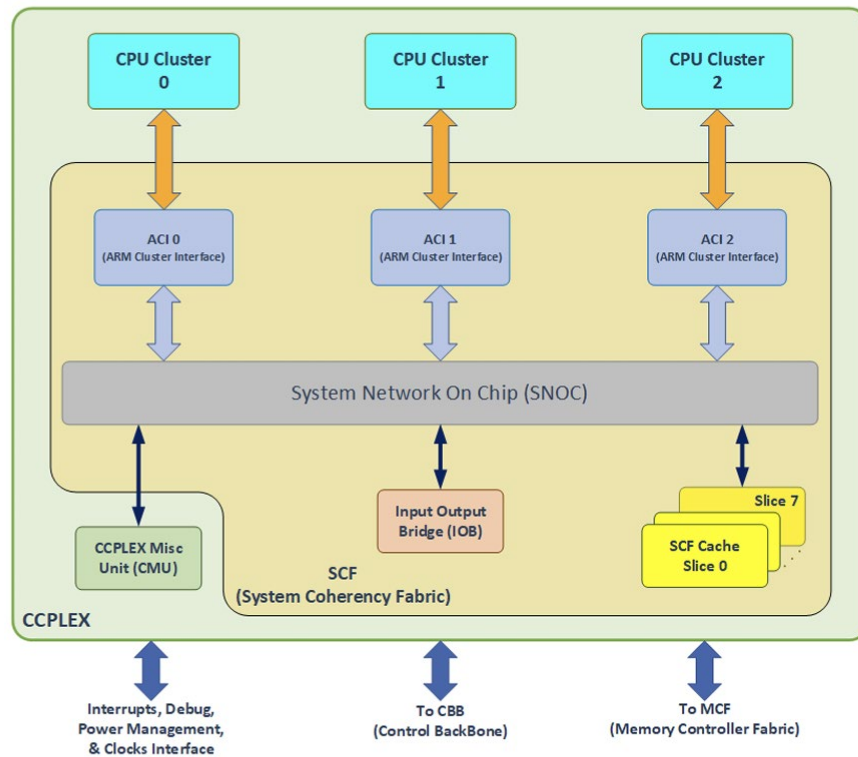


Figure 14 – Block diagram of the CCPLEX, including the SCF [6].

The L4 cache is set-associative and is partitioned in 8 512KB slices, for a total of 4MB. Memory requests from the CPU clusters are routed through the L4 cache and to the Memory Controller Fabric (MCF) as seen in Figure 14. Transactions to memory mapped Input/Output (MMIO) are routed through the Input Output Bridge (IOB).

3.2.5 Interference channel identification

In order to identify the sources of inter-process interference, BSC performed an analysis of the hardware architecture as presented in this Section. We studied possible bottlenecks in the design. We also interact with the use case providers to understand how their application uses the platform. The combination of the potential source of contention at hardware level and how the application actually uses the hardware, leads us to conclude the actual sources of contention.

3.2.5.1 Domains of resource sharing

We use the term resource sharing domain or resource sharing level to categorize how applications share hardware shared resources. It is worth noting that the hardware resources that are shared among cores and may become a contention point are related mainly to the datapath to memory:

- Intra CPU core. Private computing resources, private first level data and instruction caches, private L2, which ultimately request data to the shared L3 on miss.
- Intra CPU cluster. The SCU, including the shared L3. This component includes the snoop control and and L3 shared withing the CPU Cluster.
- Inter CPU cluster. The SCF, including the shared L4. This component centralises accesses to memory from the three CPU Clusters and the GPU.
- Accelerators. Accelerators can be shared among applications. This includes the GPU and specific accelerators like the DLA.

3.2.5.2 Usage of shared resources

We developed a questionnaire that we shared with the use case providers. The questionnaire included questions about the usage of shared resources in the Orin. The main conclusions from the questionnaire were the following:

1. End users are using 1 CPU cluster and might need to use several.
2. GPU is being used by one application only. Hence, the GPU is time shared and not space shared.
3. The accelerators are not being used and in case they are they will be time shared.

Hence our focus on the contention side goes on the intra CPU cluster and inter CPU cluster.

3.2.6 Empirical results

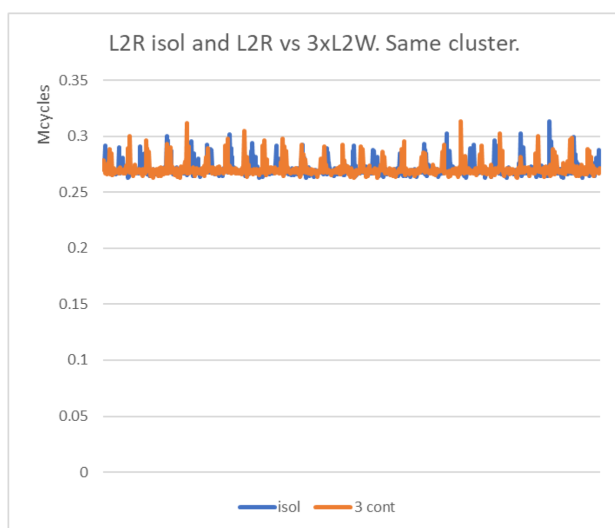
In order to empirically assess the impact of contention we develop a set of benchmarks that read/write to a given cache level. By running several of those benchmarks together we can assess whether they suffer contention in a given resource level The benchmarks are:

- L2R and L2W. Benchmarks that perform mainly reads (R) and (W) most of which hit in L2.
- L3R and L3W. Benchmarks that perform mainly reads (R) and (W) most of which hit in L3.
- L4R and L4W. Benchmarks that perform mainly reads (R) and (W) most of which hit in L4.
- MEMR and MEMW. Benchmarks that perform mainly reads (R) and (W) most of which do not hit in any cache level and go to memory.

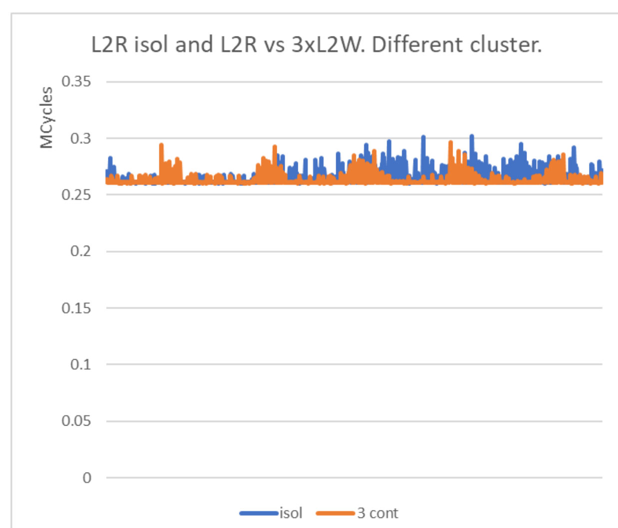
It is noted that when a benchmark hits in a cache level it is implicitly missing the lower cache levels.

3.2.6.1 L2

In this first experiment we run a L2R benchmark against 3 instances of the L2W benchmarks. Each benchmark executes in one of the cores of a given cluster. Figure 15**Error! Reference source not found.** below shows the execution time of the L2R benchmark. As it can be seen in Figure 15 it suffers no significant increase in the execution time when running in multicore vs when running in isolation, providing evidence that the L2 are private per core. Comparing figures Figure 15a and Figure 15b we also appreciate no difference between running all contenders in the same cluster or in a different one, due to L2 being private per core.



(a) Execution time of L2R in isolation and with contenders running on the same cluster.



(b) Execution time of L2R in isolation and with contenders running on a different cluster.

Figure 15 - Execution time in ms of the L2R benchmark in isolation (blue) and contending with 3 instances of L2W (orange).

Figure 16 shows the L2 miss rate distribution for the L2R benchmark running in isolation and when with 3 contenders running L2W. It can be seen that the miss rate remains very low in both cases (mostly sub 1%), and with a very similar distribution, proving that contenders do not increase miss rate and hence that the L2 caches are private per core.

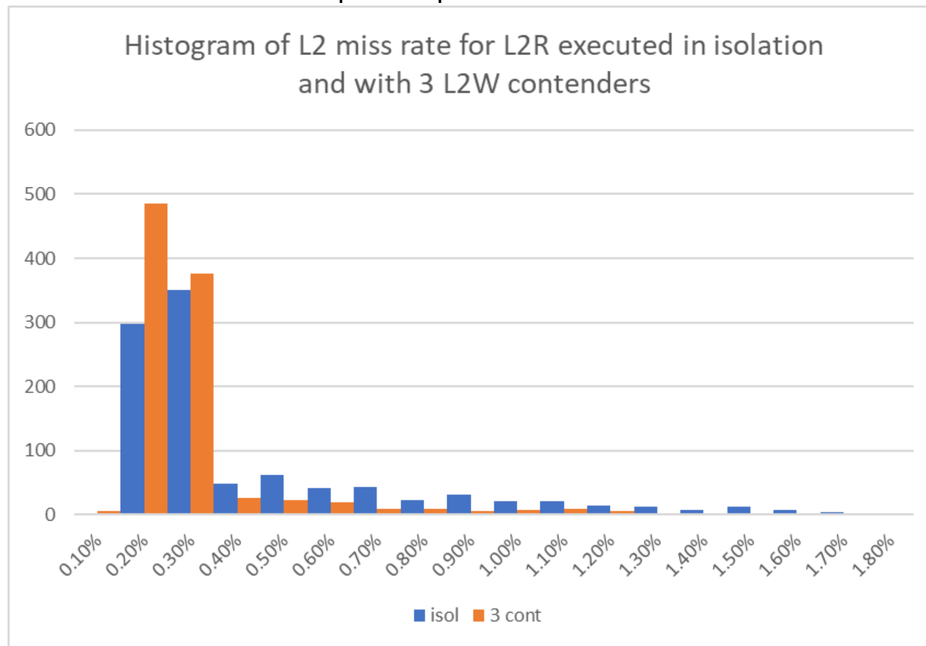
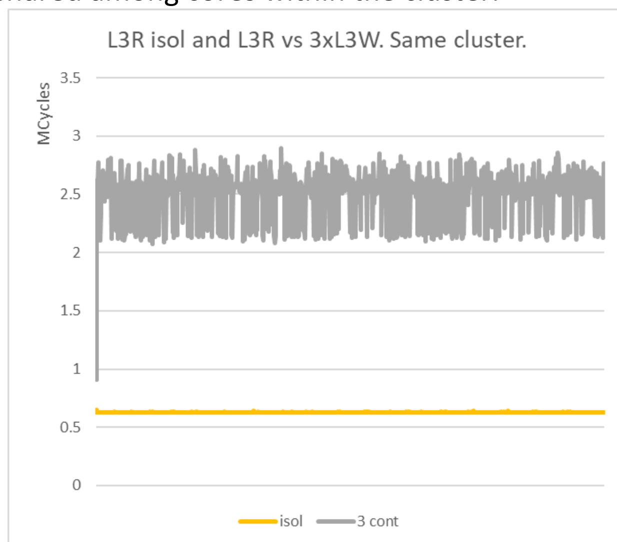


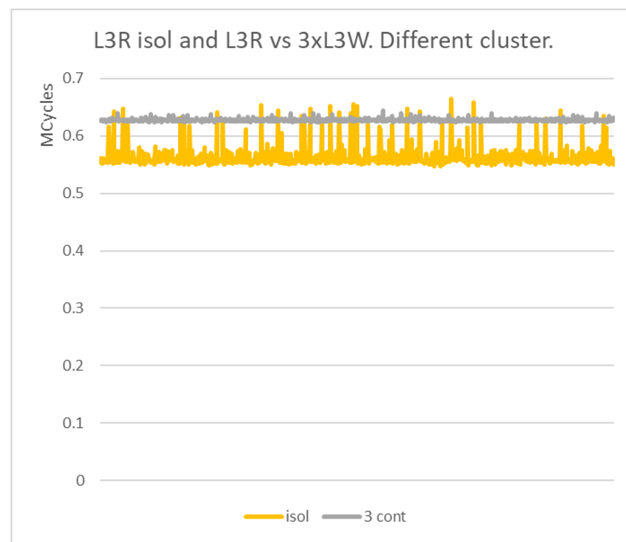
Figure 16 - Histogram with the distribution of the miss rate across different executions of L2R, in isolation (blue) and with 3 contenders running L2W (orange). This includes execution in a single cluster and in multiple clusters.

3.2.6.2 L3

In this experiment we run a L3R benchmark against 3 instances of L3W benchmarks. Each benchmark executes in one of the cores of a given cluster. Figure 17 below shows the execution time of the L3R benchmark. As it can be seen in Figure 17b it suffers no significant increase in the execution time when running in multicore vs when running in isolation, while in Figure 17a a very significant difference can be seen. This provides evidence that the L3 are private per cluster but shared among cores within the cluster.



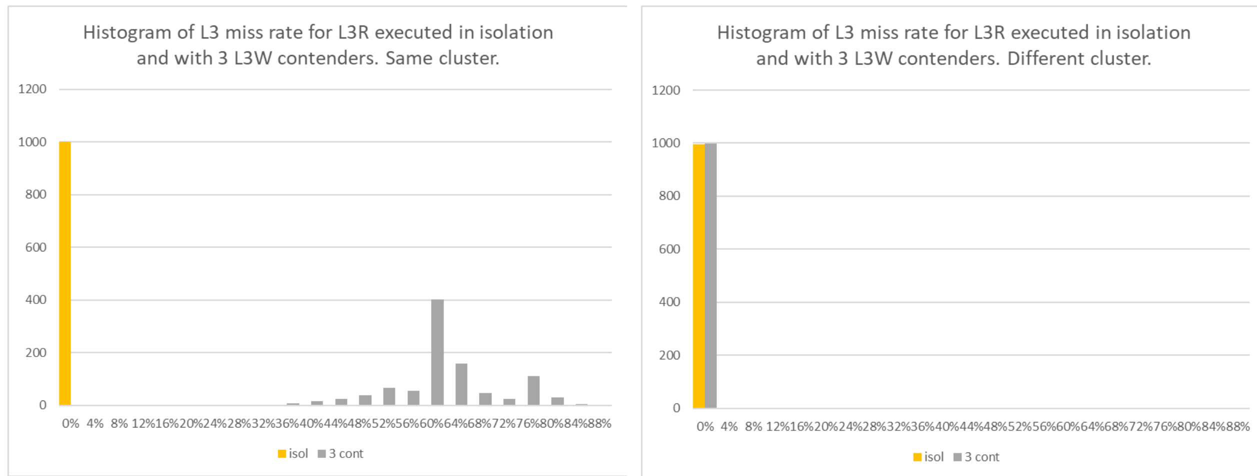
(a) Execution time of L3R in isolation and with contenders running on the same cluster.



(b) Execution time of L3R in isolation and with contenders running on a different cluster.

Figure 17 – Execution time in ms of the L3R benchmark in isolation (yellow) and contending with 3 instances of L3W (grey).

Figure 18 shows the L3 miss rate distribution for the L3R benchmark when running in isolation and with 3 contenders running L3W. In Figure 18a, it can be appreciated how multicore execution in a single cluster causes a high amount of L3 cache misses, as the contenders compete with the task under analysis for the shared L3. In Figure 18b we can see how L3 miss rate does not vary between execution in isolation and with contenders when the latter run in a different cluster, proving that the L3 is private per cluster.



(a) Execution time of L3R in isolation and with contenders running on the same cluster.

(b) Execution time of 3R in isolation and with contenders running on the different cluster.

Figure 18 – Histogram with the distribution of the miss rate across different executions of L3R, in isolation (yellow) and with 3 contenders running L3W (grey).

3.2.6.3 L4

In this experiment we run a L4R benchmark against 3 instances of L4W benchmarks. Each benchmark executes in one of the cores of a given cluster. Figure 19 below shows the execution time of the L4R benchmark. As it can be seen in Figure 19b, in average the execution time is not greatly affected by multicore execution when contenders are running in a different cluster. This is due the L3 being the bottleneck in the memory hierarchy, so we only appreciate an increase in contention when running in the same cluster, as seen in Figure 19a. This is true for this scenario, where the GPU is not being used and hence the L4 cache is only used by cores, but the L4 may become the bottleneck when the GPU is in use.

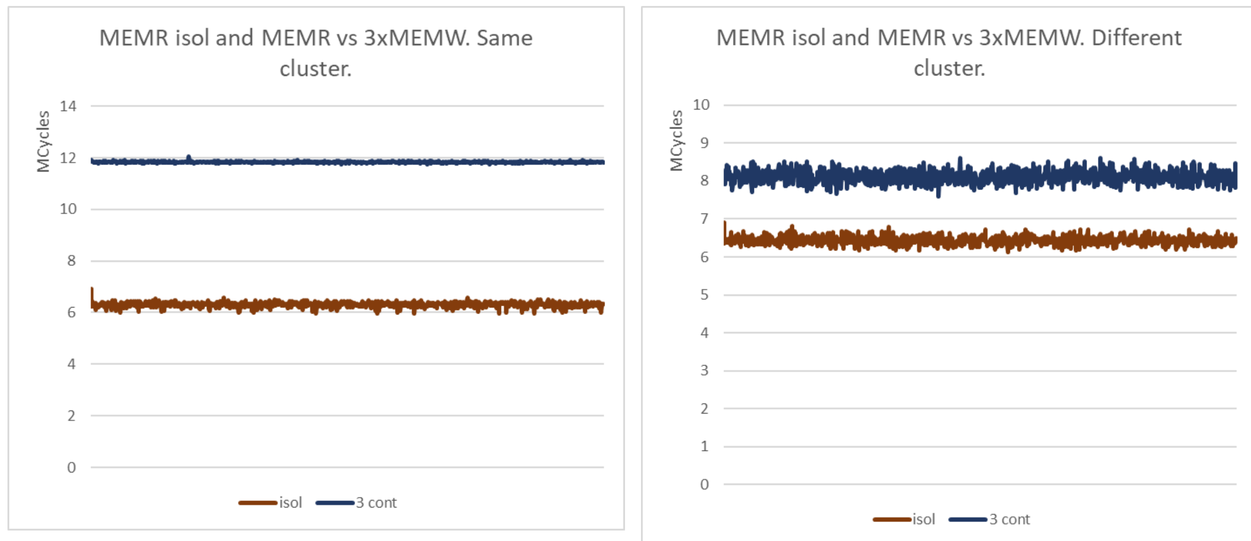
(a) Execution time of L4R in isolation and with contenders running on the same cluster.

(b) Execution time of L4R in isolation and with contenders running on the different cluster.

Figure 19 – Execution time in ms of the L4R benchmark in isolation (blue) and contending with 3 instances of L4W (green).

3.2.6.4 MEM

In this experiment we run a MEMR benchmark against 3 instances of MEMW benchmarks. Each benchmark executes in one of the cores of a given cluster. Figure 20 below shows the execution time of the MEMR benchmark. As it can be seen in Figure 20a, the execution time is greatly affected by contenders running in the same cluster. Meanwhile, Figure 20b shows a smaller difference in execution time when contenders run in a different cluster.



(a) Execution time of MEMR in isolation and with contenders running on the same cluster.

(b) Execution time of MEMR in isolation and with contenders running on the different cluster.

Figure 20 – Execution time in ms of the MEMR benchmark in isolation (brown) and contending with 3 instances of MEMW (blue).

3.3 Technological assessment

In accordance with the task main objectives, we have been performing an in-depth analysis of the hardware platform to understand and identify the major sources of timing interference at the software and hardware level. At the software level, we have been focusing on the SAFEXPLAIN software stack and adapted our analysis, when possible, to address the contribution of the different software layer (operating system, ROS2 layer, and SAFEXPLAIN middleware). At the hardware level, we complemented the available documentation on the NVIDIA Orin with results from an ad-hoc test campaign, supporting hypothesis defined building on long-standing hardware and software expertise.

With the work done until month 18, we have already captured the main objectives of this task and produced a detailed analysis of the hardware platform, covering both functional and non-functional aspects. We have also identified the sources of timing interferences (interference channels) at both software and hardware level, which serves as an input to T4.3 and WP2.

Delivered tool and positioning in the SAFEXPLAIN stack
The result of this task is not a tool but consolidated knowledge on which further project-level decisions are taken. As such, it does not occupy concrete position in the SAFEXPLAIN stack.

Intra-WP dependencies
Within the scope of WP4, the obtained results are fundamental inputs to other tasks.

T4.2: The identification of the main sources of hardware interference steers the selection of those events and monitors that are critically relevant for timing and interference analysis. It is also important that both tasks work consistently towards achieving WP4 objectives.

T4.3: The sources of timing interference and the mitigation actions identified in T4.1 are in fact determining the strategy to follow for timing and interference analysis, where the particular solution for supporting isolation among application must be modelled.

Inter-WP contribution and alignment

The contributions of this task are also relevant in the scope of other work packages. The analysis results are supporting many of the deployment decisions that are taken when tailoring and bringing software and concepts to the actual target.

WP2: The sources of hardware interference and segregation solutions are critical aspects to be considered in the definition of the FUSA architecture and in particular of the Safety Patterns, where a concrete mapping of applications/components to the platform is required. Task 2.4 is explicitly defined to favor the transition from FUSA concepts to corresponding deployment configurations.

WP5: The hardware analysis results can be exploited by use case providers to take informed decisions on where and how to deploy the applications.

The alignment with other work packages, and WP2 in particular, is guaranteed by the continuous interaction between the work packages.

T4.1 Next steps

Task T4.1 is running until m24. The hardware and software analyses have already produced a deep understanding of the platform, providing a more than appropriate level of details to execute the other project tasks. We have no relevant component or major area of interest in the hardware left to be analysed. In the next project period, the task will be capturing any emerging requirements from other tasks and work-packages, with bearing on hardware and low-level software aspect. The task will also support WP2 for incremental deployment of Safety Patterns.

4 Observability Channels (T4.2)

Information on low-level hardware events is typically made available in modern Commercial Off The Shelves (COTS) platform via specialized hardware support provided by more or less complex Platform Monitoring Units (PMU). This task identifies and tests the platform monitors (hardware event monitors or HEMs) providing information about timing behavior, multicore timing interference, shared resource usage, and many more metrics. In the scope of WP4 observability is fundamental to run other tasks. Among all HEMs, we then make a selection of those needed to properly predict the timing behaviour of running applications, as needed by the statistical timing estimation techniques in T4.3. Another key activity of this task is producing a monitor configuration support library integrated in the software stack to collect as many measurements as needed for relevant monitors for timing prediction in T4.3. We refer to this library as *PMULib* (Performance Monitoring Unit Library).

In this section, we start with identifying the need for observability in SAFEXPLAIN. We then identify the set of HEMs that are available in the platform. We introduce PMULib, a fundamental tool in SAFEXPLAIN to make HEMs accessible to all other tasks on top of SAFEXPLAIN stack. We then make an analysis of the HEMs accuracy and single out those HEMs that better help tracking timing-related aspects, including multicore contention.

4.1 PMU and HEM analysis

Observability is a fundamental property embedded critical platform must fulfil in order to enable the collection of evidence on the behavior of the system at execution and exploit such information to perform various types of analysis. Different hardware platforms come with different observability support: depending on the debug support for the different hardware units, the degrees of observability may largely vary across products.

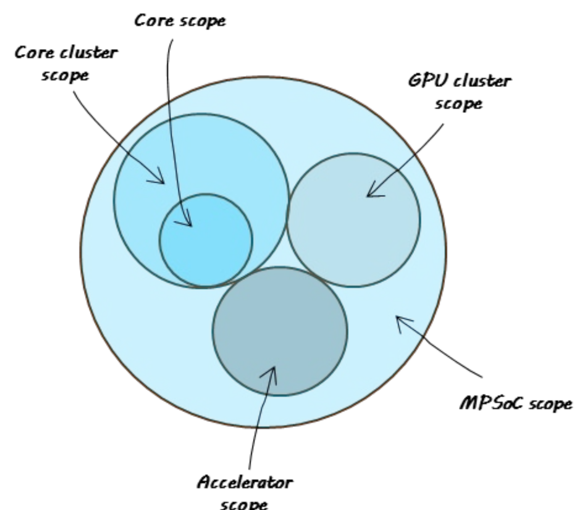


Figure 21 - Observability scopes.

When considering the Orin platform, we were aware of the observability support granted by ARM Cortex CPUs, the main interconnect, as well as in the previous family of AGX platform (Jetson AGX). Due to the complexity and heterogeneity of the platform, we had first to understand the different support available in each of the different scopes in the Orin: CPU cluster, GPU cluster, Accelerators, and MPSoC levels (see Figure 21).

4.1.1 Use of hardware-level information in SAFEXPLAIN

The set of events happening at the hardware level, and gathered through HEMs, has been increasingly considered as a valuable source of information to decipher the deepest details of hardware design and ultimately of software execution. SAFEXPLAIN fully recognizes the usefulness of hardware-level information and identified the need to collect this type of information and make it available to the tools and analyses designed and developed in the project.

We identified different aspects and activities in SAFEXPLAIN that justify the collection of hardware events:

- *Hardware analysis:* Collecting empirical evidence to support reverse engineering activity on the platform, which is often required to corroborate or clarify hardware features and operational details that are loosely documented in the officially available documentation. This cover, for example, the exact ID of hardware modules or the assessment of a partitioning mechanism. This objective is therefore instrumental to **T4.1** goals.
- *System monitoring*: Supporting on-line monitoring of low-level behaviour of target applications. The library allows to track relevant events, for example, to measure and limit the impact of multicore timing interference on a software partition. This objective is therefore instrumental to **T4.1** goals and ultimately to **WP2 (T2.4)** goals.
- *Timing characterization*: Supporting timing characterization by enabling the collection of timing information while the program executes and use it later to reason on the application timing behavior. Timing information, including execution time and other resource usage metrics, are fundamental information for measurement-based timing analysis approaches and SAFEXPLAIN is indeed focusing on statistical measurement-based methods for the analysis of complex AI-based systems. This objective is therefore instrumental to **T4.3** goals.

These aspects motivated the analysis of observability support on the target platform and, subsequently, the design and development of a user-level library to make hardware events easily accessible to SAFEXPLAIN tools and on top of the project hardware and software stack.

As the first step in the overall strategy followed to adequately exploit platform-level observability we analysed the actual support offered by the target hardware, which implies understanding the support on the many hardware modules and the complementary support offered by module-specific monitoring units and monitoring libraries. The support, in fact, depends on the manufacturer design but also on the specific model as support may largely vary even within the same family of products. This is the case, for example, of configuration and monitoring control in advanced accelerator. We assessed the offered support against the planned use in the scope of the project. We concluded that, although the support is not the same as that offered in high-end platform for mainstream (i.e. not embedded) market, all SAFEXPLAIN scopes are sufficiently covered by the available support.

Moreover, we also had to ensure the sources of information was ultimately trustworthy, in terms of correctness and accuracy. This activity consisted in selecting test scenario with well-known behavior in terms of hardware events and assessing HEMs observed values against expected ones.

4.2 HEMs identification

There are two main hardware blocks that expose HEMs that can be used. The A78 [3] cores and the SCF [8].

- The HEMs of the A78 focus on cpu, DL1 caches, L2 and L3. They are architectural HEMs described in the architecture manual [3]. They can be accessed with very low latency.
- The HEMs of the System Coherency Fabric (SCF), an uncore component connecting the CPU Complex and the GPU Complex to the Memory Controller Fabric. These HEMs are poorly documented and take a very high latency to be accessed.

We analysed a total of 130 HEMs and developed a taxonomy organizing the best candidates in 8 classes:

- cpu-pipeline, counting events generated at core level by the pipeline
- bus-memory accesses, counting accesses to the main bus
- TLB, grouping different types of events involving DL1 (data cache), IL1 (instruction cache), and L2 TLBs (Transaction Lookaside Buffer)
- IL1, grouping different types of accesses to the instruction cache
- DL1, grouping different types of accesses to the data cache
- L2, grouping different types of accesses to private L2 cache
- L3, grouping different types of accesses to cluster-shared L3 cache
- SCF, grouping different types of accesses to shared SCF (including L4 cache events)

Using this taxonomy, we selected a series of HEMs as most promising. Their selection is based on their expected usefulness to predict the impact the software could have on the shared resources.

As it can be seen the HEMs in the subsections below are related to the number of accesses and misses to each cache level, the accesses to shared resources in the data path, and to program execution. Those HEMs are the ones that most closely could predict the amount of pressure put on the hardware shared resources by the program being executed.

4.2.1 CPU-pipeline HEMs

Mnemonic	Description	ID
INST_RETIRED	Instruction architecturally executed. This event counts all retired instructions, including those that fail their condition check.	0x8
BR_PRED	Predictable branch speculatively executed. This event counts all predictable branches.	0x12
LD_SPEC	Operation speculatively executed, load	0x70
ST_SPEC	Operation speculatively executed, store	0x71

4.2.2 Instruction cache HEMs (L1I)

Mnemonic	Description	ID
L1I_CACHE_REFILL	L1 instruction cache refill. This event counts any instruction fetch which misses in the cache. The following instructions are not counted: • Cache maintenance instructions • Non-cacheable accesses	0x1
L1I_CACHE	L1 instruction cache access or L0 Macro-op cache access. This event counts any instruction fetch which accesses the L1 instruction cache or L0 Macro-op cache. The following instructions are not counted: • Cache maintenance instructions • Non-cacheable accesses	0x14
L1I_CACHE_LMISS	L1 instruction cache long latency miss	0x4006

4.2.3 Data cache HEMs (L1D)

Mnemonic	Description	ID
L1D_CACHE_REFILL	L1 data cache refill. This event counts any load or store operation or page table walk access which causes data to be read from outside the L1, including accesses which do not allocate into L1. The following instructions are not counted: • Cache maintenance instructions and prefetches • Stores of an entire cache line, even if they make a coherency request outside the L1 • Partial cache line writes which do not allocate into the L1 cache • Non-cacheable accesses.	0x3

	This event counts the sum of L1D_CACHE_REFILL_RD and L1D_CACHE_REFILL_WR.	
L1D_CACHE	L1 data cache access. This event counts any load or store operation or page table walk access which looks up in the L1 data cache. In particular, any access which could count the L1D_CACHE_REFILL event causes this event to count. The following instructions are not counted: • Cache maintenance instructions and prefetches • Non-cacheable accesses This event counts the sum of L1D_CACHE_RD and L1D_CACHE_WR.	0x4
L1D_CACHE_WB	L1 data cache Write-Back. This event counts any write-back of data from the L1 data cache to L2 or L3. This counts both victim line evictions and snoops, including cache maintenance operations. The following instructions are not counted: • Invalidations which do not result in data being transferred out of the L1 • Full-line writes which write to L2 without writing L1, such as write streaming mode	0x15
L1D_CACHE_LMISS_RD	L1 data cache long-latency miss	0x39
L1D_CACHE_RD	L1 data cache access, read. This event counts any load operation or page table walk access which looks up in the L1 data cache. In particular, any access which could count the L1D_CACHE_REFILL_RD event causes this event to count. The following instructions are not counted: • Cache maintenance instructions and prefetches • Non-cacheable accesses	0x40
L1D_CACHE_WR	L1 data cache access, write. This event counts any store operation which looks up in the L1 data cache. In particular, any access which could count the L1D_CACHE_REFILL_WR event causes this event to count. The following instructions are not counted: • Cache maintenance instructions and prefetches • Non-cacheable accesses	0x41
L1D_CACHE_REFILL_RD	L1 data cache refill, read. This event counts any load operation or page table walk access which causes data to be read from outside the L1, including accesses which do not allocate into L1. The following instructions are not counted: • Cache maintenance instructions and prefetches • Non-cacheable accesses	0x42
L1D_CACHE_REFILL_WR	L1 data cache refill, write. This event counts any store operation which causes data to be read from outside the L1, including accesses which do not allocate into L1. The following instructions are not counted: • Cache maintenance instructions and prefetches • Stores of an entire cache line, even if they make a coherency request outside the L1 • Partial cache line writes which do not allocate into the L1 cache • Non-cacheable accesses	0x43
L1D_CACHE_REFILL_INNER	L1 data cache refill, inner. This event counts any L1 data cache linefill (as counted by L1D_CACHE_REFILL) which hits in the L2 cache, L3 cache or another core in the cluster.	0x44
L1D_CACHE_REFILL_OUTER	L1 data cache refill, outer. This event counts any L1 data cache linefill (as counted by L1D_CACHE_REFILL) which does not hit in the L2 cache, L3 cache or another core in the cluster, and instead obtains data from outside the cluster.	0x45
L1D_CACHE_WB_VICTIM	L1 data cache write-back, victim	0x46
L1D_CACHE_WB_CLEAN	L1 data cache write-back cleaning and coherency	0x47
L1D_CACHE_INVAL	L1 data cache invalidate	0x48

4.2.4 L2 cache HEMs (L2)

Mnemonic	Description	ID
L2D_CACHE	L2 unified cache access. This event counts any transaction from L1 which looks up in the L2 cache, and any write-back from the L1 to the L2. Snoops from outside the core and cache maintenance operations are not counted.	0x16

L2D_CACHE_REFILL	L2 unified cache refill. This event counts any Cacheable transaction from L1 which causes data to be read from outside the core. L2 refills caused by stashes and prefetches that target this level of cache, should not be counted.	0x17
L2D_CACHE_WB	L2 unified cache write-back. This event counts any write-back of data from the L2 cache to outside the core. This includes snoops to the L2 which return data, regardless of whether they cause an invalidation. Invalidations from the L2 which do not write data outside of the core and snoops which return data from the L1 are not counted.	0x18
L2CACHE_INV	L2 unified cache invalidate	0x58
L2D_CACHE_LMISS_RD	L2 unified cache long latency miss	0x4009

4.2.5 L3 cache HEMs (L3)

Mnemonic	**Description**	**ID**
L3D_CACHE_ALLOCATE	Attributable L3 unified cache allocation without refill. This event counts any full cache line write into the L3 cache which does not cause a linefill, including write-backs from L2 to L3 and full-line writes which do not allocate into L2.	0x29
L3D_CACHE_REFILL	Attributable L3 unified cache refill. This event counts for any cacheable read transaction returning data from the SCU for which the data source was outside the cluster. Transactions such as ReadUnique are counted here as 'read' transactions, even though they can be generated by store instructions. Prefetches and stashes that target the L3 cache are not counted.	0x2a
L3D_CACHE	Attributable L3 unified cache access. This event counts for any cacheable read transaction returning data from the SCU, or for any cacheable write to the SCU.	0x2b
L3_CACHE_RD	L3 cache read	0xa0
L3D_CACHE_LMISS_RD	L3 unified cache long latency miss	0x400b

4.2.6 Bus-memory HEMs

Mnemonic	**Description**	**ID**
BUS_ACCESS	Bus access. This event counts for every beat of data transferred over the data channels between the core and the SCU. If both read and write data beats are transferred on a given cycle, this event is counted twice on that cycle. This event counts the sum of BUS_ACCESS_RD and BUS_ACCESS_WR.	0x19
BUS_ACCESS_RETRY	Bus access write. This event counts for every beat of data transferred over the write data channel between the core and the SCU.	0x61
MEM_ACCESS	Data memory access. This event counts memory accesses due to load or store instructions. The following instructions are not counted: • Instruction fetches • Cache maintenance instructions • Translation table walks or prefetches This event counts the sum of MEM_ACCESS_RD and MEM_ACCESS_WR.	0x13
MEM_ACCESS_RD	Data memory access, read. This event counts memory accesses due to load instructions. The following instructions are not counted: • Instruction fetches • Cache maintenance instructions • Translation table walks • Prefetches	0x66
MEM_ACCESS_WR	Data memory access, write. This event counts memory accesses due to store instructions. The following instructions are not counted: • Instruction fetches • Cache maintenance instructions • Translation table walks • Prefetches	0x67
REMOTE_ACCESS	Access to another socket in a multi-socket system	0x31

4.2.7 TLB HEMs

Mnemonic	Description	ID
L1I_TLB_REFILL	L1 instruction TLB refill. This event counts any refill of the instruction L1 TLB from the L2 TLB. This includes refills that result in a translation fault. The following instructions are not counted: • TLB maintenance instructions This event counts regardless of whether the MMU is enabled.	0x2
L1D_TLB	L1 data TLB access. This event counts any load or store operation which accesses the data L1 TLB. If both a load and a store are executed on a cycle, this event counts twice. This event counts regardless of whether the MMU is enabled.	0x25
L1I_TLB	L1 instruction TLB access. This event counts any instruction fetch which accesses the instruction L1 TLB. This event counts regardless of whether the MMU is enabled.	0x26
L2TLB_REFILL	Attributable L2 unified TLB refill. This event counts on any refill of the L2 TLB, caused by either an instruction or data access. This event does not count if the MMU is disabled.	0x2d
L2TLB_REQ	Attributable L2 unified TLB access. This event counts on any access to the L2 TLB (caused by a refill of any of the L1 TLBs). This event does not count if the MMU is disabled.	0x2f
L2TLB_RD_REFILL	L2 unified TLB refill, read	0x5c
L2TLB_WR_REFILL	L2 unified TLB refill, write	0x5d
L2TLB_RD_REQ	L2 unified TLB access, read	0x5e
L2TLB_WR_REQ	L2 unified TLB access, write	0x5f

4.2.8 SCF HEMs

Mnemonic	Description	ID
SCF_BUS_ACCESS	Bus accesses in the SCF	0x10190
SCF_BUS_ACCESS_RD	Read bus accesses in the SCF	0x10600
SCF_BUS_ACCESS_WR	Write bus accesses in the SCF	0x10610
SCF_CACHE_ALLOCATE	SCF L4 cache allocates	0x10f00
SCF_CACHE_REFILL	SCF L4 cache refills	0x10f10
SCF_CACHE	SCF L4 cache accesses	0x10f20
SCF_CACHE_WB	SCF L4 cache write-backs	0x10f30

4.3 Hardware Event Monitors PMULib

Once we identified the HEMs that could be used to track contention, our next step relates to developing a library to read it.

4.3.1 PMULib

During this first period, BSC worked on the design and implementation of **PMULib** a platform specific lightweight library to configure and extract (collect) values from the different event monitors in the Orin platform, exploiting the available support at the different scopes. PMULib is a fundamental tool for supporting the required degree of platform observability necessary to support diverse elements in the SAFEXPLAIN technological stack (see Figure 22):

- Collecting empirical evidence to support reverse engineering activity on the platform, which is often required to corroborate or clarify hardware features and operational details that are loosely documented in the officially available documentation. This cover, for

example, the exact ID of hardware modules or the assessment of a partitioning mechanism. This objective is therefore instrumental to **T4.1** goals.

- Supporting on-line monitoring of low-level behaviour of target applications. The library allows to track relevant events, for example, to measure and limit the impact of multicore timing interference on a software partition. This objective is therefore instrumental to **T4.1** goals and ultimately to **WP2 (T2.4)** goals.
- Supporting timing characterization by enabling the collection of timing information while the program executes and use it later to reason on the application timing behavior. Timing information, including execution time and other resource usage metrics, are fundamental information for measurement-based timing analysis approaches and SAFEXPLAIN is indeed focusing on statistical measurement-based methods for the analysis of complex AI-based systems. This objective is therefore instrumental to **T4.3** goals.

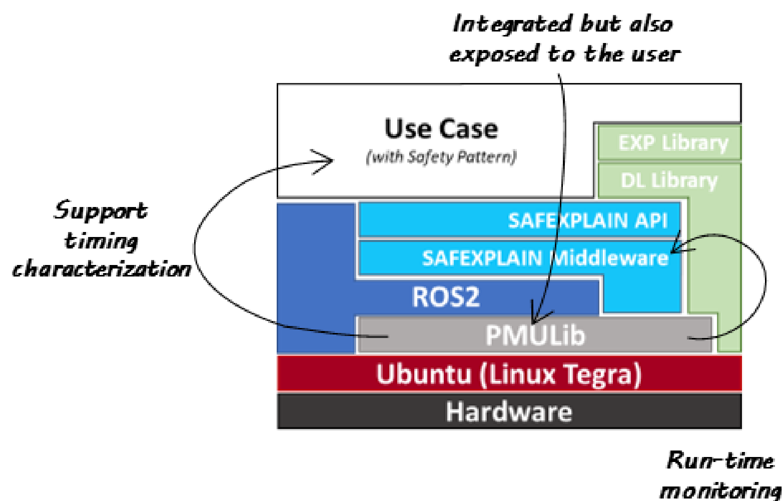


Figure 22 - PMULib role and interactions.

The provided library greatly reduces the complexity for the engineers and analysis tools for retrieving platform-level information. Further, the library is fundamental to increase modularity and automation of the proposed V&V solutions.

PMULib works at very low level, in conjunction with the OS or even at lower level (see Figure 22). For this reason, the PMULib includes a platform specific layer, in this case tailored to the Orin, and a more generic layer, which has been modelled to allow future portability and to adapt to the generic Linux-based software stack used in SAFEXPLAIN.

We are following an incremental strategy in the design and development of PMULib. The main efforts in this first phase of the project have been devoted to secure extensive support to the events related to the core clusters and the interconnect (shared among clusters and accelerators).

We offer a lightweight C interface and implementation of PMULib as the main goal was to integrate it on top the hardware/OS layer to make it compatible with the SAFEXPLAIN Middleware layer, which will be deeply discussed in Section 6.1. Currently, PMULib can also be exploited standalone, outside the middleware framework but its integration in the middleware allows for better automation of the supported analysis processes.

We currently offer no consolidated support to the GPU cluster as the NVIDIA libraries for this platform do not support fine grain control of the GPU debug modules. We are anyway guaranteeing observability at the boundaries of GPU workloads and on the interconnect. The restricted scope seems to be reasonable also in considerations of the deployment scenario

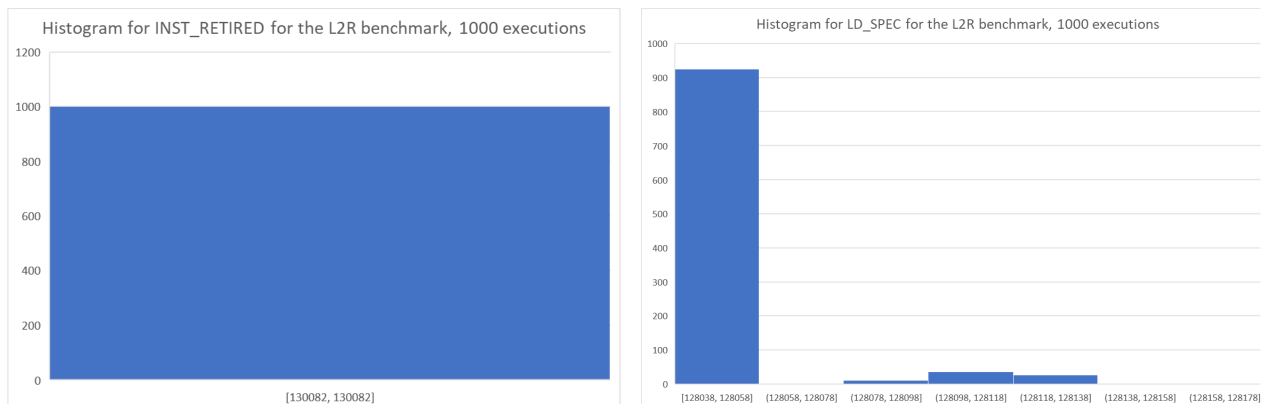
observed in the case studies. In any case, alternative and complementary observability solutions are currently under assessment.

4.3.2 PMULib Validation and accuracy

In order to validate the correctness of the PMULib we perform some experiments in which we compared, for some small code snippets, the expected value for observed HEMs with respect to the values observed with the HEMs and read with the PMULib. Our focus is on the so called functional counters like instruction count, load count, and store count that are less subject to variability across runs.

In particular, we focus on the L2R benchmark which we expect to have 130,000 instructions out of which 128,000 are reads (load operations). Some extra instructions and load operations are expected as part of the code setting the initial conditions after the PMULib is called, but with very small contribution in number (less than 1%).

We have run the benchmark a thousand times and collected counters 0x8 (INST_RETIRED) and 0x70 (LD_SPEC), which we expect to closely match the aforementioned numbers.



(a) Histogram for the INST_RETIRED showing no variability

(b) Histogram for LD_SPEC showing small variability

Figure 23 – Accuracy of PMULib, shown as histograms for 1000 executions of the L2R benchmark.

In Figure 23a, we can see no variability for the INST_RETIRED counter, which suggests the PMULib does not introduce noise, or if introduced, it is constant. We also see that the deviation from the expected value is just 0.06%, a very small amount.

In Figure 23b we observe minimal variability in LD_SPEC for more than 90% of accesses, but we observe a few accesses that are up to 0.13% higher than the expected value. Again, we see minimal deviation from the expected value, suggesting high accuracy and small variability in both the counters and the library.

The higher variability in LD_SPEC results probably arises from the fact that it counts speculative LD instructions, not retired LD instructions. Still the observed variability is minimal and well within reasonable margins (0.13%).

4.4 SCF HEMs overhead

One of the main insights when using our library is that we detected that using SCF counters carries a very big overhead in terms of execution time.

In the table below we present the increase in execution time of the eight benchmarks presented in Section 3.2.6. The baseline is the execution time (Wall time) of the benchmarks when reading core counters.

OH wrt core cnt

Cpu	1cnt	6cnt
L2R	36.4	646.8
L2W	43.5	779.7
L3R	22.8	394.2
L3W	26.8	479.0
L4R	11.0	192.4
L4W	10.8	190.7
MEMR	1.5	26.4
MEMW	5.5	99.4

Although all SCF counters seem to report values very close to our expectation as seen in Figure 24, the Wall time is increased up to 44x when reading a single SCF counter, up to 780 when reading 6 counters.

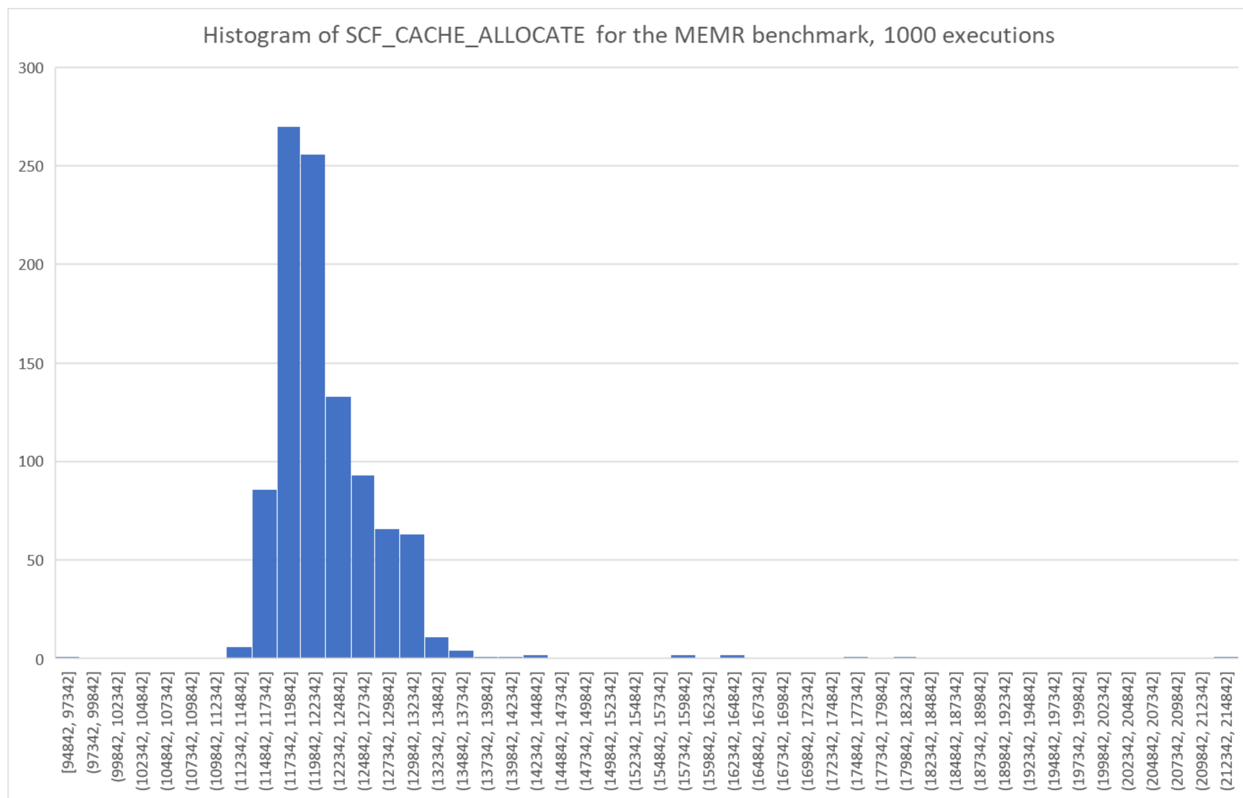


Figure 24 – Accuracy of PMULib for SCF_CACHE_ALLOCATE, shown as a histogram for 1000 executions of the MEMR benchmark

Hence, we cannot use SCF HEMs at operation time due to the overhead the introduce. However, in pre-operation / analysis phases they can be used to provide additional evidence on the timing behaviour of application running on the Orin.

4.5 Technological assessment

In accordance with the task main objectives, we have been performing an analysis of observability support on the Orin platform to understand and identify the available (and relevant) set of HEMs.

We have designed, developed, and validated a lightweight software library for configuring and reading HEMs on the target platform. The library, which has been tested and deployed on top of the SAFEXPLAIN software stack, is meant to capture all monitoring and analysis needs in the project.

With the work done until month 18, we have already captured the main objectives of this task and produced an exhaustive analysis of observability support on the hardware and software stack. The developed PMULib is covering an exhaustive (though preliminary) set of events, which can be further extended to support other relevant metrics in the project.

Delivered tool and positioning in the SAFEXPLAIN stack
This task is also delivering a software tool.

The PMULib is a low-level software library, interfacing with lowest-level software and hardware interfaces in the platform support package and operating system layers. The integration of PMULib on the SAFEXPLAIN software stack answers a two-fold usage scenario:

- *Within the Middleware*: the PMULib can be transparently integrated in the Middleware layer, making it possible to transparently collect timing information at the functional node level. This approach builds on injecting instrumentation code at the node boundaries, hence wrapping each execution of the monitored function.
- *As a user-land API*: the PMULib can also be used explicitly by the end user, thus enabling the user to define the monitoring scope and type of events (HEMs) to be tracked. This approach enables the use of the library to collect metrics other than timing and allows to define on-line monitoring approaches using a standardized, common interface to the HEM layer.

Intra-WP dependencies
Within the scope of WP4, the obtained results, including PMULib, are fundamental inputs to other tasks.

T4.1: The PMULib support has been instrumental to perform well-concocted tests to support the reverse-engineering efforts required to fill the gap in the available technical documentation. The PMPULib has been also exploited to perform an empirical characterization of the software-level interference arising through the software stack, including but not limited to operating system noise.

T4.3: The PMULib is clearly at the basis of any measurement-based timing analysis approach and even more in those approaches requiring some sort of automation for collecting large execution samples, like statistical timing analysis methods. PMULib is also instrumental in confirming properties on the software under analysis or instrumental to the analysis (e.g. synthetic code and interference templates). Additionally, the support offered by PMULib at run-time is the fundamental enabler for the monitoring of shared resource usage, at the basis of interference limitations techniques.

Inter-WP contribution and alignment
The contributions of this task are also relevant in the scope of other work packages. The PMULib is meant to be exploited for supporting WP2, WP3 and WP5 objectives:

WP2: The PMULib is meant to support the validation of Safety Patters to check the intended degree of segregation and performance isolation is achieved at run-time. PMULib is also used to collect various metrics, including but not limited to timing, that can be used for diagnostic task within the FUSA architecture, through the Middleware layer. Finally PMULib is used to offer resource usage monitoring in the scope of SAFEXPLAIN timing multicore interference mitigation strategy.

WP3: The PMULib can be exploited to collect and convey run-time information to support explainability methods and supervision tasks.

WP5: The use cases can use PMULib to support performance tuning and optimization tasks.

The alignment with other work packages, and WP2 in particular, is guaranteed by the continuous interaction between the work packages.

T4.2 Next steps

Task T4.2 is running until m30. PMULib is already offering an extensive coverage of hardware events. The main focus has been so far on most critical aspects form the FUSA perspective, to support V&V and qualification through timing-related analysis and resource usage monitoring. As next steps, we foresee we will devote most effort in extending PMULib support for non-timing-related metrics in the scope of WP3 tasks and to accommodate potential emerging needs in the different use cases. In general, T4.2 will be providing continuous support to other tasks and work packages by refining existing features and developing novel ones, hence covering all observability needs on hardware and low-level software aspects.

5 Timing Prediction Methods and Tools (T4.3)

This task aims at covering timing V&V requirements for AI-based safety-critical systems. SAFEXPLAIN seeks statistical timing prediction methods suitable for DL-based software. The developed methods are expected to exploit statistical methods, and particularly Markov's inequality. The timing verification strategy is extended also to coping with multicore timing interference within a FUSA strategy.

This section starts by providing a description of the overall timing V&V verification strategy and then proceeds by developing a preliminary consideration on the characteristics of the input samples related to the way HEMs are read from the platform. We then move to describe the work done on statistical timing analysis and multicore interference characterization.

5.1 Timing characterization strategy

The work performed in T4.1 and T4.2 led us to conclude that the Orin is a massively parallel architecture with high degree of resource sharing. In fact, the Orin is a high-end MPSoC or edge computing that is more complex than other existing MPSoCs in this domain.

1. The interaction among tasks in shared resources is going to be high. As a result, tasks are going to have a distribution of execution times rather than a single value. This calls for the use of statistical analysis tools to produce WCET estimates.
2. The theoretical (analytical) worst-case scenarios that a task can face in terms of execution time in multicore setups can be simply too pessimistic. To provide for the worst-case timing interference, those scenarios would need to assume all requests conflict in every access to every shared resource, resulting in a WCET estimate that can be too pessimistic to be usable in practice.

In order to capture the first point, we build on several statistical methods as described in Sections 5.2 and 5.3. In order to capture the latter, we need to create a set of contention scenarios, or contention templates, under which the contender tasks are limited in the contention they can introduce on the analysis task (Section 5.4). Templates are used to derive realistic contention scenario that can be used to derive bounds and can be enforced at run-time with a contention monitoring mechanism.

5.2 Inter-Run Variability

One of the first problem we addressed relates to inter-run variability or IRV. IRV relates to the fact that a platform as complex as the Orin keeps an internal hardware state that is impossible to reset after every run. This means that every experiment we carry out starts from different initial conditions, which translates into IRV. This problem has been studied in the literature [9] with focus on processors arguably much simpler than the Orin, where HEMs relevant to multicore analysis increase up to 50% from the minimum value observed. Furthermore, the HEM readings are limited by the number of PMCs available in the platform. Then if the number of HEMs to analyse is greater than the number of PMCs, the observation needs to be done in separate experiments of HEM groups. For instance, if we have 10 HEMs to compare and 5 PMCs, we need to perform 2 separate experiments with HEM groups of 5, from HEMs 1-5 and another from HEMs 6-10. HEMs 1-5 can always be compared among themselves, same for HEMs 6-10, because they were measured at once. But if the IRV is high, runs observed from HEM1 and HEM10, for instance, cannot be directly compared because the initial conditions are unknown for each run. The combination of IRV and a low number of PM Cs causes a limited observability. This can be a problem if the number of

relevant HEMs for the platform is high, like in the Orin as seen in Section 4.2. In the next section we will observe how the IRV applied also to non-functional counters which monitor shared resources like cache accesses or misses.

5.2.1 Empirical evidence

Our first experiments focused on running several times the same application reading the same HEMs in every run. In order to show the IRV we observe the HEM L2D_CACHE_REFILL for different benchmarks. In case of considerably variable values for the event we can conclude the platform (the stack indeed) exhibit Inter-run variability and it must be dealt with.

In Figure 25 below, we show the IRV for L2W, L3W, L4W, MEMW benchmarks.

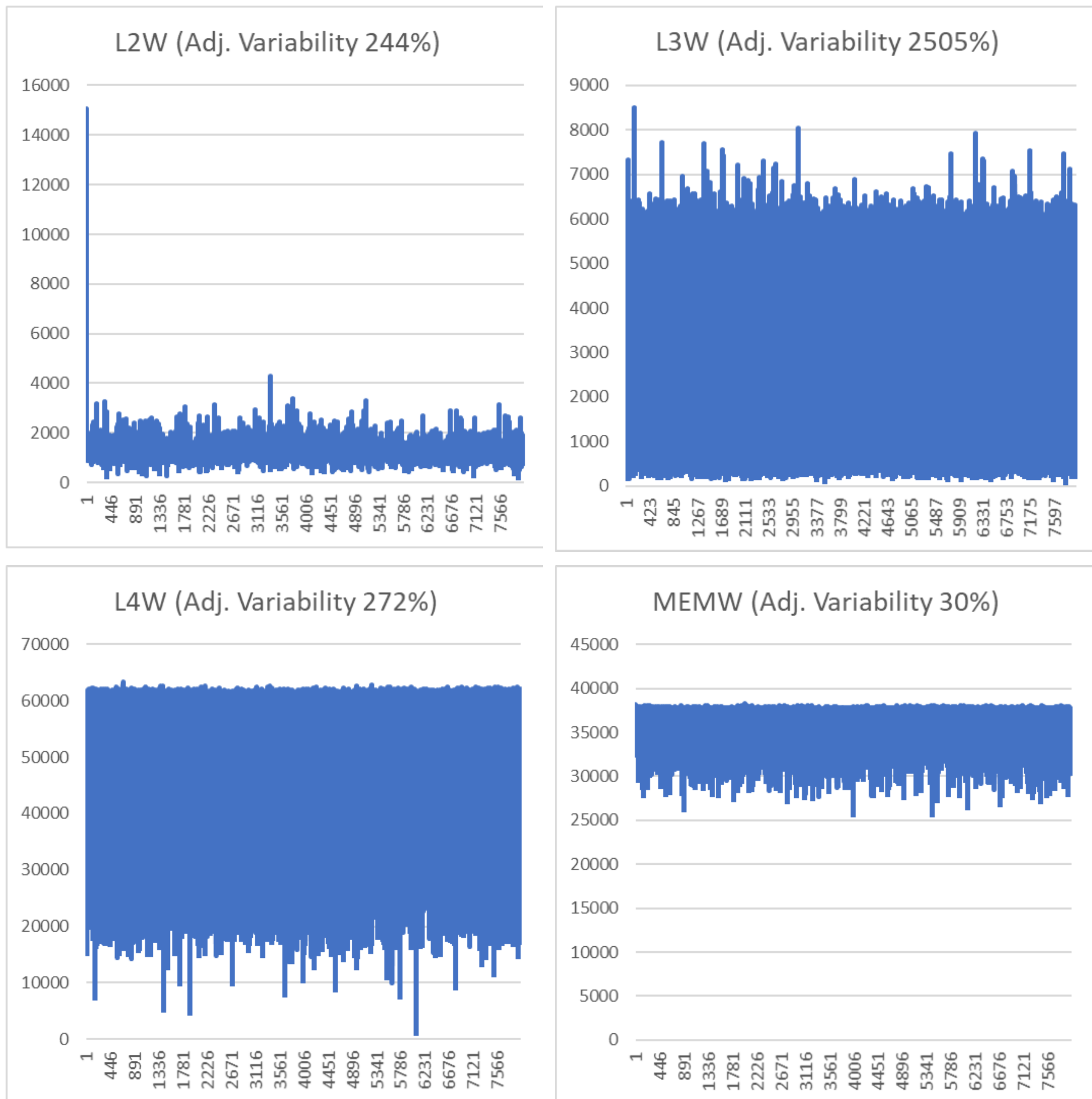


Figure 25 - Inter-run variability for selected benchmarks.

We can observe how, even when not accounting for outliers, all benchmarks suffer from IRV. More specifically, we computed an adjusted variability (ADJ_VAR) to remove the influence of outliers. In the adjusted variability, we compute the ratio between the quantile 1% and 99% of the observed

runs i.e. $ADJ_VAR = \frac{Quantile(99\%)}{Quantile(1\%)}$. In this group of benchmarks, the lowest adjusted variability observed is 30% in MEMW and the highest at 2505% in L3W. Even in benchmarks like L2W where L2D_CACHE_REFILL usually would report low activity, in the Orin the IRV is still high. These results show how, even for a fixed HEM, the IRV is still high across different benchmarks. Now, we will show in Table 1: Relevant HEMs for L2W benchmark another set of observations coming only from benchmark L2W.

Table 1: Relevant HEMs for L2W benchmark.

L2W	ADJ_VAR	MAG	COR
0x17	244%	-2.22	0.74
0x18	373%	-2.26	0.93
0x19	372%	-1.29	0.93
0x23	687%	-3.04	0.28
0x26	423%	-3.1	0.14
0x29	382%	-2.26	0.92
0x2b	384%	-1.89	0.92
0x36	314%	-2.15	0.91
0x52	240%	-2.22	0.67
0x56	371%	-2.27	0.92
0x60	375%	-1.54	0.91
0x61	375%	-1.67	0.91
0xa0	320%	-2.14	0.91
0x4005	211%	-1.2	0.86
0x4009	240%	-2.22	0.68

Here we want to highlight how for a single benchmark there can be many HEMs which show high IRV. In L2W we have adjusted variability between 211% and 687%. The second column represents the difference in order magnitude of the HEM w.r.t. the ET. As we can observe, these HEMs are within 2 orders of magnitude below the ET. This is done in order to avoid HEMs that can be in low order of magnitude w.r.t. the ET which would produce trivial high variability.

Finally, we show the correlation between these HEMs and the ET. The correlation is shown to give evidence that those HEMs contribute to the ET in a significant way. HEMs which have correlation 0 do not provide information on the timing. Similarly, HEMs with correlation 1 reproduce the behaviour of ET, and therefore provide no information. We filtered the HEMs with correlation between 0.05 and 0.95. With this table we want to highlight a group of HEMs which i) have high variability, ii) have magnitude comparable to the ET, and iii) are correlated with the ET in a significant manner. This implies that, in order to characterise the ET for the L2W, ideally, we need to observe all HEMs on the table at once. But because of the high IRV this is not directly possible, and we need a solution that let us merge observation from different HEM groups.

5.2.2 MUCH

As mentioned, the IRV plus the limited number of PMCs cause a limited observability. Due to the IRV we can treat the HEMs as marginal distributions, which are governed by a joint probability distribution which is not possible to observe, i.e. an observation with all HEMs at once. Fortunately, there exists techniques to estimate the joint probability distribution from the observed marginal

distributions of HEMs [10]. The approach we are using is based on modelling the dependency structure of the HEMs with copulas. Specifically, we use the technique MUCH which provides a single output from partially observed data maintaining the dependencies as if they were measured at once. MUCH is based on the multivariate Gaussian distribution (MVG) to model the relationships between HEMs and integrate them while maintaining the pairwise correlations among each pair of HEMs. The multivariate Gaussian distribution is defined as follows,

$$p(x; \hat{\mu}, \hat{\Sigma}) = \frac{1}{(2\pi^{p/2})|\hat{\Sigma}|^{1/2}} \exp\Big(-\frac{1}{2}(x-\hat{\mu})^T \hat{\Sigma}^{-1}(x-\hat{\mu})\Big)$$

Where $\hat{\mu}$ are the estimated mean of each HEM, and $\hat{\Sigma}$ is the estimated pairwise covariance matrix of all measured HEMs, and p is the number of HEMs. The dependency structure between the HEMs is stored in the covariance matrix of the MVG. MUCH requires each pair of HEMs to be measured together in order to estimate their pairwise correlation. Then, each element of the correlation matrix R is the pairwise correlation of a pair of HEMs. Then, the covariance matrix can be calculated exactly using the formula:

$$\Sigma = \mathrm{diag}(S) \times R \times \mathrm{diag}(S)$$

where $diag(S)$ is a diagonal matrix of the standard deviations of the HEMs. It is worth noting that, even when measuring all pairwise correlations, preserving them at once when merging is a very complex optimization problem. Measuring every pair of HEMs can become expensive very fast as the number of relevant HEMs increases. In combinatorics, it is an optimization problem to find the smallest number of readings required to observe all pairs of HEMs with a given number of PMCs. For instance, that the optimal experimental design for measuring all pairs from 15 different HEMs with 6 PMCs is 10 different readings, with 100 runs for each group of HEMs to account for variability, with 1000 runs we could produce a merge with MUCH in this case. For other cases, in [11] one can search for the optimal covering design with multiple combinations of HEMs and PMCs

In the following Figure, we show the procedure of MUCH. After generating the optimal number of readings (1) we compute the correlation matrix (2) and feed it to the MVG model (3). The MVG allows us to generate synthetic data (4) that mimics the dependency structure of the experimental data. With (4) we have a model of the joint distribution of HEMs, but the values are synthetic. The final merge should be with the experimental data. In order to do that, we make use of order statistics to transfer the dependency of the synthetic data to the experimental data. Given a random variable X are the order statistic $X_{(k)}$ is the k-th lowest value of X. We translate the data generated from the MVG to its order statistics (5). What this gives us is a map to arrange the experimental data which preserves the experimental pairwise correlations.

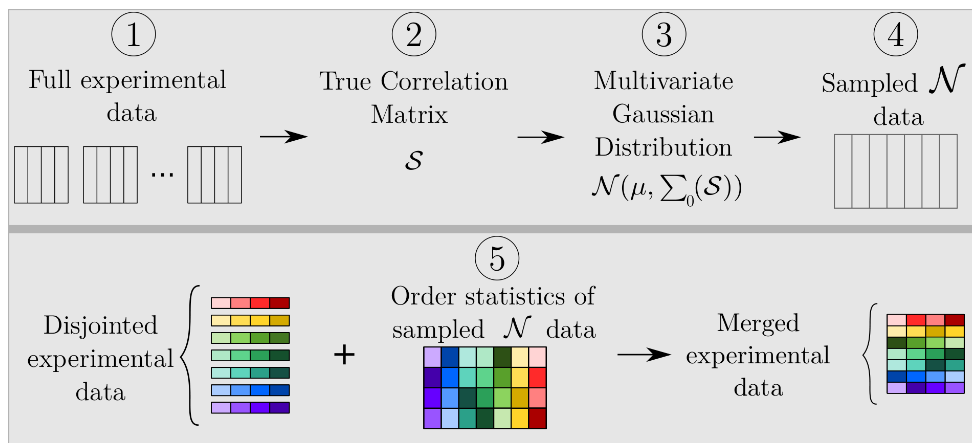


Figure 26: Procedure for the MUCH algorithm.

For instance, let us say that the synthetic MVG data in terms of order statistics looks like this X_{MVG} = {$HEM^1_{(8)}$, $HEM^2_{(3)}$, $HEM^3_{(4)}$, $HEM^4_{(7)}$, $HEM^5_{(1)}$, $HEM^6_{(5)}$}, this means that when HEM 1 takes the 8th lowest value, then HEM 2 takes its 4th lowest value, and so on. We use the order statistics from the MVG synthetic data in this way to arrange the experimental data. After arranging, we have a single output of merged experimental data that maintains the pairwise correlations. While MUCH requires additional runs, and for this reason we use it in our methodology Figure 26: Procedure for the MUCH algorithm when we cannot prevent IRV, the accuracy on the preservation of the pairwise correlation increases with the number of runs. Overall, MUCH deals with IRV as if all HEMs could be read in the same run, effectively removing IRV given that now HEMs that were measured in different runs can be compared and analyzed.

We show in Table 2 the output of MUCH from the merging of experimental runs coming from the L4R benchmark. The HEMs measured were selected from the list of relevant HEMs in Section 4.2, and specifically those which have high IRV.

Table 2: merged experiments from MUCH.

0x11	0x45	0x15	0x2a	0x2b	0xa0	0x18	0x17	0x400b	0x19
1365418	41692	466	89348	258608	143488	140366	52524	89254	1131969
1367885	42510	475	99571	271593	151482	143760	53278	100836	1192341
1321751	38681	475	91269	272021	150383	142908	49725	91169	1176444
1375869	42813	476	93540	259103	145442	140724	53631	93534	1144997
1346175	42589	463	97651	273419	152187	143978	53371	98047	1199099
1367048	43350	473	94906	260957	142982	139957	54036	94866	1128890
1314183	37701	619	88345	266902	144565	140022	48833	88276	1139224
1336464	41682	476	95189	270054	146733	141496	52514	95199	1154454
1361514	41295	468	93081	267294	148630	142777	52206	93086	1164818

As a way to measure the accuracy of the merge we show the pairwise correlation difference of each pair of HEMs before and after merging with MUCH. Note that the challenge in MUCH is preserving all pairwise correlations simultaneously: in Table 3 we can see how the correlation difference is low for all pairs of HEMs.

Table 3: Correlation difference before and after merging.

	0x11	0x45	0x15	0x2a	0x2b	0xa0	0x17	0x18	0x400b	0x19
0x11	0	-0.02	0.01	-0.02	0	-0.02	-0.01	-0.01	-0.05	-0.15
0x45	-0.02	0	0.03	-0.02	0.01	0	0.03	-0.02	0.01	-0.05
0x15	0.01	0.03	0	0	0	-0.02	0.04	0.02	0.06	0.02
0x2a	-0.02	-0.02	0	0	0.02	-0.01	0.07	-0.01	0	-0.06
0x2b	0	0.01	0	0.02	0	0	0.01	0	0.01	0.03
0xa0	-0.02	0	-0.02	-0.01	0	0	-0.03	-0.01	-0.02	0
0x17	-0.01	0.03	0.04	0.07	0.01	-0.03	0	-0.03	0.09	-0.08
0x18	-0.01	-0.02	0.02	-0.01	0	-0.01	-0.03	0	-0.01	-0.06
0x400b	-0.05	0.01	0.06	0	0.01	-0.02	0.09	-0.01	0	-0.06
0x19	-0.15	-0.05	0.02	-0.06	0.03	0	-0.08	-0.06	-0.06	0

5.3 Statistical Analysis based on the Markov Inequality

As discussed above, the inherent complexity of heterogenous high-performance MPSoCs and the AI-based applications running on top of them is jeopardizing the application of traditional methods for timing analysis, which are well suited for comparatively simpler systems and applications [12]. Probabilistic timing analysis techniques have been increasingly considered as alternative approaches to comply with timing verification requirements in an effective and efficient way in complex scenarios. Probabilistic methods entail a paradigm shift with respect to deterministic approaches and their use for supporting certification of critical systems is still under debate.

The approach we intend to follow in SAFEXPLAIN is oriented towards certification and is therefore considering a practical approach making the use of statistical methods more appealing from a traditional perspective. SAFEXPLAIN considers a more holistic approach where (probabilistic) timing bounds are used in combination of an augmented assured safety net concept that is not anymore limited to capturing and reacting to timing failures but provides adequate coverage also for timing failures. Diagnostic and monitoring mechanisms are therefore responsible for capturing residual risk of timing failure. This approach allows to consider timing overruns within the concept of residual random fault in ISO-26262 [13].

The score of probabilistic methods in the literature is mainly focused on Extreme Value Theory (EVT). EVT has been shown to provide proper conservative timing upperbounds in general. In timing analysis, the peak over threshold (PoT) methodology has been studied extensively. The basis of PoT is that the tail of the distribution, defined from a threshold where the extreme values belong, pertains to the generalised pareto distribution (gpd). In general, PoT works adequately when the threshold for the tail can be adequately estimated. However, in more complex scenarios EVT can struggle to assess the nature of the tail.

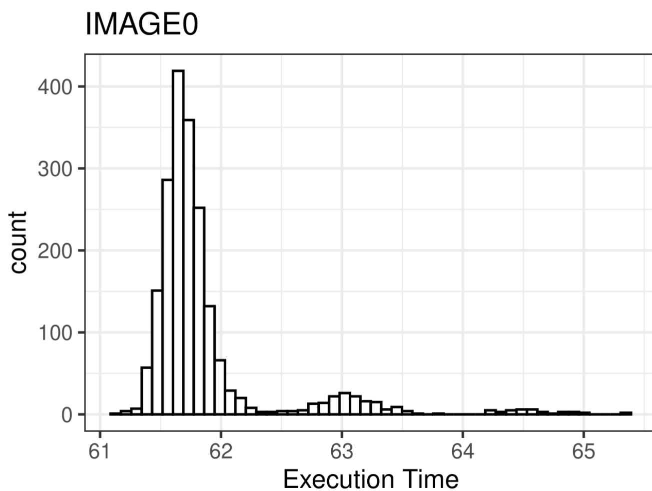


Figure 27: Histogram for IMAGE0 of the toy model.

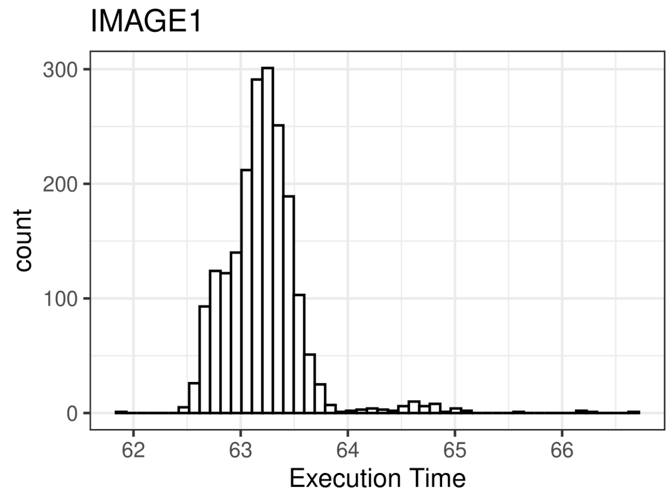


Figure 28: Histogram for IMAGE1 of the toy model.

In Figure 27 and Figure 28, we show the histograms of the execution time distribution of processing two images with the Toy Model developed by WP5 [14]. The rest of the images show a similar pattern. These execution time profiles show a mixture distribution, which is complex to analyse from an EVT point of view due to the wide shape of the distribution and the lack of information on the tails.

Recently, an alternative method to upperbound extreme value based on Markov's Inequality was proposed [15]. Let $X > 0$ a positive random variable, then Markov's Inequality is defined as:

$$P(X \geq b) \leq \frac{E(X)}{b}$$

Which means that the cumulative probability that X takes a value bigger than b is bounded by the expected value of X divided by b. Markov's Inequality works properly for lower values of b, but for more extreme values it gets overly pessimistic. Markov's Inequality can be modified to provide bounds which are much close to the real cumulative probability with an increasing non-negative function like the power function. Here we define Markov's Inequality to the power-of-k function as:

$$P(X \geq b) \leq \frac{E(X^k)}{b^k}$$

where $E(X^k)$ are the moments of the distribution of X. Because in general these moments are not known, we use the sample moment estimator:

$$\hat{E}(X^k) = \frac{1}{N} \sum_{i}^{N} X_i^k$$

This estimator is unbiased, but it is inconsistent for higher values of k. As we increase the value to be upperbounded, b, the value of k needs to increase also to get tight upperbounds. In order to obtain tight upperbounds, in [15] there is an implementation of Markov's Inequality to the power-of-k with an algorithm called Restricted k (RESTK) that limits the value of k used in accordance to the target probability to upperbound. To obtain a limit on k we exploit a linear relationship between the logarithm of the probability and the highest value of k for a tight upperbound.

Here we show some preliminary results for the timing analysis of the Toy Model [14] for the timing profile of two images. In this analysis outliers were removed to provide a more consistent projection of the extreme values. We provide two projections of the timing for extreme quantiles with two models. One based on EVT, by fitting an exponential distribution to the tail of the empirical values; and the other using the RESTK algorithm based on Markov's Inequality. The EVT methodology to select the threshold for the tail is based on minimizing the estimated quantiles mean absolute error, of the empirical quantiles and the estimated model [16]. In Figure 29Figure 29: extreme value estimates with the Exponential and RESTK for IMAGE0 and Figure 30 we show the estimates of the extreme values for IMAGE0 and IMAGE1 respectively. The exponential distribution is regarded to be a proper upperbound for the extreme values. However as we can see, while in Figure 29 it seems to estimate quantiles which are quite far from the empirical distribution; in Figure 30 it seems to be too conservative in its extreme value estimations. In contrast, RESTK provides a more consistent projection of the extreme values. Because we do not have the reference values for extreme quantiles to assess their accuracy, we can compare these models by their consistency in the extreme value projections. In Figure 31 we provide some results computed with the ratio between the estimated extreme value at probability $p = 10^{-8}$ and the maximum value in the empirical data. There we can see how RESTK provides a much more consistent extreme value estimation at around 8% increase w.r.t. the maximum value observed. The Exponential model instead is much more inconsistent across all timing profiles with estimates up to 20% increase w.r.t. the maximum value observed.

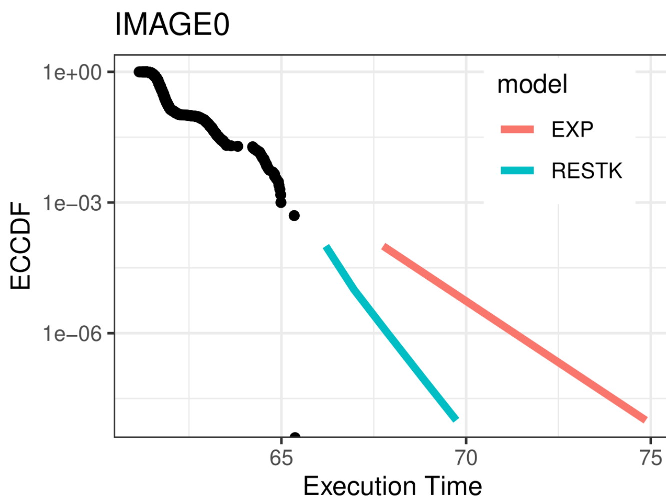


Figure 29: extreme value estimates with the Exponential and RESTK for IMAGE0.

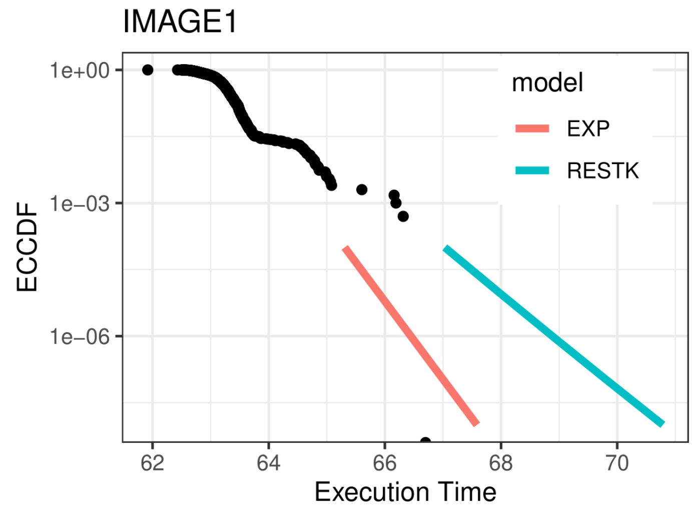


Figure 30: extreme value estimates with the Exponential and RESTK for IMAGE1.

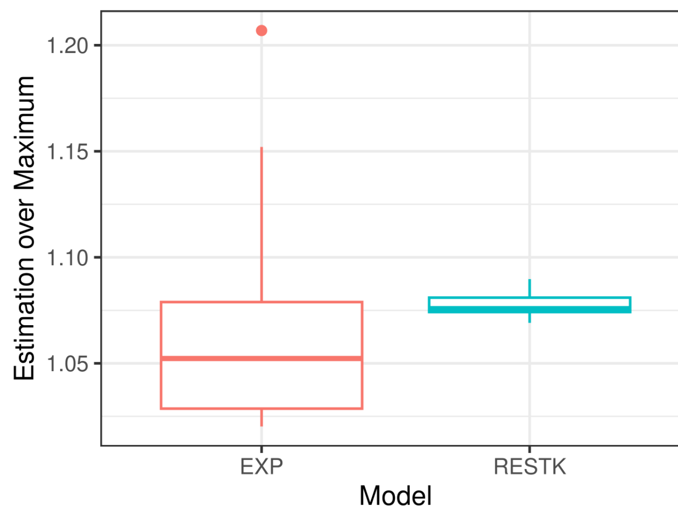


Figure 31: Ratio of the estimated value and the maximum observed value in the sample for all 21 images.

5.4 Interference monitoring mechanism and Templates

While MPSoCs offer the necessary performance for complex AI-based applications, they also expose to multicore timing interference or contention impact. Timing interference stems from massive hardware resource sharing and the delays potentially incurred when requests hit the same resource simultaneously. The incurred variability cannot be disregarded [17] and can have disruptive and disproportionate effects on the timing bounds computed in isolation. The main problem with contention modelling approaches is that they are meant to be conservatively bounding the impact of interference, which translates into assuming all concurrent requests to a given resources are serialized. This leads to overly conservative bounds, accounting for scenarios that, despite being theoretically possible, will never happen in practice as overlapping in time of requests to the same device cannot always happen.

More practical approaches, in view of avoiding excessive pessimism, build on observation of contention impact under pre-defined multicore scenarios where the application under analysis is deployed against synthetic applications or attackers that put very high pressure on specific shared resources, intended as interference channels. We note, however, that there is no silver bullet solution and also these methods based on empirical observations can fall short: on one hand, they can only support arguments based on the observed scenarios so that if the attackers are not

aggressive enough, we may not consider critical scenarios at run-time. On the other hand, too aggressive attackers may be overly effective and hitting the interference channels more than actually possible at run-time, given the actual applications that will be executing in the systems at operation.

5.4.1 Templates

The approach adopted in SAFEXPLAIN exploits the idea of *templates* as a set of configurable attackers that are built and configured in a way that resembles (and only slightly over-represent) the non-functional behavior of the contender applications in the system. The concept of *templates* builds on the concept of *signatures* introduced in [18]. The approach then consists in defining different contention thresholds by adjusting the 'aggressiveness' of templates and perform an empirical assessment of the interference suffered by the target application, under different but realistic contention scenario mimicking the behaviour of the actual co-runner applications, thus preventing excessive pessimism. We also address the possible residual lack of representativeness of contention scenarios at analysis time by deploying a safety mechanism, intercepting when contention scenario at run-time exceeds that assumed and synthetically enforced at analysis time.

We have developed a set of prototype templates that offer a coarse-grained configurability, which we aim at improving in the next steps while adapting templates to automatically capture use case scenarios. We report below results on the experiments conducted to assess the behavior of templates.

In Figure 32 we can see the execution times of different templates tailored to put different amounts of pressure on the shared L3. The execution time is presented relative to the single core execution time of L3R. The contenders are presented as different series, where L3W is the benchmark putting full pressure on the L3 cache, while L3W-6% to L3W-1% put a decreasing amount of pressure on the L3 cache, being L3W-1% the one putting less pressure of all the templates. As expected, it can be observed that the L3W benchmark causes the most slowdown due to it being the most aggressive contender, while L3W-1% has an execution time almost identical to the baseline case where no pressure is observed.

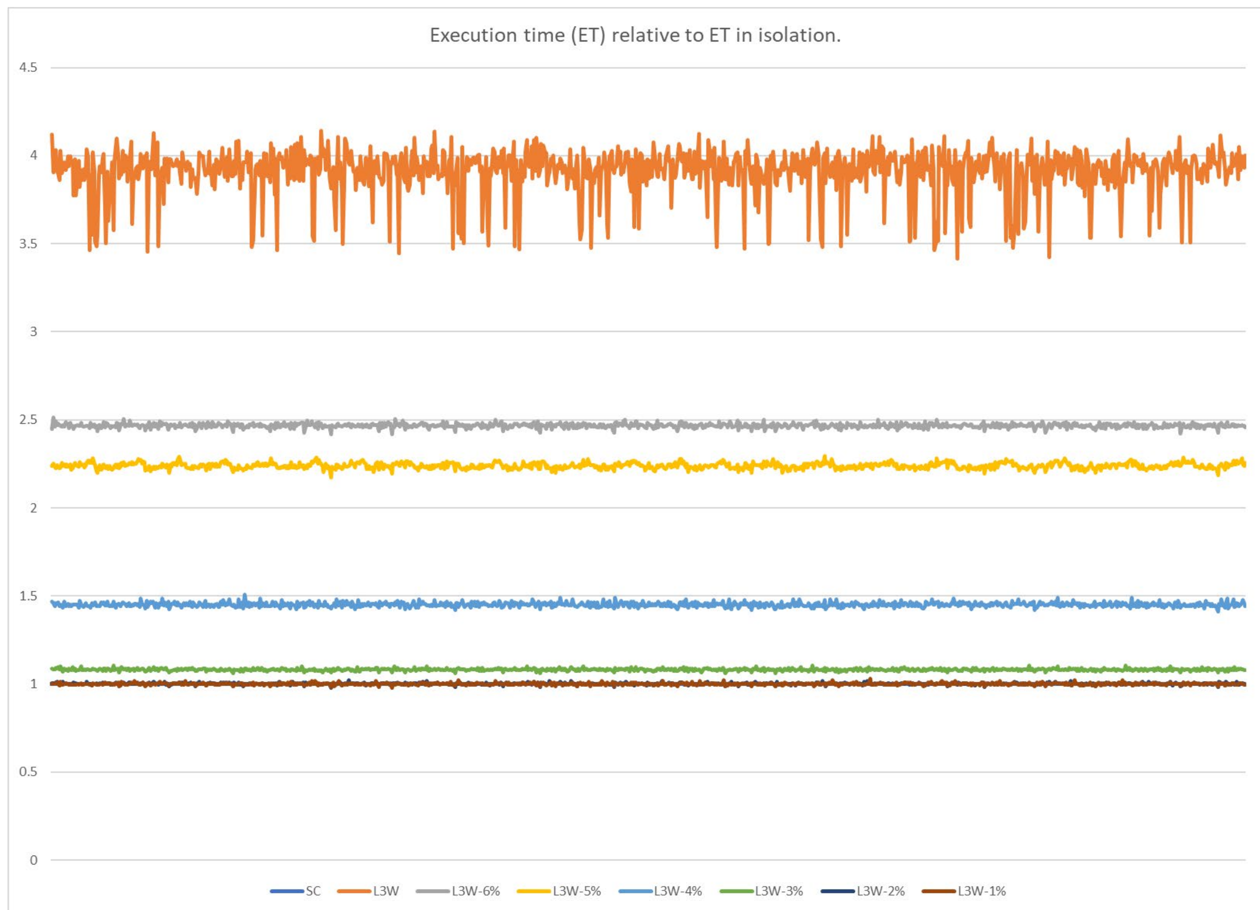


Figure 32 – Execution time of different templates of L3W compared to the baseline in single core. All contenders running in the same cluster.

We have been also working on an early prototype for the contention monitoring mechanism. Such mechanism is mainly consisting in two modules: one responsible for detecting whenever contention thresholds are exceeded, and another one responsible for reacting to such events. In this first phase we focused on the capability of intercepting contention thresholds overruns by focusing on the identification of what hardware events to exploit and how to monitor them. We converged on an initial set of HEMs that are accessible via PMULib and can be arguably related to activities on shred resources and, hence, contention. We will further refine this selection by applying correlation analysis techniques to an extensive set of HEM tests. The reaction module is prototyped as a warning mechanism that notifies contention anomalies at system level, exploiting Middleware system level diagnostic support (see Section 6.2).

5.5 Technological assessment

In accordance with the task main objectives, we have been working on the definition of an effective strategy for the timing analysis of DL-based components, covering both characterization of timing behavior and impact of multicore timing interference.

In this first reporting period we have been focusing on the two aspects by developing an holistic strategy, combining statistical timing analysis and interference monitoring together with interference mitigation solutions identified after an in-depth analysis of the hardware and software stack.

On the timing analysis aspect, we focused on the identification of the most appropriate statistical method to capture the inherent variability (hence complexity) of AI modules. We have taken

advantage of early release of the SAFEXPLAIN stack (including the Middleware) and an illustrative AI module (Toy Model, see [14]) developed in WP5 to evaluate different methods in a representative setup. On the timing interference analysis part, we have been identified an effective strategy (aligned with WP2 FUSA approach) to analyse timing interference in a tight way and provide assurance at run-time by leveraging contention monitors. Prototype tools have been developed and will be refined/complemented for the early technological integration.

Delivered tool and positioning in the SAFEXPLAIN stack

Prototype tools, while available in the common project code repository, they are not meant to be exploited yet by the end users. Prototypes will be mature in time for early tool integration, as planned.

- *Statistical timing analysis tool*: it consists in a set of R scripts that can be used to analyse raw sample data corresponding to execution time samples. The tool is functionally ready and will be shared with the project partners together with exhaustive documentation and examples.
- *Templates*: consisting in a set of synthetic code snippets that can be configured to mimic a real application in terms of timing interference they can produce. Templates have been already deployed for preliminary analyses and will be undergo few improvements on configurability before being shared with project partners in time for tool integration.
- *Contention monitoring*: early prototype tool to monitor HEMs correlated to resource contention through PMULib. Will complement monitoring with a reaction strategy in case predefined resource usage thresholds are exceeded.

Intra-WP dependencies

Within the scope of WP4, the activities in T4.3 are building on the results achieved in other WP4 tasks:

T4.1: The multicore timing interference strategy devise in T4.3 builds on the conclusions developed from the in-depth hardware analysis on the Orin platform. The identification of the interference channels is a prerequisite for the identification of the HEMs that are correlated to resource contention and must be therefore tracked to monitor contention at run-time.

T4.2: For this task, we built on PMULib as a fundamental enabler for collecting the execution time samples provided in input to the timing analysis tools developed in this task. The same library is also enabling the monitoring of resource usage to intercept activity from contender tasks exceeding the expected resource usage thresholds at operation.

T4.4 The resource usage monitoring mechanism we are developing in this task is building on the functionalities offered by the SAFEXPLAIN middleware to perform the monitoring as independent thread and to exploit system level diagnostic support.

Inter-WP contribution and alignment

The contributions of this task are also relevant in the scope of other work packages. Within the extended scope of the project, hence outside of WP4, T4.3 software tools are meant to be exploited for supporting other work packages objectives, but mainly WP2 ones:

WP2: Timing analysis tools are meant to support the (timing) V&V strategy, covering both timing characterization and multicore timing interference mitigation strategy, including run-time monitoring of resource usage. The timing analysis and interference mitigation strategy and tools are fundamental inputs for the deployment of the timing V&V strategy on the use cases, in alignment with WP2 FUSA strategy and Safety Patterns [1]

WP5: The use cases can use PMULib to support performance tuning and optimization tasks.

The alignment with other work packages, and WP2 in particular, is guaranteed by the continuous interaction between the work packages. The main aspects on which we consolidated the technical alignment with WP2 can be summarized as follows:

1. **Platform configuration**. Platform configuration is a critical concept form the FUSA perspective as it affects performance, functional correctness, and eventually also the degree of freedom from interference that can be obtained on the system. Configuration must be fixed, analysed, and protected from unintended changes at operation [1]. The most prominent configuration affecting performance is the power mode. Our approach advocates for fixing it and keeping it unchanged during system operation. When it comes to interference mitigation, the timing characterization strategy defined in this section is largely dependent on the configuration of the system in terms of, for example, execution mode of COU clusters (e.g. lockstep), partitioning of the cache hierarchy, and application to core mapping. The exact configuration is determined by the Safety Pattern, which can also be seen a selection among the possible configuration options surveyed in this document.
2. **Usage of resources**. Safety Patterns also need to accommodate specific requirement from the system and the set of provided functionalities. We considered it relevant to intercept potential limitation and constraints from the case studies as early as possible in the definition of the Patterns. We submitted a questionnaire to the case study providers to understand the resources exploited in their case studies. We have determined that they can use from 1 to 3 CPU clusters and the GPU. At the moment, they are not using other accelerators like PVA or DLA. This has left ample freedom for the definition of the Safety Patterns.
3. **Interference channels identification**. Among all hardware features in the Orin, we have identified the cache levels and the path to memory as the main source of interference aka interference channels. The first and second level (DL1 and UL2) are private, and we have shown that tasks do not suffer increase in execution time or L2 miss rate, when they run with other contenders. For the UL3 we propose to enable hardware cache partitioning. For the L4 and memory not simple ways of space isolation are possible, so we advocate for contention templates and probabilistic analysis to cover the residual interference stemming from them (see Section 5.1).
4. **WCET and timing analysis strategy**. As discussed in Section 5.1, we adopt an holistic approach to timing analysis that complements statistical pWCET analysis with timing interference control mechanism based on shared resource monitoring. We add as part of the configuration the contention template that limits the number of accesses that contender CPUs can perform to the L4 and memory (as lower memory layers can be partitioned). Under the specific configuration scenarios, we will perform stress testing to derive execution time measurements of the task under analysis in stress conditions induced by templates. The values observed will be used to feed statistical timing analysis and obtain reference thresholds to be used to implement run-time control mechanism, reacting to over-use of shared resources by tasks.

T4.3 Next steps

Task T4.3 is running until m30. Timing analysis tools are already consolidated and may require only minor refinements and tailoring. After assessing different statistical methods, in the next period we aim at improving the degree of automation we can offer from sample collection to obtaining timing results. To better support timing-related analysis and resource usage monitoring, we aim at developing a stronger integration with SAFEXPLAIN middleware, with a view to making timing

analysis as effortless (and transparent) as possible to the end users. In general, T4.2 will be providing continuous support to other tasks and work packages by refining and further developing tool in support to the timing verification strategy. We foresee no obstacles in proceeding towards the tool integration as expected.

6 Platform- and System-level V&V support (T4.4)

This task focuses on platform and system-level support for integration and validation of DL libraries and explainable AI approaches on top of the target platform. The main objective is to support the integration of the different FUSA and AI solutions on the platform and facilitate (also by means of enhanced automation) the V&V campaign by providing a consolidated, standardized testing environment supporting the collection of relevant metrics for functional and non-functional verification (e.g., AI metrics).

The approach taken in SAFEXPLAIN is building on the definition of an abstraction layer to facilitate integration of SAFEXPLAIN solutions (both FUSA and AI) on the platform and across use cases. The abstraction layer, which we call SAFEXPLAIN Middleware is meant to capture the requirements in terms of traditional V&V support and off-line/on-line support to (explainable) AI modelling, training, and prediction methods.

In this section, we start with an introduction to the Middleware concept and how it fits in the SAFEXPLAIN middleware. Next, we provide a description of the main features it provides both for capturing FUSA/AI needs and for supporting V&V activities.

6.1 SAFEXPLAIN Middleware concept

After considering the objectives of this tasks and the need for supporting diverse configurations we concluded that we needed an abstraction layer to separate the common platform- and system-level concerns from the particular AI system and applications.

The two main aspects we wanted to capture with the proposed abstraction are:

- *Provide a common setup for porting use cases*: this aspect covers both the need for simplifying the porting by taking care of low-level hardware configurations and setups, and the benefits (in terms of WP4 objectives) of having a common environment across all use cases.
- *Compliance with FUSA architecture*: we aimed at enforcing compliance with FUSA architecture from WP2 by construction. The user is not required to re-implement or heavily adapt their system to cover FUSA aspects and modules (e.g., diagnostic levels, etc.). The middleware is meant to provide a set of consolidated components that can be deployed and used as containers for the particular applications. Not all components are mandatory, but a shortlist is necessary to comply with the baseline FUSA requirements and to exploit at full the benefits of SAFEXPLAIN explainable and dependable AI solutions.
- *V&V support:* as SAFEXPLAIN solutions need to be tested and assessed in the scope of a standard V&V campaign aiming at system qualification and certification. In this respect, the goal of the Middleware is to provide support for standard V&V practice by means of: (i) support automated, repeatable testing; (ii) collection of functional results; (iii) collection of timing metrics and execution time samples; (iv) collection of AI metrics (accuracy, deviation, errors, etc.); and (v) support life-cycle management features.

The Middleware is therefore meant to support the execution of the use cases in a consistent environment where FUSA and explainable AI concepts are implemented. The middleware is also supporting the integration of DL and Explainability libraries on top of the hardware and software stack and consistently with the FUSA architecture and the concrete Safety Patterns.

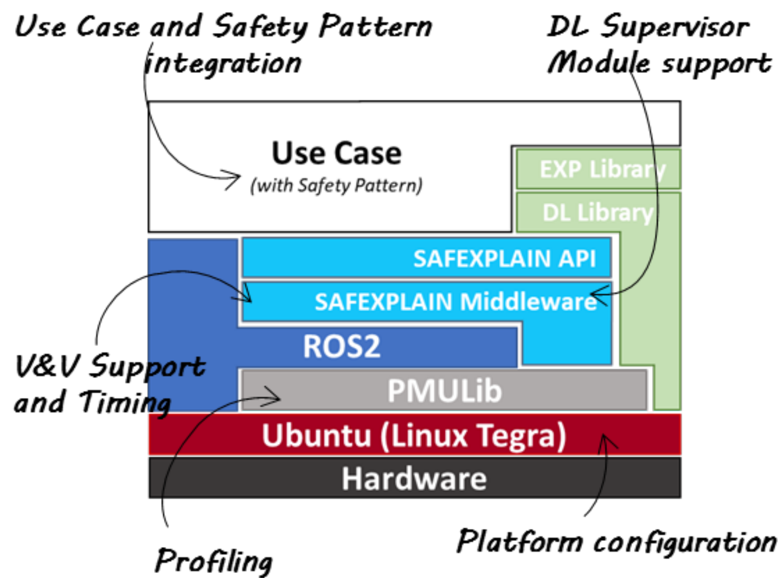


Figure 33 - SAFEXPLAIN Middleware overview.

Figure 33 illustrates the positioning of the Middleware layer (and its provided API) within the SAFEXPLAIN stack. We observe how the Middleware support use cases and AI libraries and integrates with the PMULib, which can be accessed through the Middleware or independently from it.

Perhaps the most interesting aspect in the diagram is the relation between the Middleware and ROS2 [5]. ROS2 stands for version 2 of the Robotic Operating System, which is a well-known and widely used middleware layer supporting a distributed semantics among nodes that communicate each other through a publisher-subscriber model. The Middleware is indeed implementing a wrapper layer around ROS2 to provide a set of standard services and features. Therefore, we exploit the well-consolidated framework to incorporate SAFEXPLAIN features and solutions in a more or less transparent way to the user application. In this way, we exploit the familiarity of end users with ROS2 semantics (most of AI based application are already modelled on ROS2 semantics) while at the same time being able to deploy ad hoc solutions. It is also worth noting that the user application is also allowed to access some ROS 2 functionalities without using the wrappers provided by the Middleware. What is relevant, instead, is that the user application is using those wrappers that implement the FUSA architecture and explainability logic.

In the next sections we will focus on the main features and support provided by the Middleware layer in terms of FUSA and AI elements, and V&V support.

6.2 SAFEXPLAIN Middleware support

6.2.1 Support to FUSA Architecture

The functional-safety architectural pattern presented on Section 2 of D2.2 [1] represents a reference architecture to build a safety critical system deploying AI-based modules, integrating different safety mechanisms to ensure the fulfilling of the system safety objectives. In particular, the proposed safety architecture can be tailored for each use-case, including (or not) software blocks in charge of:

- Implementing diverse redundancy within the DL model.
- Implementing a non-AI fallback element.
- Supervising the DL output.

- Stablishing a safety envelope (i.e. for safe operation).
- ...

Nonetheless, it's expected that all use-cases make use of diagnostic and monitoring mechanisms common also to traditional functional safety applications. The SAFEXPLAIN Middleware provides several libraries and services to standardize access to the core functionalities of platform applications, including to those traditional diagnostic and monitoring mechanisms. The Middleware is designed to deploy a set of standard and custom components to allow a straightforward and modular mapping between FUSA architectural pattern (and concrete Safety Pattern) and concrete use case deployment.

In the following discussion, we'll discuss the packages developed in the context of the SAFEXPLAIN project. The SAFEXPLAIN libraries and executables hereafter discussed are named as smw_*.

In the SAFEXPLAIN Middleware, the supervision of all platform applications is provided by a platform health manager (package smw_health_manager), as illustrated in Figure 34.

When it detects a violation of the configured temporal, logic or health constraints, it can trigger an appropriate error handling. Depending on the detection violation, the health manager shall be configured by the application developer (through a Safe State Setup file) to:

- Trigger an application stop or reset
- Trigger a platform reset, through an external device (e.g. external watchdog)
- Request a pre-defined action from the safety control

Application developers are advised to develop their safety-related applications inheriting from the smw_base_application::BaseApplication (package smw_base_application), so that they can be configured as a platform supervised entity. Each supervised entity can communicate the platform health manager that critical checkpoints have been reached, thus allowing for temporal (alive and deadline) monitoring.

Currently, the Platform Health Manager already supports automatic discovering of all safety-related base applications, and it provides alive and deadline monitoring upon user configuration. Logical supervision might also be performed through checkpoint reporting, but it's not supported yet. Indeed, no use-case has manifested the need for it, so it has been handled as a secondary feature that might be implemented in the future.

Health status monitoring instead can be used to collect diagnostic and monitoring information from other elements of the system, such as the Safety Control and the Supervision Components of the AI-based subsystem. For instance, the L0-L1 diagnostics can notify the platform health manager whether redundant instances are providing too mismatching results or that a particular AI instance is using excessively a system resource. In such a case, through a Safe State Setup file, rules can be defined by the application developers upon the notification of a certain health status, leading to the execution of a different action list. Even though the platform has already been designed for supporting this Safe State Setup, the implementation is still a working-in-progress activity and it's one of the next steps in implementation.

In the SAFEXPLAIN platform, the inter-process communication (IPC), including checkpoint and health status reporting, is addressed at its core by the ROS2 DDS. However, instead of employing the usual publish/subscribe mechanisms of ROS2, application developers are advised to use the package smw_comm.

Currently, the smw_comm is a light wrapper on top of the ROS2 IPC utilities. In addition to the standardized features, it allows data consumers to have access to input data buffering. This feature is absent in ROS2, but it's especially useful for the use-cases, since the input data streaming possibly happen at a faster rate than the AI processing constituents. A synchronization mechanism between different data sources – allowing them to be processed together – is foreseen to be implemented as part of the next steps. Finally, the smw_comm might support access control policies that mediate the request to data and services maintained by the platform.

In SAFEXPLAIN Middleware repository (https://gitlab.bsc.es/safexplain/safexplain_middleware), the full package list developed for the SAFEXPLAIN project can be found. Besides the ones previously discussed, this list includes a wrapper for the ROS2 logging libraries (package smw_logging), helper classes for testing (package smw_testing) and a collection of executables and libraries used internally in the platform to support the aforementioned functionalities provided (e.g. packages smw_core, smw_lifecycle_manager, smw_state_manager, etc.).

Finally, application developers can count on the examples provided in the package smw_examples to support their development. It includes a fully functional toy model, that will be keep updated with the new additions to the platform.

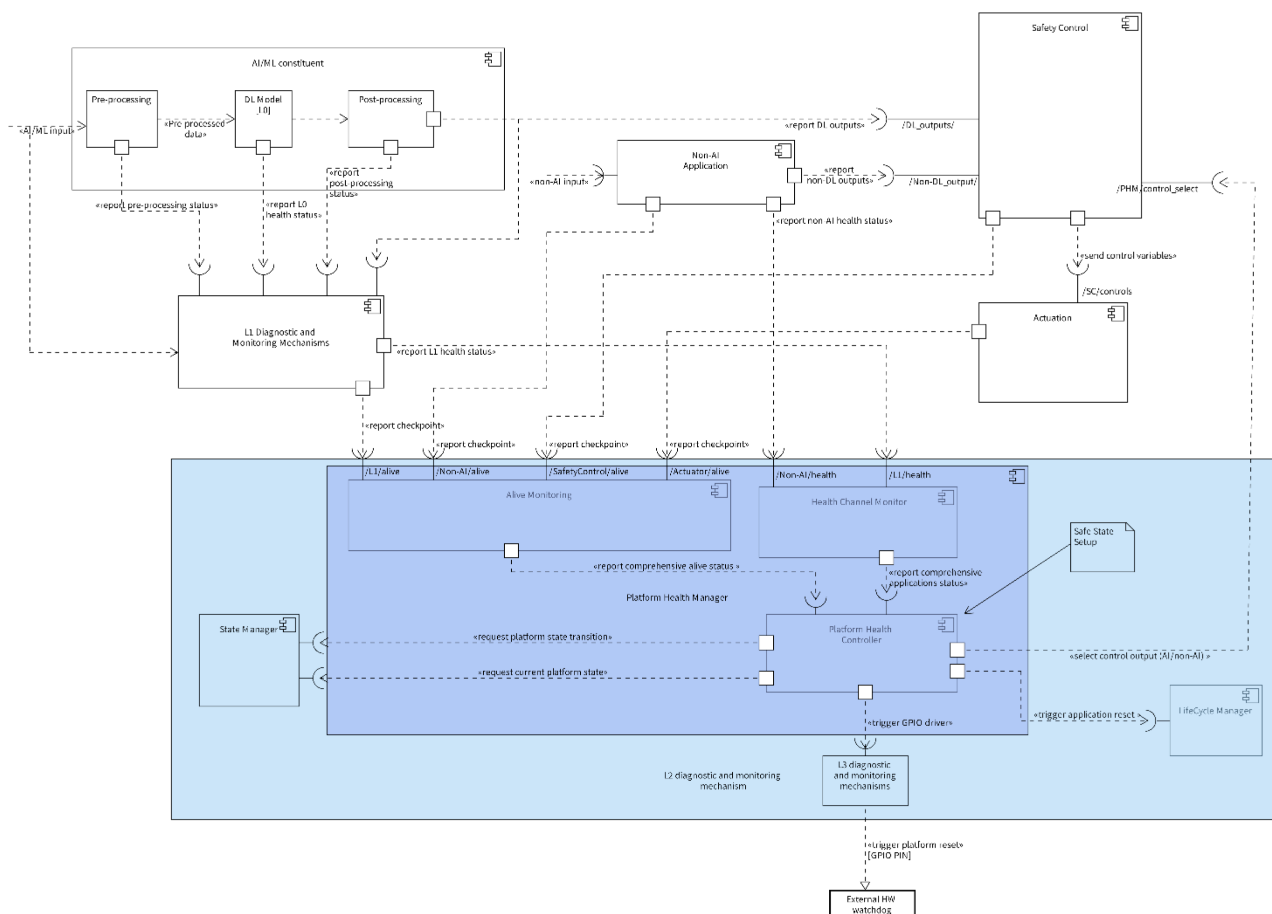


Figure 34 - Middleware architecture overview.

6.2.2 Support to Verification and Validation

The Middleware software architecture accommodates also standard Verification & Validation concepts, as illustrated in Figure 35. CI/CD pipelines enhance the process of merging new code in Gitlab by providing automated tests of three types:

- Unit tests: targets the smallest units of code (e.g. a function or an object).
- Component tests: verifies the correct behaviour of a module, which, in the case of ROS2, is a node. The tests shall verify the correct behaviour of the node by calling its interfaces, regardless the details of the internal implementation.
- Integration tests: verifies that multiple modules of the system are capable of cooperating; through integration tests, we shall make sure that applications can use the core functionalities of platform applications.
    - For example, we shall make sure that the Platform Health Manager will be able to react in case of a violation of a configured temporal, logic or health constraint from an application.

Currently, unit and components tests have been written in C++ on top of the Google Testing and Mocking Framework for some fundamental packages of the system (as the smw_lifecycle_manager).

The current work in progress is in the following fronts:

- Creating the first unit and component tests for code developed in Python, using the pytest framework.
    - The first package targeted to have tests written in Python is the smw_base_application. Since the applications will inherit from the class smw_base_application::BaseApplication, that will also allow developers to have samples of how to test their own modules.
- Improve and measure the coverage of the unit and component tests, so to reveal inadequacy or unintended functionalities.
    - Test coverage can be verified through gcov (C++), coverage.py (Python) or other similar tools.
- Construct easy-to-use utilities that will allow application developers test their packages not only singularly, but in the context of the platform.
    - For example, we'll offer some sample examples in which a Rosbag will be played alongside the test; in this way, application developers can provide data collected by sensors on the field to their application tests.

As a final step, SIL testing can be employed to verify and validate the correct operation of the platform, in case the simulators chosen by the application developers are capable to be integrated with ROS2. Finally, hardware tests with the Jetson Board are foreseen to verify the core functionalities of the platform work as intended as well as the correct interaction with external hardware, since for the L3-level diagnostics the platform health manager shall communicate with an external device (watchdog).

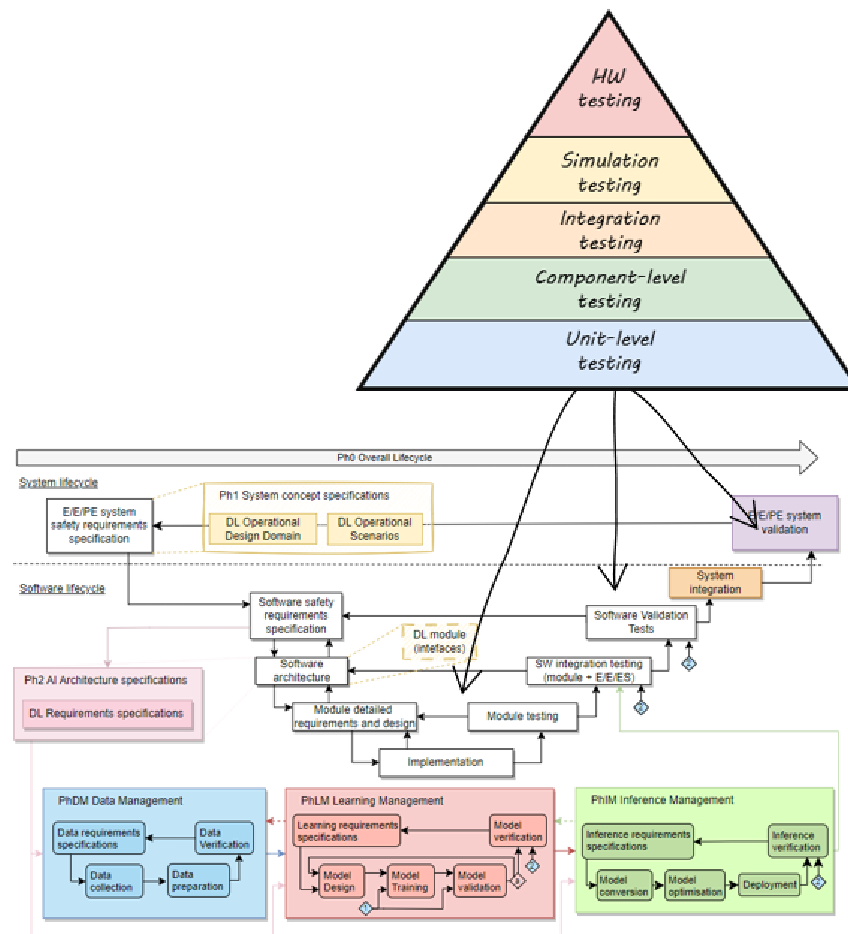


Figure 35 - Overview of the verification and validation process for the platform development.

6.3 Technological assessment

In accordance with the task main objectives, we have been designing and developing a set of software libraries to allow use-case applications to have access to the platform core features. The SAFEXPLAIN Middleware have been identified as an efficient means to provide the required functionalities while at the same time supporting the deployment of Safety Patterns [1].

Delivered tool and positioning in the SAFEXPLAIN stack

At M18, we have been able to consolidate a partial implementation of those libraries, including the main interfaces with the use-case applications. More specifically, this task has delivered:

I. a functional set of libraries that allows the use-case applications to have access to the platform resources (smw_base_application, smw_comm, smw_logging).
II. a functional set of the libraries that constitute the components needed to support the health management of the platform (smw_health_manager, smw_health_client, smw_lifecycle_manager, smw_state_manager, smw_state_client).
III. a prototype library (smw_testing) to support testing on the platform.
IV. a set of examples that allow use-case developers to port their applications to the platform and at the same time meet FUSA architectural constraints.

The developed tools allowed for early testing and initial integration efforts both in the scope if WP4 (PMULib, timing analysis instrumentation, etc.) and WP2 (FUSA architectural patterns) and WP5 (case study porting). Tools are going to be further refined and improved in the next months,

in line with early feedback received while advancing in the porting of the use cases and Safety Patterns to the platform.

Intra-WP dependencies

Within the scope of WP4, the SAFEXPLAIN middleware has been developed to provide an easy integration of the PMULib (Section 4.3) to the use-case applications running on the platform.

- *PMULib integration*: the instrumentation library has been integrated in the Middleware so that it can be deployed automatically to track timing information at node level, allowing configuration, instrumentation, and data collection in a transparent way, without requiring any modification to the functional specification.

Inter-WP contribution and alignment

The contributions of this task are also relevant in the scope of other work packages, and the alignment with them has been ensured through continuous interaction.

WP2: T4.4 provides a concrete implementation of the reference safety architecture proposed by WP2 (T2.3 and T2.4). As a result, the software libraries developed within T4.4 facilitate compliance of the use-case applications with the reference safety architecture and serves as a baseline for deploying the Safety Patterns described in [1].

WP5: T4.4 provides use case developers a set of libraries that render easier the access to communication, logging, diagnostic and safety mechanisms by the use-case applications. Additionally, the Middleware is being developed to provide partially automated V&V support. Finally, a set of examples have been provided for developers to integrate their applications on the platform.

T4.4 Next steps

As previously discussed, we will devote next most of the effort to complete the implementation of the platform health management, to improve the coverage of unit, component, and integration tests as well as to develop utilities to provide automated V&V support. Finally, we will continue to provide support to FUSA architecture mapping, use-case applications porting, by improving and integrate new features and to refine the already existing ones.

7 Acronyms and Abbreviations

CCPLEX	CPU Complex
COTS	Commercial Off The Shelves
CUDA	Compute Unified Device Architecture
DSU-AE	DynamIQ™ Shared Unit
FUSA	Functional Safety
GPC	Graphics Processing Clusters
GPU	Graphics Processing Unit
HEMs	Hardware Event Monitor
MCF	Memory Controller Fabric
OS	Operating System
PMC	Performance Monitoring Counter
PMU	Platform Monitoring Unit
ROS2	Robotic Operating System version 2
RT	Ray Tracing
SCF	System Coherency Fabric
SCU	Snoop Control Unit
SDK	Software Development Kit
SM	Streaming Multiprocessor
V&V	Verification and Validation

8 References

[1] SAFEXPLAIN, "D2.2 DL Safety Architectural Patterns and Platform," 2024.

[2] NVIDIA, "NVIDIA Jetson AGX Orin Developer Kit User Guide," [Online]. Available: https://developer.nvidia.com/embedded/learn/jetson-agx-orin-devkit-user-guide/index.html .

[3] ARM, "Cortex-A78AE," [Online]. Available: https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a78ae.

[4] NVIDIA, "NVIDIA Jetson AGX Orin Series - A Giant Leap Forward for Robotics and Edge AI Applications - Technical Brief," 2022.

[5] "ROS2 - Version 2 of the Robot Operating System (ROS) software stack," [Online]. Available: https://github.com/ros2.

[6] NVIDIA, NVIDIA Orin Series Technical Reference Manual (DP-10508-002), 2022.

[7] NVIDIA, "Integrated GPU cache coherence on Orin," [Online]. Available: https://forums.developer.nvidia.com/t/integrated-gpu-cache-coherence-on-orin/263662.

[8] NVIDIA, "Xavier Series SoC Technical Reference Manual," [Online]. Available: Xavier_TRM_DP09253002.pdf.

[9] S. Vilardell, I. Serra, E. Mezzetti, J. Abella and F. J. Cazorla, "MUCH: exploiting pairwise hardware event monitor correlations for improved timing analysis of complex MPSoCs," in *Symposium on Applied Computing*, 2021.

[10] L. V. e. a. Montiel, "Approximating Joint Probability Distributions Given," *Decision Analysis,* 2013.

[11] D. Gordon, "Covering Designs," [Online]. Available: https://www.dmgordon.org/cover/.

[12] J. Abella, C. Hernandez, E. Quinones, F. J. Cazorla, P. Ryan COnmy, M. Azkarate-askasua, J. Perez, E. Mezzetti and T. Vardanega, "WCET analysis methods: Pitfalls and challenges on their trustworthiness," in *IEEE Symposium on Industrial Embedded Systems (SIES)*, 2015.

[13] I. Agirre, F. J. Cazorla, J. Abella, C. Hernández, E. Mezzetti, M. Azkarate-askatsua and T. Vardanega, "Fitting Software Execution-Time Exceedance into a Residual Random Fault in ISO-26262," in *IEEE Trans. Reliability*, 2018.

[14] SAFEXPLAIN, "D5.1 Case study stubbing and early assessment of case study porting," 2024.

[15] S. Vilardell, I. Serra, E. Mezzetti, J. Abella, F. J. Cazorla and J. del Castillo, "Using Markov's Inequality with Power-Of-k Function for Probabilistic WCET Estimation," in *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, 2022.

[16] L. F. Arcaro, K. P. Silva and R. S. D. Oliveira, "On the Reliability and Tightness of GP and Exponential Models for Probabilistic WCET Estimation," in *ACM Trans. Des. Autom. Electron. Syst.*, 2018.

[17] P. K. Valsan, H. Yun and F. Farshchi, “Taming Non-Blocking Caches to Improve Isolation in Multicore Real-Time Systems,” in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[18] G. Fenandez, J. Jalle, J. Abella, E. Quinones, T. Vardanega and F. J. Cazorla, “Resource usage templates and signatures for COTS multicore processors},” in *ACM Design Automation Conference (DAC)*, 2015.

[19] European Union Aviation Safety Agency (EASA), “EASA Concept Paper: guidance for Level 1 & 2 machine learning applications,,” 2023.

[20] R. Padilla, S. Netto and E. da-Silva, “A Survey on Performance Metrics for Object-Detection Algorithms,” in *International Conference on Systems, Signals and Image Processing (IWSSIP)*, Niteroi, Brazil, 2020.

9 Annex 1 – PMULib interface

In this annex PMULib interfaces are described. The functional documentation refers to the following custom types and constants:

typedef int pmu_result	
static const pmu_result	A78AE_PMU_RESULT_OK = 0
static const pmu_result	A78AE_PMU_RESULT_ERR = -1

9.1 Function Documentation

9.1.1 a78ae_pmu_configure()

pmu_result a78ae_pmu_configure (unsigned int *mask*,
const unsigned int * *events*
)

Configure the counters specified in mask to count the events specified in the events array.

Parameters

mask	A mask of the counters to reconfigure in this call. If the nth bit is set, the nth will be configured to count events[m]
events	Array of event IDs to count. It must contain exactly as many items as bits are set in <mask>.

Returns

A78AE_PMU_RESULT_OK if the operation was successful, a different value otherwise.

9.1.2 a78ae_pmu_counters_available()

unsigned a78ae_pmu_counters_available (void)

Return the number of counters available in the platform for simultaneous use.

Returns

The number of counters that can be used simultaneously in the platform.

9.1.3 a78ae_pmu_read_counters()

pmu_result a78ae_pmu_read_counters (unsigned int *mask*,
uint32_t * *values*
)

Read the counters specified in mask and store its values in the supplied array.

Parameters

mask	A mask of the counters to read in this call. If the nth bit is set to one, the value of counter n will be written to values[m]
values	Array where counter values will be stored. It must contain exactly as many items as bits are set in <mask>.

Returns

A78AE_PMU_RESULT_OK if the operation was successful, a different value otherwise.

9.1.4 a78ae_pmu_reset_counters()

pmu_result a78ae_pmu_reset_counters (unsigned int *mask*)

Reset the counters specified in mask.

Parameters

mask A mask of the counters to reset in this call.

Returns

A78AE_PMU_RESULT_OK if the operation was successful, a different value otherwise.

9.1.5 a78ae_pmu_start()

void a78ae_pmu_start (unsigned int *mask*) inline

Starts counters, causing them to increment when the configured event takes place. Callers MUST NOT assume that all counters are started at the same time.

Parameters

mask Counters to start. Nth counter will be started if the nth bit is set.

9.1.6 a78ae_pmu_start_global()

void a78ae_pmu_start_global (void) inline

Starts all counters globally, allowing all of them to increment. Whether this call is equivalent to pmu_start with all bits set is implementation dependent, but its usage is preferred over the latter, as most PMUs support a global enable/disable in hardware which will be used by this function (if present) but never will for the non-global variant.

9.1.7 a78ae_pmu_stop()

void a78ae_pmu_stop (unsigned int *mask*) inline

Stops counters, preventing them from incrementing. Callers MUST NOT assume that all counters are stopped at the same time.

Parameters

mask Counters to stop. Nth counter will be stopped if the nth bit is set.

9.1.8 a78ae_pmu_stop_global()

void a78ae_pmu_stop_global (void)

Stops all counters globally, preventing all of them from incrementing. Whether this call is equivalent to pmu_stop with all bits set is implementation dependent, but its usage is preferred over the latter, as most PMUs support a global enable/disable in hardware which will be used by this function (if present) but never will for the non-global variant.

9.2 Macro Documentation

Library event values can be configured using the event ID as specified in the manual, or the macros below.

Macro	Value
PMU_A78AE_SW_INCR	0x0
PMU_A78AE_L1I_CACHE_REFILL	0x1
PMU_A78AE_L1I_TLB_REFILL	0x2
PMU_A78AE_L1D_CACHE_REFILL	0x3
PMU_A78AE_L1D_CACHE	0x4
PMU_A78AE_L1D_TLB_REFILL	0x5
PMU_A78AE_INST_RETIRED	0x8
PMU_A78AE_EXC_TAKEN	0x9
PMU_A78AE_EXC_RETURN	0x0A
PMU_A78AE_CID_WRITE_RETIRED	0x0B
PMU_A78AE_BR_MIS_PRED	0x10
PMU_A78AE_CPU_CYCLES	0x11
PMU_A78AE_BR_PRED	0x12
PMU_A78AE_MEM_ACCESS	0x13
PMU_A78AE_L1I_CACHE	0x14
PMU_A78AE_L1D_CACHE_WB	0x15
PMU_A78AE_L2D_CACHE	0x16
PMU_A78AE_L2D_CACHE_REFILL	0x17
PMU_A78AE_L2D_CACHE_WB	0x18
PMU_A78AE_BUS_ACCESS	0x19
PMU_A78AE_MEMORY_ERROR	0x1A
PMU_A78AE_INST_SPEC	0x1B
PMU_A78AE_TTBR_WRITE_RETIRED	0x1C
PMU_A78AE_BUS_MASTER_CYCLE	0x1D
PMU_A78AE_COUNTER_OVERFLOW	0x1E
PMU_A78AE_CACHE_ALLOCATE	0x20
PMU_A78AE_BR_RETIRED	0x21
PMU_A78AE_BR_MIS_PRED_RETIRED	0x22
PMU_A78AE_STALL_FRONTEND	0x23

PMU_A78AE_STALL_BACKEND	0x24
PMU_A78AE_L1D_TLB	0x25
PMU_A78AE_L1I_TLB	0x26
PMU_A78AE_L3D_CACHE_ALLOCATE	0x29
PMU_A78AE_L3D_CACHE_REFILL	0x2A
PMU_A78AE_L3D_CACHE	0x2B
PMU_A78AE_L2TLB_REFILL	0x2D
PMU_A78AE_L2TLB_REQ	0x2F
PMU_A78AE_REMOTE_ACCESS	0x31
PMU_A78AE_DTLB_WLK	0x34
PMU_A78AE_ITLB_WLK	0x35
PMU_A78AE_LL_CACHE_RD	0x36
PMU_A78AE_LL_CACHE_MISS_RD	0x37
PMU_A78AE_L1D_CACHE_LMISS_RD	0x39
PMU_A78AE_OP_RETIRED	0x3A
PMU_A78AE_OP_SPEC	0x3B
PMU_A78AE_STALL	0x3C
PMU_A78AE_STALL_SLOT_BACKEND	0x3D
PMU_A78AE_STALL_SLOT_FRONTEND	0x3E
PMU_A78AE_STALL_SLOT	0x3F
PMU_A78AE_L1D_CACHE_RD	0x40
PMU_A78AE_L1D_CACHE_WR	0x41
PMU_A78AE_L1D_CACHE_REFILL_RD	0x42
PMU_A78AE_L1D_CACHE_REFILL_WR	0x43
PMU_A78AE_L1D_CACHE_REFILL_INNER	0x44
PMU_A78AE_L1D_CACHE_REFILL_OUTER	0x45
PMU_A78AE_L1D_CACHE_WB_VICTIM	0x46
PMU_A78AE_L1D_CACHE_WB_CLEAN	0x47
PMU_A78AE_L1D_CACHE_INVAL	0x48
PMU_A78AE_L1D_TLB_REFILL_RD	0x4C
PMU_A78AE_L1D_TLB_REFILL_WR	0x4D
PMU_A78AE_L1D_TLB_RD	0x4E
PMU_A78AE_L1D_TLB_WR	0x4F
PMU_A78AE_CACHE_ACCESS_RD	0x50
PMU_A78AE_CACHE_ACCESS_WR	0x51
PMU_A78AE_CACHE_RD_REFILL	0x52
PMU_A78AE_CACHE_WR_REFILL	0x53
PMU_A78AE_CACHE_WRITEBACK_VICTIM	0x56
PMU_A78AE_CACHE_WRITEBACK_CLEAN_COH	0x57
PMU_A78AE_L2CACHE_INV	0x58
PMU_A78AE_L2TLB_RD_REFILL	0x5C
PMU_A78AE_L2TLB_WR_REFILL	0x5D
PMU_A78AE_L2TLB_RD_REQ	0x5E
PMU_A78AE_L2TLB_WR_REQ	0x5F
PMU_A78AE_BUS_ACCESS_REQ	0x60
PMU_A78AE_BUS_ACCESS_RETRY	0x61

PMU_A78AE_MEM_ACCESS_RD	0x66
PMU_A78AE_MEM_ACCESS_WR	0x67
PMU_A78AE_UNALIGNED_LD_SPEC	0x68
PMU_A78AE_UNALIGNED_ST_SPEC	0x69
PMU_A78AE_UNALIGNED_LDST_SPEC	0x6A
PMU_A78AE_LDREX_SPEC	0x6C
PMU_A78AE_STREX_PASS_SPEC	0x6D
PMU_A78AE_STREX_FAIL_SPEC	0x6E
PMU_A78AE_STREX_SPEC	0x6F
PMU_A78AE_LD_SPEC	0x70
PMU_A78AE_ST_SPEC	0x71
PMU_A78AE_DP_SPEC	0x73
PMU_A78AE_ASE_SPEC	0x74
PMU_A78AE_VFP_SPEC	0x75
PMU_A78AE_PC_WRITE_SPEC	0x76
PMU_A78AE_CRYPTO_SPEC	0x77
PMU_A78AE_BR_IMMED_SPEC	0x78
PMU_A78AE_BR_RETURN_SPEC	0x79
PMU_A78AE_BR_INDIRECT_SPEC	0x7A
PMU_A78AE_ISB_SPEC	0x7C
PMU_A78AE_DSB_SPEC	0x7D
PMU_A78AE_DMB_SPEC	0x7E
PMU_A78AE_EXC_UNDEF	0x81
PMU_A78AE_EXC_SVC	0x82
PMU_A78AE_EXC_PABORT	0x83
PMU_A78AE_EXC_DABORT	0x84
PMU_A78AE_EXC_IRQ	0x86
PMU_A78AE_EXC_FIQ	0x87
PMU_A78AE_EXC_SMC	0x88
PMU_A78AE_EXC_HVC	0x8A
PMU_A78AE_EXC_TRAP_PABORT	0x8B
PMU_A78AE_EXC_TRAP_DABORT	0x8C
PMU_A78AE_EXC_TRAP_OTHER	0x8D
PMU_A78AE_EXC_TRAP_IRQ	0x8E
PMU_A78AE_EXC_TRAP_FIQ	0x8F
PMU_A78AE_RC_LD_SPEC	0x90
PMU_A78AE_RC_ST_SPEC	0x91
PMU_A78AE_L3_CACHE_RD	0xA0
PMU_A78AE_CNT_CYCLES	0x4004
PMU_A78AE_STALL_BACKEND_MEM	0x4005
PMU_A78AE_L1I_CACHE_LMISS	0x4006
PMU_A78AE_L2D_CACHE_LMISS_RD	0x4009
PMU_A78AE_L3D_CACHE_LMISS_RD	0x400B
PMU_SCF_BUS_ACCESS	0x10190
PMU_SCF_BUS_ACCESS_RD	0x10600
PMU_SCF_BUS_ACCESS_WR	0x10610

PMU_SCF_BUS_ACCESS_SHARED	0x10620
PMU_SCF_BUS_ACCESS_NOT_SHARED	0x10630
PMU_SCF_BUS_ACCESS_NORMAL	0x10640
PMU_SCF_BUS_ACCESS_PERIPH	0x10650
PMU_SCF_BUS_CYCLES	0x101d0
PMU_SCF_CACHE	0x10f20
PMU_SCF_CACHE_ALLOCATE	0x10f00
PMU_SCF_CACHE_REFILL	0x10f10
PMU_SCF_CACHE_WB	0x10f30

9.3 Usage Example

```
#include "a78ae-pmu.h"

int main() {
        const unsigned int mask = 0b111111;
        const unsigned int events[] =
{ PMU_A78AE_L1D_CACHE_REFILL, PMU_A78AE_INST_RETIRED, 0x11, 0x17, 0xA0, 0x400B };
        int values[6];
        if(a78ae_pmu_configure(mask, events) != 0) {
                printf("ERROR configuring the events\n");
                exit(-1);
        }
        a78ae_pmu_reset_counters(mask);
        a78ae_pmu_start_global();
        for (volatile int i = 0; i< 1000; i++); // counting events for this loop
        a78ae_pmu_stop_global();
        if (a78ae_pmu_read_counters(mask, values) != 0) {
                printf("ERROR reading the events\n");
                exit(-1);
        }
        printf("%d,%d,%d,%d,%d,%d\n", values[0],  values[1],  values[2],  values[3],
values[4],  values[5] );
}
```