D4.2 Platform Technologies Report

Version 1.0

Documentation Information

Contract Number	101069595
Project Website	www.safexplain.eu
Contratual Deadline	31.03.2025
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.

Change Log

Version	Description Change
V0.1	First draft
V0.2	First release after internal review
V1.0	Final version

Table of Contents

Executive Summary

This deliverable reports on the technical and technological progresses achieved in WP4 during the **Phase 3** of the project, spawning from m19 to m30. In particular, this report captures the advancements and final outcomes of WP4 tasks T4.1-T4.4 (including refinements and support activities under T4.5) by **MS3** hence covering further developments of SAFEXPLAIN solutions at hardware (HW) and software (SW) level and their consolidation and integration on top of the SAFEXPLAIN execution platform to support the activities of other work packages and enable the evaluation of the case studies. As this deliverable is a natural extension and update to D4.1, whenever possible we will reference the latter to avoid redundancies.

Similarly to the approach followed for D4.1, we provide an assessment on the achievement of the main objectives for each WP4 task and relate them to respective outcomes (technologies and tools and integration in the execution platform).

This deliverable does not provide a detailed description of the specific integration of WP4 solutions with WP5 use cases and their evaluation. These activities are still ongoing, with WP4 activities falling in the scope of T4.5.

1 Introduction

This document reports on the progress achieved in the scope of WP4 during the third phase of the project. 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 this same work package (WP4) and other technical work packages (WP2 and WP3) and supporting the deployment of SAFEXPLAIN case studies (WP5) on top of SAFEXPLAIN execution platform.

1.1 Scope

The scope of WP4 consists in intercepting all platform-level requirements and constraints and offering a FUSA-compliant and high-performance execution platform. To this extent WP4 develops through 4 main tasks and a higher-level meta-task to support the integration of WP2 and WP3 solutions in the case studies. Additionally, in the final phase 4 of the project, the support task T4.5 is deployed in order to capture further refinements and adaptations emerging in the final tailoring and evaluation phase. Therefore, WP4 has strict relations with all SAFEXPLAIN work packages and, in fact, facilitates their alignment. Figure 1 below, taken from D4.1, captures the main tasks in WP4 and how they support SAFEXPLAIN technologies and integration by capturing explicit and implicit requirements from other WPs.

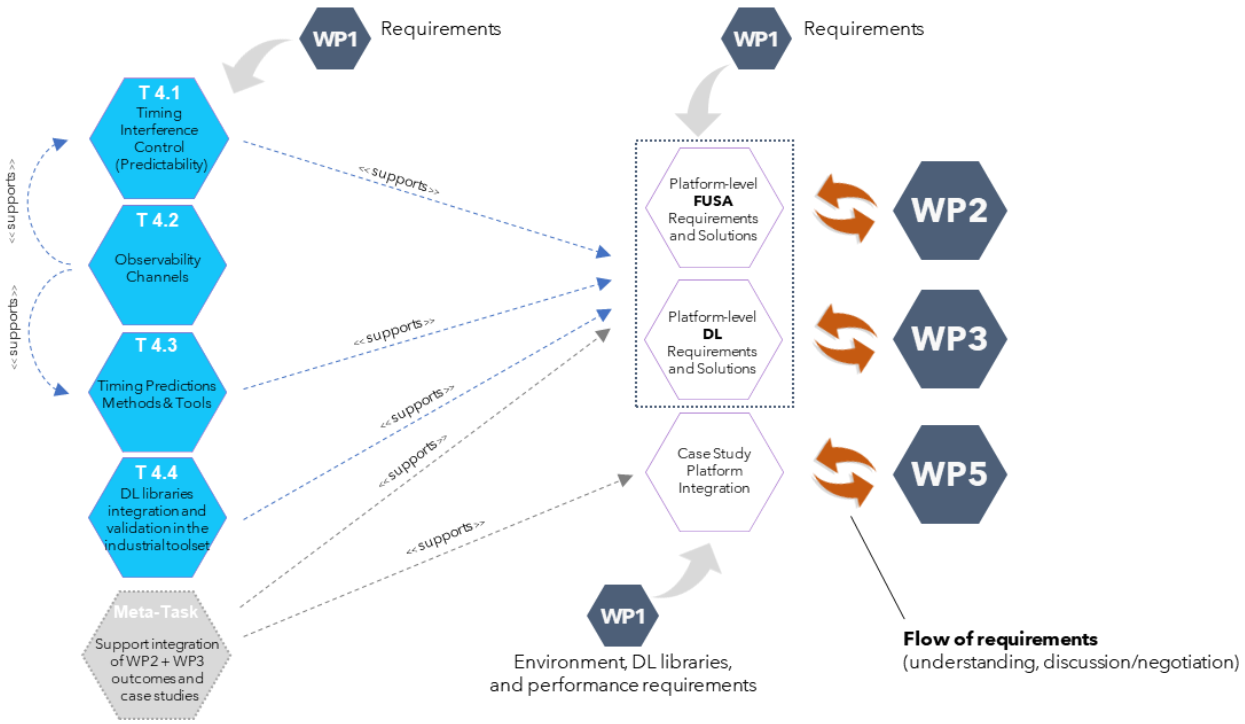

Figure 1 - WP4 role and relation with other WPs [1].

We recall below the scope and objectives of the main technological tasks in WP4:

- ***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 [2]. T4.1 finished at m24.

- ***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 tasks through offline V&V activities and run-time monitoring.
- ***T4.5 Refinements and integration updates***, capturing final refinements and tailoring of WP4 solutions in order to closely support the execution and assessment of the project use cases. T4.5 started at m25, right after T4.1 termination.

1.2 Structure of the Document

In the following sections we provide a review of WP4 activities and progresses up to MS3. Large part of the hardware analysis and conceptualization of SAFEXPLAIN software solutions have been already captured in D4.1. The reader is encouraged to refer to that document, especially for all aspects related to the SAFEXPLAIN target hardware platform (NVIDIA AGX Orin [3]) and system software stack.

The structure of the document will follow the task structure of the WP. Each section will include a short recap of the task objectives, the strategy followed, the obtained results, and an assessment thereof with respect to integration on the execution platform and adaptations to the project case studies.

2 Updates to the hardware and software stack

The NVIDIA AGX Orin [3] has been selected as the common target platform for the case studies project, for its representativeness from the FUSA perspective and its capability to sustain the execution of performance intensive AI-based applications, hence providing support for general-purpose and AI-specific hardware accelerators. Relevant details on the target features are extensively covered in D4.1 [1].

Below we summarize the changes/upgrades on the hardware setup and software stack.

2.1 NVIDIA AGX Orin setup

The NVIDIA Jetson AGX Orin is a family of heterogenous MPSoC (Orin 32/64 Nano) developed by NVIDIA to cover the emerging requirements from 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 [4] each, a NVIDIA Ampere GPU, ad-hoc AI-oriented accelerators 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 enable the execution of multiple concurrent AI applications.

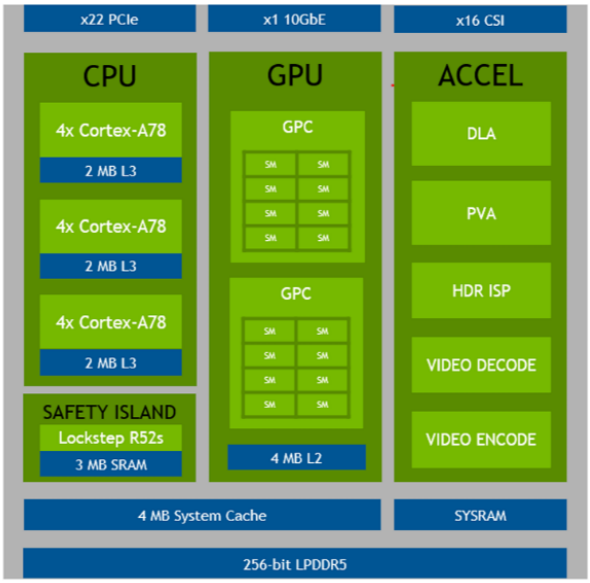

Figure 2 - Block Diagram of our target platform (from [5]).

The amount of DRAM memory provided by the selected 32GB AGX Orin Dev Kit, while adequate for the deployment of WP4 software solutions and user applications, resulted to be quite limited for a flexible and efficient development environment. For this reason, we extended the memory capability of the board by connecting a 1TB Non-Volatile Memory express (NVMe) module. The addition of such module is not a stringent requirement for the execution of the SAFEXPLAIN stack and solutions.

2.2 Default software stack

The NVIDIA AGX Orin [3] 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 - NVIDIA AGX Orin

The specific support for the target hardware was not fully consolidated at the beginning of SAFEXPLAIN and many software libraries were only provided in older versions. We were therefore expecting changes to the setup to happen during the project. After checking the compatibility of the updates with the partners assumptions and requirements, we opted for moving to the latest release of the Jetson Linux and JetPack, which also involved an update to the supported AI libraries. Figure 4 illustrates the latest versions for the low-level software layer in the SAFEXPLAIN stack.

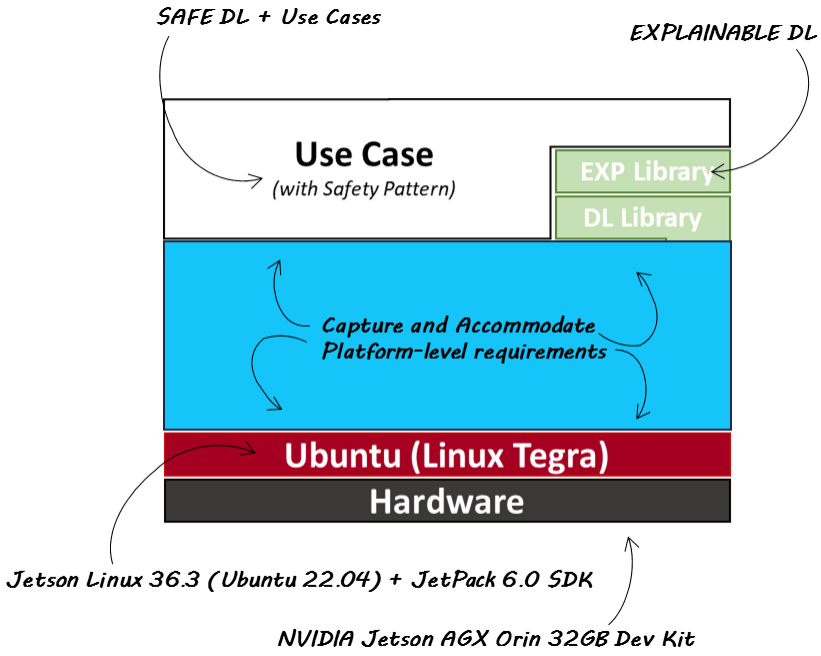

Figure 4 - Updated SW stack.

Updated SAFEXPLAIN setup consists in the following elements and versions:

- Jetson Linux 36.3 (Ubuntu 22.04)
- Linux Tegra 5.15
- JetPack 6.0.1 SDK

Specific libraries

- TensorRT: 8.6.2
- CuDNN: 8.9.4.25
- CUDA: 12.2.12-1
- OpenCV python: 4.10.0-dev
- Python3: 3.10.12
- PyTorch: 2.5.0
- Vulkan: 1.3.204
- Vulkan SC: 36.0

3 Timing Interference Control (T4.1)

During the third phase of the project, Task 4.1 kept focusing on timing interference related aspects, refining the results from the analysis of interference channels and available support in the execution platform. The work led to the identification of a set of HW/SW level solutions for limiting timing interference either by construction, in alignment with WP2 strategy [2], or by deploying ad hoc run-time mechanisms. It is worth noting that T4.1 has been active for just 6 months in this phase of the project, hence half of the time allocated for T4.2-T4.4.

In terms of the contents of this section, Section 3.1 reports on the results from applying kernel-level solutions on the underlying OS in order to reduce the OS-related variability and guarantee more predictable execution times. Section 3.2, instead, focuses on hardware and software related aspects and reports on the updates on the HW/SW configurations for supporting the required degrees of segregation while meeting the performance requirements.

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 as opposed to real-time ones, introduces some interference or jitter in the execution of tasks stemming from the variability incurred by the many system calls and background activities the OS is undergoing. As real-time OSes were not ready and available on the target platform at the beginning of the project, we opted for a middle ground solution consisting in exploiting two complementary approaches: (i) applying the real-time (RT) patch to the Linux kernel and (ii) exploiting the capability to force system calls and interrupts to a subset of the cores. It is noted that the RT patch [6] (recently accepted in the mainstream development branch) is at the basis of the many commercial real-time Linux distributions.

3.1.1 Kernel interference mitigation

The Linux OS kernel can become a source of interference that may affect the execution time of other software running in the system. This is always true, but it can become especially apparent when the system is under heavy workloads, as the kernel will schedule out some tasks if there are not any free cores available.

The Linux kernel provides a patch [6] to replace the default scheduler with `PREEMPT_RT`, which is better suited to run critical tasks as it is intended to reduce latency and hence execution time variability. NVIDIA provides instructions for kernel customization[1][2] that can be followed in order to replace the default kernel for one with `PREEMPT_RT` enabled.
However, even when using a patched kernel, critical tasks may become affected by OS noise when the system is under heavy loads. A simple improvement is to configure such tasks as high priority. This can be done, for instance, using `pthread_attr_setschedparam` and setting the `sched_priority` to a high value, but this is insufficient if the kernel still needs to interrupt our critical task.
In parallel, the impact of the kernel can be mitigated by configuring it to force some cores to be isolated and configuring some others to take care of interrupt requests. To this end, we can leverage the clustered architecture of the Orin AGX board and reserve a single cluster for all OS

[1] https://docs.nvidia.com/jetson/archives/r36.2/DeveloperGuide/SD/Kernel/KernelCustomization.html
[2] https://docs.nvidia.com/jetson/archives/r36.2/DeveloperGuide/AT/JetsonLinuxToolchain.html#at-jetsonlinuxtoolchain

related activities, while keeping the other two for critical tasks. This can be achieved by adding the following parameters to the kernel boot `extlinux.conf` configuration:

```
isolcpu=4-11 irqaffinity=0-3
```

In principle, different `cpuisol` sets can be configured but the above setup is the one providing better isolation, if the overall load of the system allows reserving the full cluster 0 to system services.

We performed an assessment of the effectiveness of these measures in reducing the OS impact. We applied the Linux kernel RT patch using the instructions provided by NVIDIA and configured the boot options with `isolcpu` and `irqaffinity` as described.

3.1.1.1 Experimental setup

To evaluate the proposed mechanism, we use the `cyclictest` tool[3]. `Cyclictest` is a Linux kernel tool that accurately and repeatedly measures the difference between a thread's intended wake-up time and the time at which it actually wakes up in order to provide statistics about the system's latencies. It can measure latencies in real-time systems caused by the hardware, the firmware, and the operating system.

The installation can be performed with the following set of commands:

```
apt-get install build-essential libnuma-dev
git clone git://git.kernel.org/pub/scm/utils/rt-tests/rt-tests.git
cd rt-tests
git checkout stable/v1.0
make all
make install
```

Output Explanations:

- **T: (Thread Number)**: This indicates the thread number.
- **(PID)**: The Process ID (PID) of the thread.
- **P: (Priority)**: The priority of the thread. The higher the number, the higher the priority
- **I: (Interval)**: The interval at which the thread wakes up, in microseconds. Here, it is set to 1000 microseconds (1 millisecond), as specified with the `--interval` option.
- **C: (Count)**: The count of wakeups or iterations that the thread has performed.
- **Min (Minimum Latency)**: The minimum latency observed, in microseconds, for the wakeup.
- **Act (Actual Latency)**: The actual latency observed for the most recent wakeup, in microseconds.
- **Avg (Average Latency)**: The average latency observed over all wakeups, in microseconds.
- **Max (Maximum Latency)**: The maximum latency observed, in microseconds, for any wakeup.

Interpreting the Output:

- **Min, Act, Avg, and Max Latencies**: These values are crucial for understanding the real-time performance. Ideally, these should be as low as possible and close to each other.

[3] https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start

Significant differences between the minimum and maximum latencies can indicate variability and potential issues in meeting real-time deadlines.

- **Consistency Across Threads**: Consistent values across different threads (T: 0, T: 1, etc.) suggest that the system handles real-time scheduling uniformly. Significant discrepancies might indicate CPU affinity issues or uneven load distribution.
- **Max Latency**: The maximum latency is particularly important in real-time systems, as it represents the worst-case scenario. This value helps determine if the system can meet the most stringent real-time requirements.

We run `cyclictest` in four separate scenarios: for high and low priority tasks, and for the regular kernel provided by NVIDIA and a custom kernel with the RT patch applied. We run `cyclictest` with the following parameters and let it collect 4000 samples or more for wakeups for each thread:

```
cyclictest  -a -t -n -p99 # For high priority tasks
cyclictest   -a -t -n # For normal priority tasks
```

3.1.1.2 Results

We analyse the results and chart the maximum latency, as the latter is the metric that better identifies execution time variability. In the tables below we present the complete results for the non-RT and the RT kernels, then we proceed to present the most relevant differences.

	Non-RT kernel
High priority	T: 0 (52671) P:99 I:1000 C: 28099 Min: 1 Act: 3 Avg: 3 Max: 41 T: 1 (52672) P:99 I:1500 C: 18732 Min: 2 Act: 2 Avg: 2 Max: 38 T: 2 (52673) P:99 I:2000 C: 14049 Min: 2 Act: 2 Avg: 3 Max: 38 T: 3 (52674) P:99 I:2500 C: 11239 Min: 2 Act: 2 Avg: 3 Max: 34 T: 4 (52675) P:99 I:3000 C: 9366 Min: 2 Act: 3 Avg: 3 Max: 15 T: 5 (52676) P:99 I:3500 C: 8028 Min: 2 Act: 5 Avg: 3 Max: 12 T: 6 (52677) P:99 I:4000 C: 7024 Min: 2 Act: 3 Avg: 2 Max: 10 T: 7 (52678) P:99 I:4500 C: 6244 Min: 2 Act: 5 Avg: 3 Max: 13 T: 8 (52679) P:99 I:5000 C: 5619 Min: 2 Act: 3 Avg: 3 Max: 18 T: 9 (52680) P:99 I:5500 C: 5108 Min: 2 Act: 7 Avg: 4 Max: 17 T:10 (52681) P:99 I:6000 C: 4683 Min: 2 Act: 3 Avg: 3 Max: 14 T:11 (52682) P:99 I:6500 C: 4322 Min: 2 Act: 4 Avg: 3 Max: 11
Normal priority	T: 0 (52686) P: 0 I:1000 C: 26910 Min: 13 Act: 54 Avg: 52 Max: 99 T: 1 (52687) P: 0 I:1500 C: 17940 Min: 19 Act: 52 Avg: 52 Max: 324 T: 2 (52688) P: 0 I:2000 C: 13454 Min: 40 Act: 52 Avg: 53 Max: 2357 T: 3 (52689) P: 0 I:2500 C: 10764 Min: 51 Act: 52 Avg: 53 Max: 60 T: 4 (52690) P: 0 I:3000 C: 8968 Min: 24 Act: 54 Avg: 54 Max: 7051 T: 5 (52691) P: 0 I:3500 C: 7688 Min: 52 Act: 53 Avg: 52 Max: 198 T: 6 (52692) P: 0 I:4000 C: 6727 Min: 51 Act: 55 Avg: 52 Max: 68 T: 7 (52693) P: 0 I:4500 C: 5980 Min: 18 Act: 56 Avg: 54 Max: 497 T: 8 (52694) P: 0 I:5000 C: 5382 Min: 6 Act: 56 Avg: 53 Max: 373 T: 9 (52695) P: 0 I:5500 C: 4892 Min: 20 Act: 53 Avg: 54 Max: 67 T:10 (52696) P: 0 I:6000 C: 4485 Min: 18 Act: 52 Avg: 53 Max: 87 T:11 (52697) P: 0 I:6500 C: 4140 Min: 5 Act: 53 Avg: 53 Max: 61

	RT kernel
High priority	T: 0 (2554) P:99 I:1000 C: 102510 Min: 1 Act: 7 Avg: 5 Max: 36 T: 1 (2555) P:99 I:1500 C: 68340 Min: 1 Act: 8 Avg: 13 Max: 219 T: 2 (2556) P:99 I:2000 C: 51255 Min: 1 Act: 6 Avg: 6 Max: 32 T: 3 (2557) P:99 I:2500 C: 41003 Min: 1 Act: 2 Avg: 6 Max: 32 T: 4 (2558) P:99 I:3000 C: 34170 Min: 2 Act: 3 Avg: 2 Max: 6 T: 5 (2559) P:99 I:3500 C: 29288 Min: 1 Act: 2 Avg: 2 Max: 5 T: 6 (2560) P:99 I:4000 C: 25627 Min: 2 Act: 2 Avg: 2 Max: 7 T: 7 (2561) P:99 I:4500 C: 22780 Min: 2 Act: 2 Avg: 2 Max: 7 T: 8 (2562) P:99 I:5000 C: 20501 Min: 2 Act: 2 Avg: 2 Max: 8 T: 9 (2563) P:99 I:5500 C: 18638 Min: 2 Act: 3 Avg: 2 Max: 6

	T:10 (2564) P:99 I:6000 C:	17084	Min:	2	Act:	3	Avg:	2	Max:	6
	T:11 (2565) P:99 I:6500 C:	15770	Min:	2	Act:	3	Avg:	2	Max:	6
Normal priority	T: 0 (2571) P: 0 I:1000 C:	42203	Min:	4	Act:	54	Avg:	58	Max:	1356
	T: 1 (2572) P: 0 I:1500 C:	28137	Min:	10	Act:	55	Avg:	75	Max:	1276
	T: 2 (2573) P: 0 I:2000 C:	21103	Min:	3	Act:	55	Avg:	60	Max:	293
	T: 3 (2574) P: 0 I:2500 C:	16882	Min:	5	Act:	69	Avg:	61	Max:	236
	T: 4 (2575) P: 0 I:3000 C:	14068	Min:	52	Act:	52	Avg:	53	Max:	63
	T: 5 (2576) P: 0 I:3500 C:	12059	Min:	52	Act:	53	Avg:	53	Max:	62
	T: 6 (2577) P: 0 I:4000 C:	10551	Min:	52	Act:	53	Avg:	53	Max:	62
	T: 7 (2578) P: 0 I:4500 C:	9379	Min:	52	Act:	54	Avg:	53	Max:	60
	T: 8 (2579) P: 0 I:5000 C:	8441	Min:	52	Act:	53	Avg:	53	Max:	59
	T: 9 (2580) P: 0 I:5500 C:	7673	Min:	52	Act:	53	Avg:	53	Max:	61
	T:10 (2581) P: 0 I:6000 C:	7034	Min:	52	Act:	53	Avg:	53	Max:	62
	T:11 (2582) P: 0 I:6500 C:	6493	Min:	52	Act:	53	Avg:	53	Max:	56

Next, we plot the maximum latency in microseconds and compare the RT kernel vs the non-RT kernel, high vs normal priority tasks, and isolated vs non-isolated cores.

First of all, we evaluate the effectiveness of setting task priority to a high value when we apply the RT kernel patch. In Figure 5 we can clearly see the difference between normal priority (left) which causes latencies between 56 us and 1356 us, and high priority (right) which only causes latencies between 5 us and 219 us.

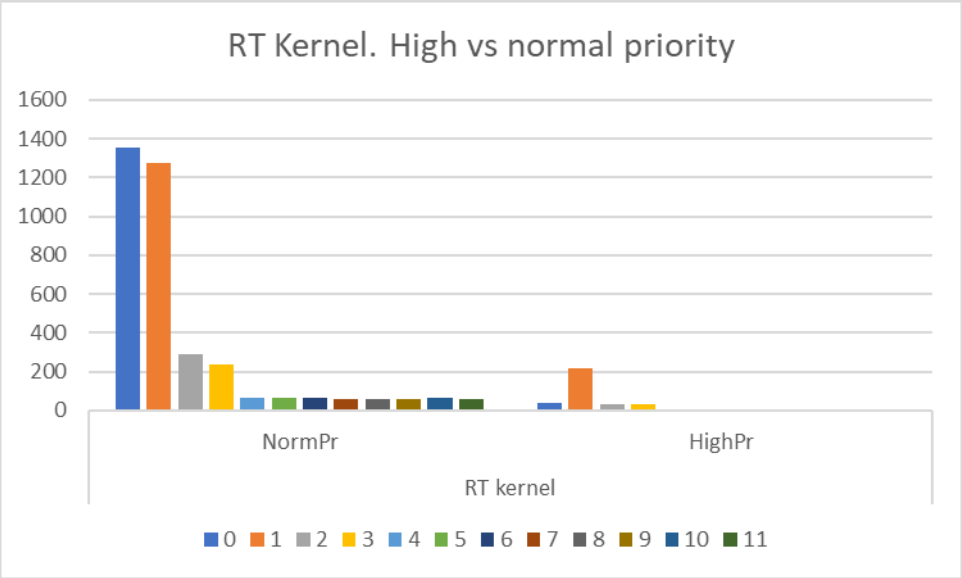

Figure 5 – RT kernel latency, high vs normal priority tasks, all cores.

Even when configuring high priority tasks, it can be clearly seen in Figure 6 that `isolcpu` and `irqaffinity` settings have a notable effect on task latency. Latencies for isolated cores in clusters 1 and 2 experience latencies of up to 8 us, while high priority tasks running in cluster 0 (not isolated and devoted to irq management) still exhibit latencies of up to 219 us. These results confirm our expectation on the complementarity of the two approaches.

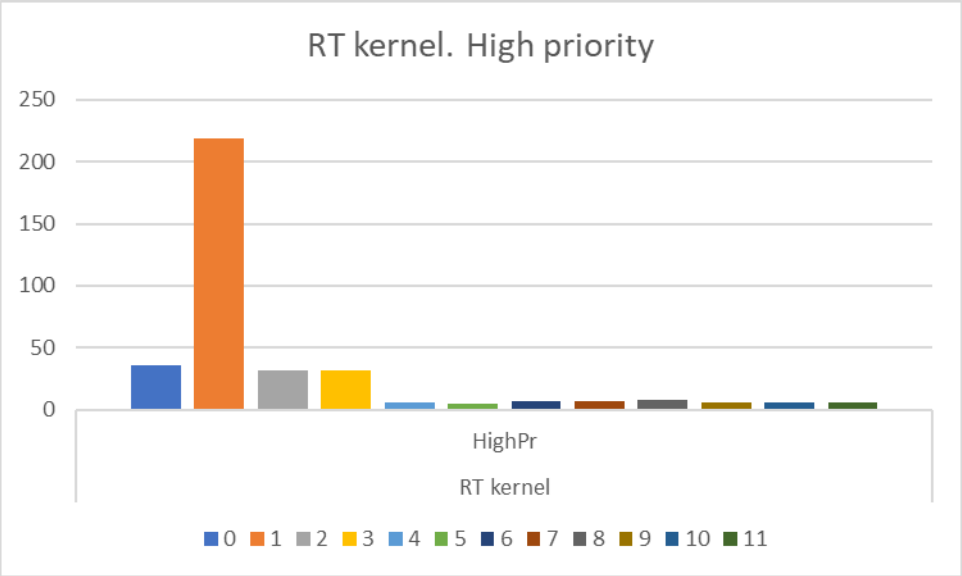

Figure 6 – RT kernel latency, isolated cores vs non-isolated clusters, high priority tasks.

Last, we assess the effectiveness of using a custom kernel with the real time patch applied, compared to the default kernel provided by NVIDIA consistently with an `isolcpu` and `irqaffinity` configuration. In Figure 7 we show latencies for high priority task in cores 4 to 11 (the `isolcpu` set) for each kernel. We observe latencies of up to 18us when running on the default kernel and always below 8 us in the RT-patched kernel.

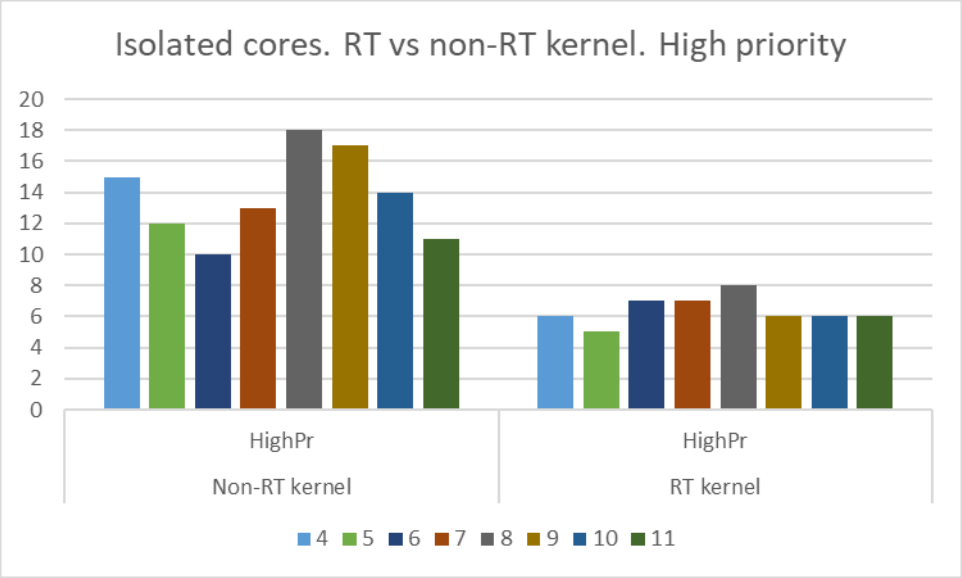

Figure 7 – RT vs non-RT kernel latency, isolated cores, high priority tasks.

The main conclusions that can be extracted from this data are:

- Tasks need to be configured as high priority. Reduces at least 10x, even in non-isolated cores.
- Tasks in non-isolated cores may suffer wake-up latencies up to 37x higher than on isolated cores. Critical tasks should always be configured in isolated cores
- We observe wake-up latencies between 25% and 3.6x higher in non-RT kernels. The RT patch should be applied and used when running critical tasks in this platform.

3.2 Configurations for interference mitigation

The SAFEXPLAIN execution platform combines the NVIDA Orin [3], as target hardware, and system software layer, comprising a tailored Ubuntu distribution. One of the main objectives in SAFEXPLAIN consisted in the identification of hardware and software support for controlling the sources of timing interference arising from contention on shared hardware resources in MPSoCs.

In this phase of the project, we consolidated the results obtained from the analysis of the Orin hardware architecture and its constituents. Interested readers may refer to [1] for more details on relevant hardware components and their expected impact on timing interference.

As an incremental contribution to the previous phases, we concluded the empirical assessment of hardware and software support for interference mitigation by experimenting with the impact of two main Orin configurations affecting both performance and segregation.

3.2.1 Platform level support and configurations

3.2.1.1 Power model impact

The Orin platform provides practical tools for configuring and enforcing a specific power model among those supported in the hardware. Power models are generally defining minimum and maximum frequency of execution for each computing element (core, gpu, etc.). They are also responsible for selectively enabling/disabling components in the platform. For example, power-modes determine how many core clusters are active on the boards, or how many SP are enabled in the GPU. Besides affecting performance, power models also configure the level of parallelism supported by the platform.

Power-modes can be selected or enabled via simple command line tool `nvpmodel` that allows to select among the available power configurations. The Orin AGX provides 4 pre-configured power models with different energy consumption caps (from 15W to 50W). Power models' main features are summarized in Table 1 below. Power model ID=0 is the *default* power model.

	POWER_MODEL	ID=0 (MAXN)	ID=1 (MODE_15W)	ID=2 (MODE_30W)	ID=3 (MODE_50W)
CPU_ONLINE	CORE_0	1	1	1	1
CPU_ONLINE	CORE_1	1	1	1	1
CPU_ONLINE	CORE_2	1	1	1	1
CPU_ONLINE	CORE_3	1	1	1	1
CPU_ONLINE	CORE_4	1	0	1	1
CPU_ONLINE	CORE_5	1	0	1	1
CPU_ONLINE	CORE_6	1	0	1	1
CPU_ONLINE	CORE_7	1	0	1	1
CPU_ONLINE	CORE_8	1	0	0	1
CPU_ONLINE	CORE_9	1	0	0	1
CPU_ONLINE	CORE_10	1	0	0	1
CPU_ONLINE	CORE_11	1	0	0	1
TPC_POWER_GATING	TPC_PG_MASK	0	248	240	0
GPU_POWER_CONTROL_ENABLE	GPU_PWR_CNTL_EN	on	on	on	on
CPU_A78_0	MIN_FREQ	0	0	0	0
CPU_A78_0	MAX_FREQ	-1	1113600	1728000	1497600
CPU_A78_1	MIN_FREQ	0		0	0

CPU_A78_1	MAX_FREQ	-1		1728000	1497600
CPU_A78_2	MIN_FREQ	0			0
CPU_A78_2	MAX_FREQ	-1			1497600
GPU	MIN_FREQ	0	0	0	0
GPU	MAX_FREQ	-1	420750000	624750000	828750000
GPU_POWER_CONTROL_DISABLE	GPU_PWR_CNTL_DIS	auto	auto	auto	auto
DLA0_CORE	MAX_FREQ	-1	614400000	1369600000	1369600000
DLA1_CORE	MAX_FREQ	-1	614400000	1369600000	1369600000
DLA0_FALCON	MAX_FREQ	-1	294400000	729600000	729600000
DLA1_FALCON	MAX_FREQ	-1	294400000	729600000	729600000
PVA0_VPS	MAX_FREQ	-1	704000000	704000000	704000000
PVA0_AXI	MAX_FREQ	-1	486400000	486400000	486400000

Table 1 - AGX Orin supported power modes.

Power models lead to different hardware configurations by limiting the subset of hardware elements that are enabled at run time. Also, the model defines the operational frequency interval for each component, that is the min and max frequency at which a core, a GPU, or a specific accelerator (e.g., PVA) may operate. It is worth noting that the power model does not set up a specific operational frequency but just an interval: this might not be desirable in time critical scenarios where timing variability arising from dynamic frequency regulation is discouraged.

It is still possible to configure a fixed frequency by using another command line tool: `jetson_clocks`. This tool allows to immediately overwrite the frequency of operation of all components to the maximum. Of course, this choice, while removing unwanted execution time variability, has clear implications on the power consumption profile.

In the following we report the results from an empirical assessment of the impact of power models and frequency on the execution (in isolation) of selected benchmarks. We consider an implementation of a matrix multiplication function, a common building block for many data-intensive functionalities, as those deployed in AI-based functions for image recognition, as an example. We execute the benchmark in isolation on the different cores and clusters in the AGX Orin under the different power modes. We measure cycles, not time, which makes observation agnostic on the CPU frequency. In fact, we are interested in understanding how each core may provide different execution conditions.

Figure 8 - Execution cycles in isolation per core under different power models.

As reported in Figure 8, the Cortex A78 cores show a similar behavior across cores and across power models. While the number of active cores varies across power models, the provided behavior in terms of execution cycles is homogeneous, as expected. It should be noted that the results are filtering the impact of the OS as observations were captured by configuring the PMULib to retrieve user-level hardware events only.

In terms of execution time, the impact of the power model is instead evident as shown in Figure 9 below, the execution time is correlated to the maximum frequency defined for each model. In the experiments, variability coming from dynamic frequency is removed by forcing the Orin to stick always to the maximum frequency.

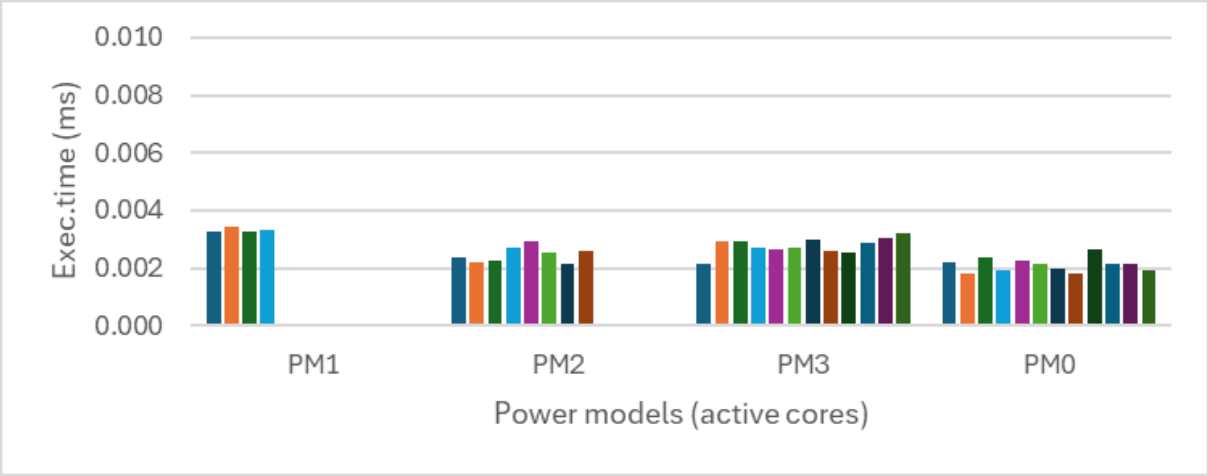

Figure 9 - Power models impact on execution time.

Results confirm that execution frequency is heavily affecting performance. The execution time under PM0 is much lower than that required under PM0, and also lower than that required under

PM3, due to the lower frequency bound for PM3. Instead, frequency upper bound for PM2 is closer to that obtained under PM0, as confirmed by the results.

It is worth noting that better performance comes at the cost of a higher energy profile. It may be worth considering further trade-offs between performance and power by defining custom power models that meet the timing requirements but with the minimum energy profile.

3.2.1.2 Review of platform config options

SAFEXPLAIN platform aims to support the FUSA and performance requirements emerging from WP2 and WP3 work. The concept itself of FUSA (software) architecture builds on the assumption that the underlying layer (hardware and software) can be configured to guarantee variable degrees of segregation among components. Segregation in the first stance can be provided by exploiting architectural features that need to be enabled and properly configured.

Table 2 below summarizes the main hardware level features and configurations that have been considered to promote segregation and mitigate interference at hardware and system-software level. For each feature, we identify what Layer in the execution platform is directly affected, the corresponding module (if HW), the main affected dimension (performance, interference, predictability, FUSA), the scope at which the feature operates, the available options and a high-level assessment of the impact expected when the configuration is enabled.

Table 2 - Summary of relevant deployment configurations.

Layer	Module	Dimension	Scope	Options	Impact
HW	Power module	Performance	System-level	Power mode	Largely affects performance and energy consumption. Fixing the power mode avoids inconsistent timing behavior.
HW	DL1	Interference	Intra-core	Partitioning	Reduce interference between applications running on same core
HW	L2	Interference	Intra-core	Partitioning	Reduce interference between applications running on same core
HW	L3	Interference	Intra-cluster	Partitioning	Not enough information available on AGX Orin. Would possibly reduce interference between applications running on same cluster
HW	L4	Interference	System-level	Partitioning	Unsupported or not enough information available on AGX Orin. Would possibly limit interference on L4 cache accesses. Expect limited interference anyway as most workloads should fit in L3.
HW	SFC	Interference	System-level	-	Interference to be captured with interference characterization and control strategy. Expect limited interference as most workloads should fit in L3.
HW	RAM	Interference	System-level	-	Interference to be captured with interference characterization and control strategy. Expect limited

					interference as most workloads should fit in L3.
HW	GPU	Interference	System-level	-	AGX Orin current JetPack does not support GPU partitioning. Potential interference to be captured with interference characterization and control strategy. Expect limited interference as GPU use in use cases is serialized (also according to case study questionnaire).
HW	PVA	Interference	System-level	-	Not used according to case study questionnaire.
HW	DLA	Interference	System-level	-	Not used according to case study questionnaire.
HW	CPU	FUSA	Core	Lockstep mode	Adds lockstep functionality to CCPLEX cluster. No particular implication on analyses.
HW/ SW	SPE	Predictability	System-level	Free-RTOS on SPE	Not considered in SAFEXPLAIN.
SW	-	Predictability	System-level	RT patch	Can be configured to reduce variability suffered by critical tasks. Requires kernel patch.
SW	-	Interference	System-level	CPU Mapping	Strategic application to ore mapping allows exploiting inherent architectural segregation, for example, among CCPLEX clusters.
SW	-	FUSA Predictability	System-level	Hypervisor	Unsupported on AGX Orin. Could allow segregation and isolation among SW partitions.

The features in Table 2 summarize the spectrum of supported mechanisms in the AGX Orin platform. The coverage these features can provide for WP2/3 requirements is not necessarily exhaustive and, as a matter of fact, it is not in the Orin case. Therefore, starting from this set of features, WP4 goal consisted in facilitating a mapping between WP2 and WP3 requirements into concrete features and configurations (Figure 10).

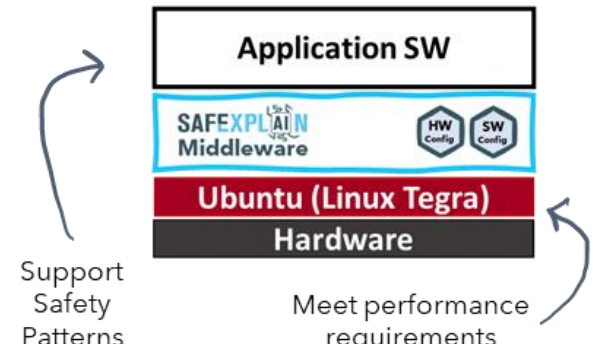

Figure 10 - Requirements steering platform configuration.

The main focus of WP2 requirements [2] is on segregation, to mitigate the impact of interference, and predictability. Obtaining full segregation on the AGX Orin is not possible without renouncing completely to performance. However, several degrees of isolation can be achieved, contributing effective means to mitigate the potential impact of interference. It is worth noting that the architectural level clusterization, with the 3 CPU clusters, already provides a good degree of

inter-cluster segregation: this means that the impact a software component executing on a cluster can incur on other components executing on other clusters is somehow limited by the resource sharing level. In particular, clusters benefit from cluster-private L3 cache. Intra-cluster interference, on the other hand, can be mitigated by enabling partitioning on the same L3 level (L1 and L2 are private per core).

On the performance side, the main driving elements are the power model, the execution frequency, and, clearly, the load on computing elements. With respect to the power model, the selection of the most appropriate configuration depends on the overall system design. As the power model defines what computing elements are enabled, it is important to consider the computational requirements of the use case: how many SW components use CPU, GPU, other accelerators and dependencies among them. In fact, the AGX Orin does not allow the sharing of the GPU among applications in parallel (i.e., no Multi-Process Service – MPS support[4]) and therefore all applications using the GPU will be necessarily serialized. As part of our work in analyzing the benefits of GPU partitioning, we analyzed the Multi-Instance GPU (MIG) feature provided by NVIDIA. While this feature offers in theory many benefits for execution time determinism, it is paradoxically only implemented for high-end GPUs used in data centers. The result of our analysis in fact shows that MIG offers important benefits on execution time determinism, while the impact of performance can be controlled via its flexible configuration options. Hence, this is an interesting feature to add to future embedded GPUs like that in the Orin architecture [7].

Execution frequency clearly impacts the time it takes for a function to execute. While power models define ranges of frequency for each mode, the actual frequency at operation for the different devices depends on the dynamic regulation mechanism. This is not well perceived in safety-critical systems where variability in execution time is negatively affecting timing predictability. The AGX Orin supports the overriding of the frequency setting and makes it possible to force the components to execute at the max frequency in the range. All experimental results and analysis in WP4 were performed under the assumption that the system operates at a fixed frequency for all scenarios and use cases.

Finally, the load on computing elements determines the response time of each software functionality (either a single function or a full functional chain). It is not uncommon, in time critical systems, that functionalities are executed uninterrupted, following a *run-to-completion* semantics. This is even more true for functional chains that are deployed to process external inputs (e.g., from sensors) and generate an action in response, through a sequence of elaboration steps. It is suggested to promote cohesive assignation of functions to computing elements so that the execution model can be also leveraged to avoid interference.

While mapping considerations are generally done on an application basis, the SAFEXPLAIN architectural patterns permit the definition of generic *mapping guidelines and patterns*, where software components are assigned to computing elements (i.e. clusters) based on cohesiveness and criticality.

[4] See https://forums.developer.nvidia.com/t/mps-on-agx-orin/219157

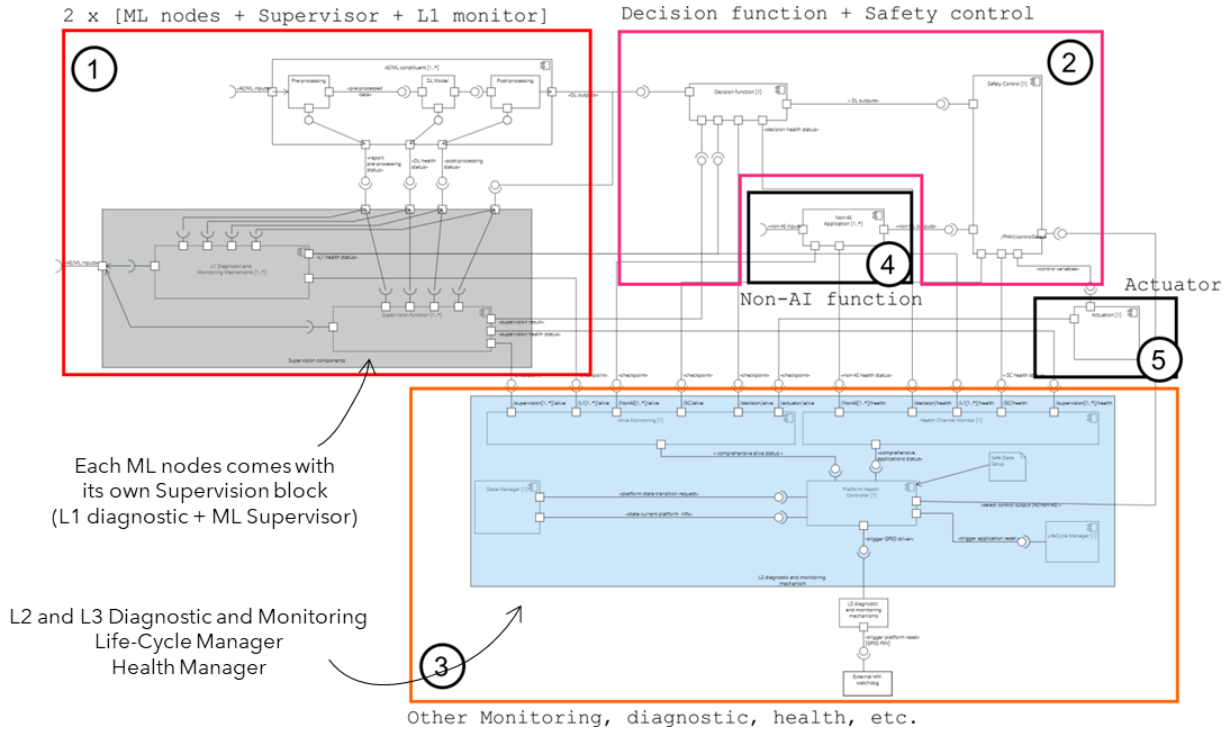

Figure 11 - Example of architectural pattern and mapping.

As an example, Figure 11 presents the software components' view of a WP2 Safety Pattern. Colored frames hint at possible mapping scopes where a mapping scope defines a set of elements that can be mapped to the same architectural cluster due to semantical or criticality affinities. In the example,

- Scope 1 comprises the AI component together with the Supervision block (taking care of both generic and ML specific monitoring and diagnostic tasks).
- Scope 2 groups the decision function and safety control, as they share the maximum criticality level.
- Scope 3 includes L2 and L3 diagnostic and monitoring services. Non-AI function and Actuators share two independent Scopes 4 and 5, but they can be eventually merged into a single scope.

By conveniently mapping Scopes to core clusters, it is already possible to limit the interference each scope may incur on each other, taking advantage of architectural clusterization. Ad hoc configurations are eventually required to meet performance requirements within the same cluster. This is generally obtained by finer grain control over execution (processes, scheduling algorithm, priorities, etc.).

3.2.2 Mapping of software components

Complex AI-based applications typically consist of several software modules that cooperate to manipulate external inputs and to produce an action. The type of systems targeted in SAFEXPLAIN are not an exception to that. It is also the case that these systems rely on complex heterogeneous platforms to provide the necessary computational power to deliver the intended functionalities in a timely manner. In the presence of multiple and possibly heterogeneous computing elements, the way each software component is concretely executed on the platform can make the difference between adequate and poor performance.

In the AGX Orin platform an application can be deployed on three different clusters and, depending on the specific AI problem at hand, can be assigned to a different accelerator within GPU, PVE, DLA. Ad hoc deployment schemes may be adopted depending on the number of components, dependences, as well as performance and segregation requirements.

Mapping and timing interference. We conducted an empirical study to assess how cluster-level deployment options may impact on execution time, with special focus on exposure to timing interference. To this extent, we have collected a large set of results executing the same matrix multiplication function used to assess power mode impact in Section 3.2.1.1, in multicore scenarios to observe the impact of timing interference. We executed an extensive set of experiments with different representative benchmarks characterized by a different degree of use of shared memory resources. In the following we report the results for a representative scenario, where the matrix multiplication function is executed in parallel multiple instances of a synthetic benchmark with *large amount of L2 misses, but hitting the L3*. Since the L3 cache is shared within the same cluster, the benchmark can be used to explore the impact of cluster sharing.

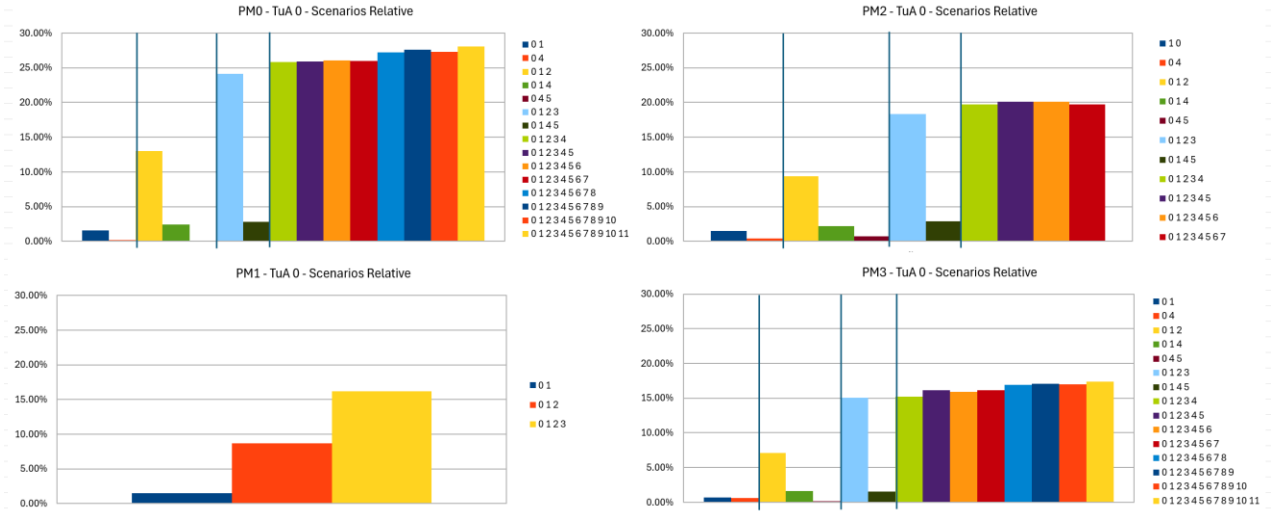

Figure 12 - Impact of core mapping in timing interference.

Figure 12 show the impact of the combination of mapping and power models on the execution time of the matrix multiplication function when run against a variable number of copies of the synthetic benchmark (generating L2miss/L3hits). The number of explored scenarios changes with the power model as the latter determines the number of active clusters and cores in the platform. As expected, when the benchmark is executed in the same cluster, it incurs larger timing interference due to the contention arising on the L3 cache (including potentially additional evictions caused by the contenders). In our experiments we focused on the changes affecting the execution cycles to isolate the impact of interference: results are relative to the execution in isolation when the matrix multiplication function is executed in core 0.

Under Power model 1 (PM0) only one Cortex A78 core cluster is enabled, which provides a nice power profile ~15W. In this scenario, contention mainly arises when requests access the cluster shared L3 cache, as L1 and L2 ones are core-private. We observe the impact of contention within the same cluster. Contention varies depending on the characteristics of the running applications. However, we notice how L3 is indeed a source of contention, and the impact is dependent on the number of contenders sharing the cache (max observed impact at ~16%), following an apparently linear relation. PM1,2,3 allows for a larger number of deployment scenarios where the system level decisions on task (node) to core mapping can make the difference in limiting the potential impact of contention. The amount of interference across PMs changes consistently with the

number of clusters enabled. In general, under all PMs, it is noted that the impact of contenders running on other clusters is negligible, which is explained by the characteristics of the task under analysis (high L3 hit ratio) and by the role of Orin architectural clusterization in supporting naturally segregation. It is interesting to observe the impact of contention in PM3: despite it shares the same set of enabled clusters PM0 the impact of contention is relatively lower, which seems to be related to the impact of max frequency on how frequently contending requests actually clash in accessing the L3. This conclusion is consolidated by the observation that PM2 (which configures middle way frequency of execution) compared to 2-cluster scenarios in PM1 and PM3 provides middle way results.

Results obtained with other benchmarks confirmed the trend. In general, the obtained results support our expectations on the importance of architectural level clusterization in the Orin to shape and limit the potential amount of timing interference suffered at run time. In turn, this conclusion motivates the need for an informed approach for fine-grained mapping of software components to the platform.

Mapping, Safety Patterns, and Middleware. The guiding principles for mapping the software components on the platform are not limited to interference reduction but also include minimum performance, predictability, and indirect requirements stemming from (software) architectural constraints.

In the SAFEXPLAIN approach, the Safety Patterns [2] bring a set of requirements on the set of mandatory elements to be deployed on the platform (for example to provide Diagnostic and Monitoring support) and on the interactions among them. These requirements translate into constraints on the feasible mapping scenarios. Furthermore, the SAFEXPLAIN Middleware [1], by introducing an additional layer, is also impacting the way software modules are deployed on top of the platform. Figure 13 shows the multiple layers involved in the mapping of software components to the execution platform: The Middleware layer, embedding a ROS2 [8] layer, and a Linux layer.

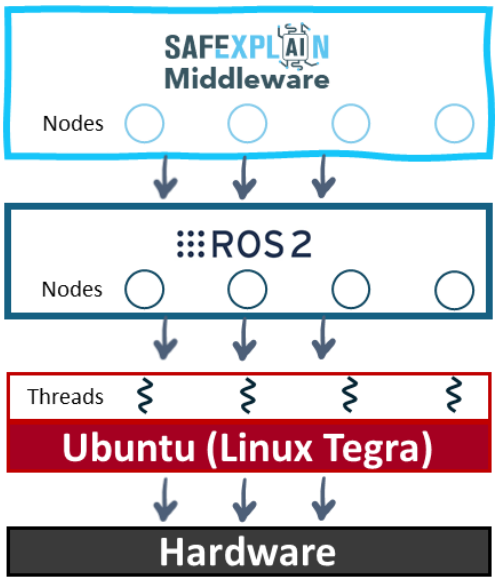

Figure 13 - Multilayered mapping from Middleware components to Linux threads.

At the middleware level, the mapping problem reduces to the problem of finding a mapping of application functionalities onto safety pattern elements and middleware nodes (abstraction and

extension of ROS2 nodes). As a preliminary step, at application level, software components need to be framed within the FUSA architecture and specifically on the Safety Pattern that better fits the role of the ML component in the system and the FUSA implications. As anticipated in Section 3.2.1.2, the mapping is partially determined by the relation between the mandatory building blocks in the Safety Patterns (e.g., Supervision Function, Decision Function, etc.). The architectural pattern can be extended/replicated to model larger systems with multiple software components, still abiding by the implicit and explicit rules dictated by the specific safety pattern. After middleware mapping is decided, we identify 2 further layers in the mapping problem.

At the ROS2 level, the mapping problem focuses on mapping (middleware) nodes to ROS2 run-time entities. ROS2 exploits the concept of *executors* for execution management to identify the run-time entities that are responsible for executing the callbacks, timers, and servers of a node. Three types of executors are supported: *Multi-Threaded Executor*, *Single-Threaded*, and *Static Single-Threaded*. The multi-threaded executor deploys a configurable number of OS threads to enable parallelism in processing messages and events. The single-threaded executor, instead, deploys one single thread so that the processing of all messages and events is serialized. The static single-threaded is a variant used when services and callbacks are fully statically defined.

Executors can be associated to (multiple) callbacks explicitly and, more importantly, can be shared across nodes. For a fine-grained control of nodes to core mapping it is therefore necessary to control explicitly the set of executors for each node in a consistent way with cohesive software functions and intended segregation objectives. In this respect, some blocks are meant to be deployed in the same core or, at least, same cluster in reason of the cohesive nature of the provided set of functionalities. This is the case, for example, of the L1DM and Supervision components concurring in the Supervisor Block. On the other hand, some other blocks need to be kept apart due to mixed-criticality concerns or to keep the controller and controlled blocks sufficiently segregated.

At (Linux) Thread level, executors are associated to one or multiple threads in the OS (in this case Linux) layer. Ultimately, it is at thread level where mapping and execution order of functions (through priorities) are determined. It is therefore important that the last layer before the actual thread execution is configured consistently with the middleware and ROS2 mapping. At this stage, it is possible to define a precise mapping of executors to threads and a mapping of threads to specific cores in the platform. The mapping must be consistent with the software architectural constraints (e.g. need of parallelism in processing messages/events) and the platform configuration (e.g. power model).

We will discuss, in Section 5.3, how this fine-grained control is instrumental for supporting specific technology for timing interference control and monitoring.

3.2.3 Timing interference control

Timing interference is a deeply studied problem in embedded critical systems as the incurred variability makes it more difficult to provide tight and trustworthy bounds on timing requirements of time-critical applications. As is well known in the state of the art, when it comes to real platforms, sources of interference cannot be completely removed. First, even if there are partitioning approaches for hardware shared resources, aggressive segregation might require renouncing completely to performance. Second, and more importantly, modern platforms lack support to prevent contention in hardware shared resources, which are growing in number and complexity in every new platform generation [9].

With respect to the SAFEXPLAIN setup, at architectural level, multicore timing interference in the Orin mainly arises from sharing the cache and memory hierarchy and the interconnect. The architectural clusterization in the platform allows to easily achieve some degree of isolation from interference already by design through an informed mapping of software elements to clusters and cores within same clusters. Multicore execution results provided in Section 3.2.2 confirm that, despite the aggressiveness of the contenders, private per-core L1 and L2 caches and the clusterization make that the observed impact of interference is not dramatic in the Orin. This is also the case for compute-intensive applications like those addressed in SAFEXPLAIN characterized by a high degree of reuse in the data path.

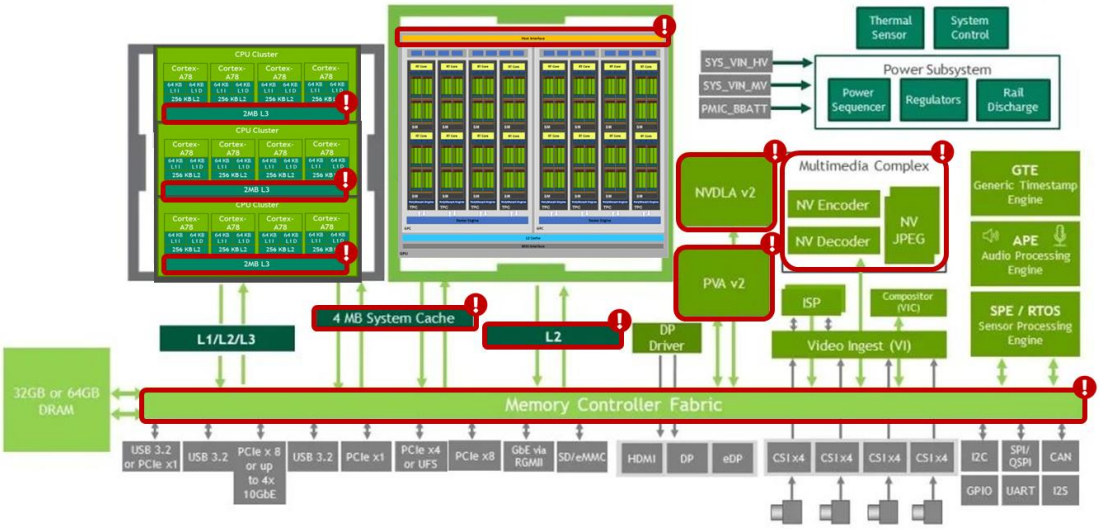

Figure 14 - Potential sources of timing interference in the Orin.

In line with the empirical evidence discussed in Section 3.2.2 , the potential timing interference impact one application can suffer can be heavily reduced by selecting and enforcing an appropriate mapping of software components and underlying threads to platform islands defined by clusters and cores. Interference arising on L3 accesses affects only those software components that share the same cluster. Arguably, a mapping strategy should be collapsing to the same cluster those applications that are not executed in parallel (e.g. because they are separate steps in a functional pipeline). Whenever the mapping strategy does not allow to exploit architectural clusterization then it is possible to resort to other mechanisms to mitigate the impact of interference.

First, at hardware architectural level, hardware and software cache partitioning solutions could be deployed to ensure each core in the cluster obtains its own share of the shared (L3) cache. As reported in Table 2, the Orin is meant to support hardware-level partitioning of the L3 cache. However, the way the cache should be configured is not adequately documented, as often happens for embedded targets and as it happens for L4 in the Orin. Despite the reverse engineering efforts, it was not possible to obtain a working configuration. It seems that the partitioning scheme, while partially effective, will eventually keep some cache ways shared among cores in the same cluster, thus essentially defeating our purpose (as we are not interested in average case performance). On the other hand, software-level partitioning would have required the introduction of an additional complexity layer in the software stack.

The embraced solution has been suggested by most recent works in the state of the art [10] [11] [12] where approaches are increasingly building on the combination of hardware and software architectural features with some kind of monitoring or regulation mechanism to keep the residual interference under control. This is mainly motivated by the necessity to have an agnostic approach that does not excessively depend on the platform level support and can ensure interference

mitigation in the realistic scenarios where, despite full segregation mechanism exist, performance cannot be renounced.

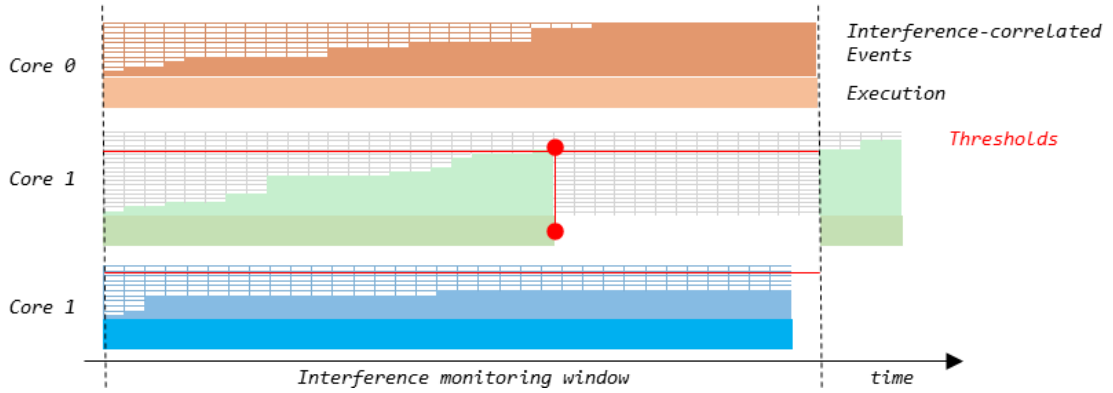

Figure 15 - High-level view of interference control scheme.

Several interference control schemes have been proposed [10], [13], [14]. The main concept they embrace is that interference can be monitored at run-time by tracking the activity on the shared hardware resources (source of interference) so that, when generated interference exceeds a predefined threshold, some applications are prevented from accessing the source of interference. Figure 15 illustrates how a generic interference control mechanism can affect the execution of the application on Core 1: the blue segment corresponding to the unrestricted execution and the green one showing how the exceedance of the threshold caused Core 1 to pause its execution until Core 0 is done and Core 1 execution can be resumed. These mechanisms typically build on two main aspects: monitoring and control/mitigation policy. The latter can vary depending on the scope and timing requirements.

In SAFEXPLAIN we build on a holistic approach for interference control where we combine:

(i) architectural level segregation offered by the Orin architecture, with (ii) an interference-aware software-level mapping strategy, and

(ii) a run-time interference monitoring and control mechanism (CGuard tool, described in Section 5.3).

All the above aspects depend on the specific application requirements.

At L4 level, the sharing involves all the CPU clusters and the GPU. Again, in the Orin no explicit support for partitioning the cache is documented [1]. Other larger and more powerful models of the same manufacturer are indeed providing better support for partitioning which hints at the possibility of having similar support in the target type of platform.

Regarding interference arising on the GPU side, when multiple software components are exploiting the GPU, the Orin is not supporting GPU time sharing scheme, which is instead supported in other platforms from NVIDIA. We do not consider this to be a huge issue in the operational scenarios we foresee for the type of applications we are considering in SAFEXPLAIN where some degree of serialization in the GPU is not detrimental to the overall performance. When the application can benefit from a high degree of parallelism in accelerators, it is possible to resort to specialized accelerators in the Orin (PVA, DLA) that are designed for specific problems and need porting to specialized API.

4 Observability Channels (T4.2)

In the third phase of the project, the main goal of this task consisted in the refinement of the bespoke observability library, PMULib, and its tailoring and integration with the project technologies. In particular, we focused on (i) refining the observability scope, to support different use cases for the library, and (ii) finalizing the integration of the library with the middleware framework and technologies.

In this section, we cover the design and validation of PMULib extended scope and discuss the main directions followed for the integration of PMULib within the SAFEXPLAIN middleware to support the analysis and run-time monitoring objectives.

4.1 Multiple instantiation support

A first important modification on the observability support has been focusing on relaxing some constraints on the use of PMULib in terms of the number of supported instances at the same time. PMULib has been improved to allow multiple instances of it being in use simultaneously. This allows different processes to configure and collect data for different events and scopes. To this end, the library keeps track internally of the current processes using it and maps each one to their own subset of data, keeping track of the file descriptors used to access the perf subsystem. To that end and to keep critical sections thread-safe the library protects system calls with `pthread_mutex_lock`. This is done internally and transparently for the user.

4.2 Specialization of PMULib scope

In the course of the project, it became evident that there were different use case scenarios for PMULib. These scenarios differentiate on the observability scope of interest: in some cases, we can be exclusively interested in collecting hardware events for a given process or thread, while, in some other cases, we can be more interested in all events happening on a given core. The way those scenarios map to SAFEXPLAIN technologies will be discussed in Section 4.3.

We extended PMULib functionalities to support multiple observability scopes by exposing to the user a means for selecting a specific PMULib mode. The PMULib provides some parameters that enable the configuration of different features in the library, allowing the user to fine-tune event collection to their needs. There are three dimensions that can be configured:

- Collect counters Per Process/Per thread
- Collect Own counters/System wide events
- Collect events occurring in Any cores/Individual core

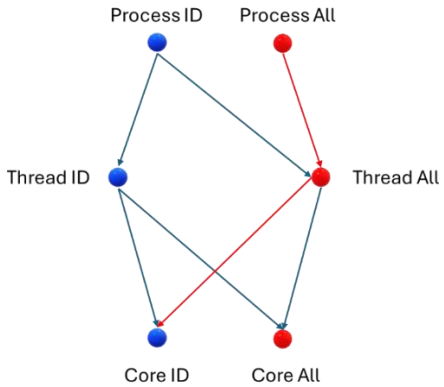

Figure 16 - PMULib configurable scopes.

Figure 16 illustrates the configurable operational scopes for the PMULib. Blue paths are representing feasible options for tracking process/thread level events, whereas red paths are relative to global events triggered by any process, which cannot be collected system wide but only for a specific core.

The configuration parameters are configured in groups of three:

```
/* Defines in a78ae-pmu.h */
#define PMU_MODE_PER_THREAD             0
#define PMU_MODE_PER_PROCESS            (1<<8)
#define PMU_MODE_SCOPE_OWN_COUNT 0
#define PMU_MODE_SCOPE_ALL_PROCESSES    (1<<9)
#define PMU_MODE_CORE_ANY        0
#define PMU_MODE_CORE_FIXED             (1<<10)

/* example configuration */
a78ae_pmu_configure(mask, events, PMU_MODE_PER_THREAD | PMU_MODE_SCOPE_OWN_COUNT|
PMU_MODE_CORE_ANY)
```

Some combinations are not supported, such as collecting system wide events occurring in any core, but most of them are. In this section we focus on the most useful configurations:

```
PMU_MODE_PER_THREAD, PMU_MODE_SCOPE_OWN_COUNT, PMU_MODE_CORE_ANY
PMU_MODE_PER_PROCESS, PMU_MODE_SCOPE_OWN_COUNT, PMU_MODE_CORE_ANY
PMU_MODE_PER_PROCESS, PMU_MODE_SCOPE_OWN_COUNT, PMU_MODE_CORE_FIXED, cpuid=X
PMU_MODE_PER_PROCESS, PMU_MODE_SCOPE_ALL_PROCESSES, PMU_MODE_CORE_FIXED, cpuid=X
```

In addition to the formal correctness validation of the PMULib carried out in the previous subsection, we also have conducted additional tests to verify that the configurable parameters behave as expected. This validation sits on the already validated PMULib, i.e. we assume in this section that we can trust the PMULib results and then go on to test the different configurations.

4.2.1 Experimental setup and validation

The experimental setup is comprised of a process that sets up the PMULib, each test with different configuration options, and then spawns two threads, waits for them to finish, and collects the PMULib results. The threads are the same for every test, although they are mapped to different cores depending on the test. Each thread executes a loop performing some memory operations as shown in the Figure 17 below:

```
void* threadFunction(void* arg) {
        const int SIZE = 1024*1024;
        unsigned char array[SIZE];
        int threadId = *(int*)arg;
        for(int i=1; i<SIZE; i++) {
                array[i] = array[i-1]+i*3;
        }
        return NULL;
}
```

Figure 17 - PMULib feature validation thread code snippet.

We spawn the threads using the pthreads library. We iterate 1000 times for each test and collect each iteration as a datapoint for the evaluation.

4.2.1.1 Test 1: PMU_MODE_PER_THREAD, PMU_MODE_SCOPE_OWN_COUNT, PMU_MODE_CORE_ANY

The main process spawns two threads but the PMULib is configured to read counter per thread. This means that the main process does not collect any events generated by the spawned threads. The outcome matches the expected result, where we see less than 3000 instructions retired (compared to the 25M executed by each thread).

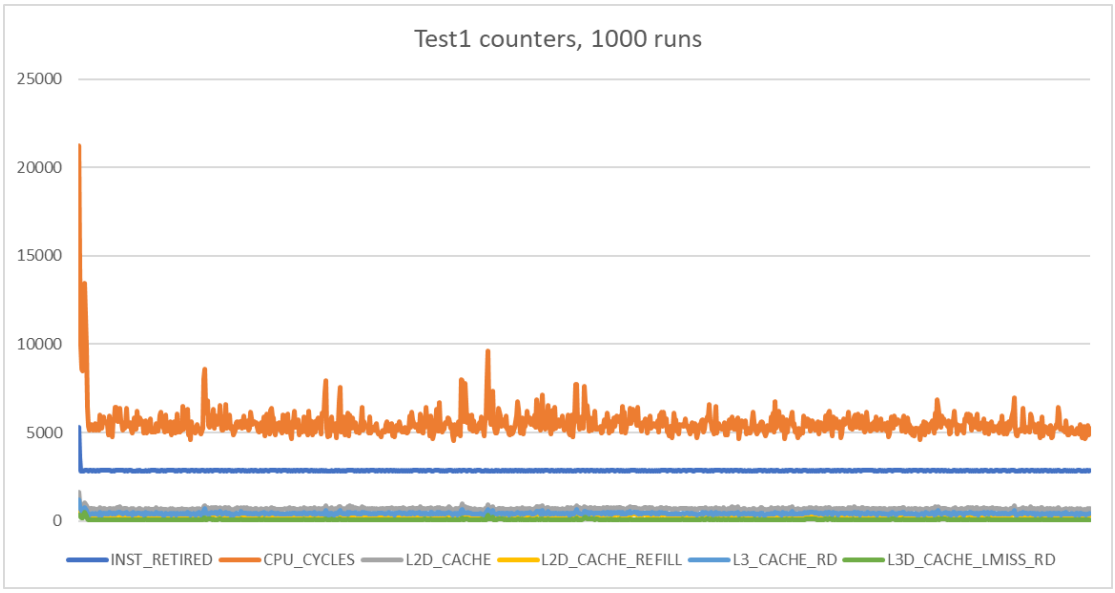

Figure 18 - Events collected with per-thread scope, own count, and any core.

The events observed, as reported in Figure 18, are very low and are related to the few instructions run by the main process to control the experiment and use the PMULib.

4.2.1.2 Test 2: PMU_MODE_PER_PROCESS, PMU_MODE_SCOPE_OWN_COUNT, PMU_MODE_CORE_ANY

In this test the main process spawns the threads after configuring `PMU_MODE_PER_PROCESS`, which causes the PMULib to also capture the events generated by the threads.

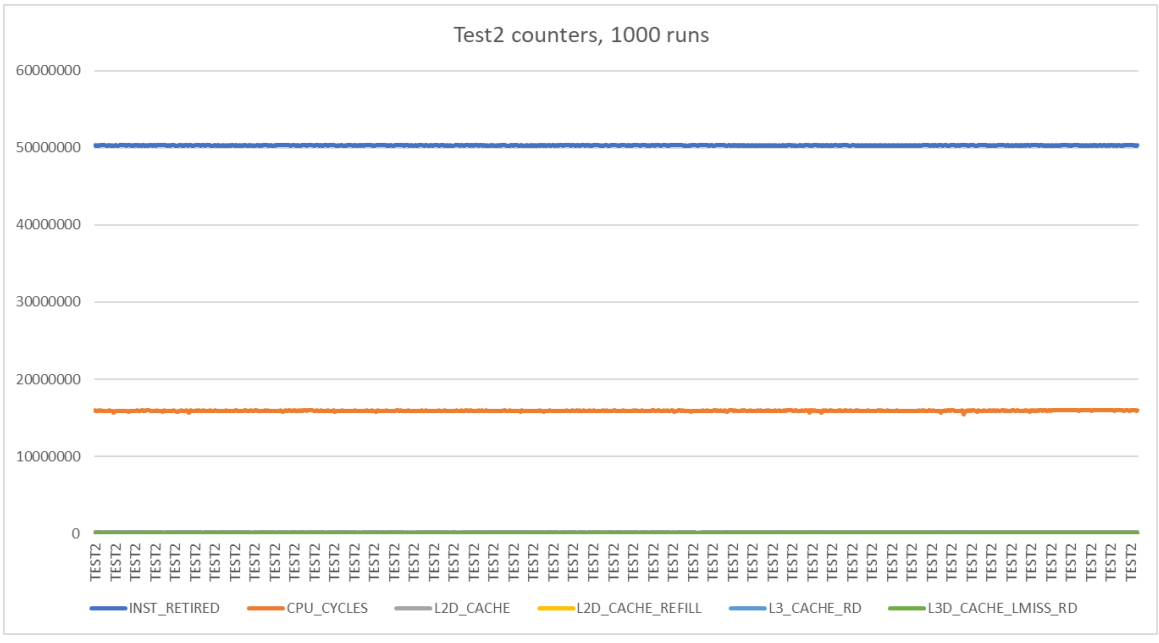

Figure 19 - Events collected with per-process scope, own count, and any core.

In Figure 19, we observe 50M instructions retired, which match the 25M per thread that we expected.

4.2.1.3 Test 3: PMU_MODE_PER_PROCESS, PMU_MODE_SCOPE_OWN_COUNT, PMU_MODE_CORE_FIXED, cpuid=4

We repeat the same experiment with a subtle change: in this test we force the mapping of the threads to cores 4 and 5 using the `pthread_setaffinity_np()` function. Then, we configure the PMULib to, instead of collecting all counts regardless of the core, collect only the results generated in core 4.

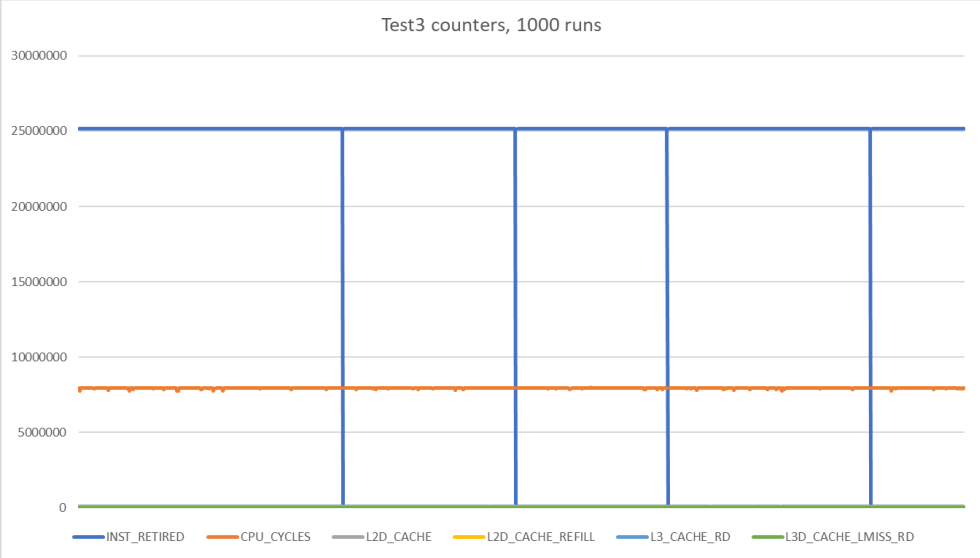

Figure 20 - Events collected with per-process scope, own count, and fixed core.

Once again, the results reported in Figure 20 match our expectations, as the number of instructions retired went down to 25M, the amount executed by a single thread.

4.2.1.4 Test 4: PMU_MODE_PER_PROCESS, PMU_MODE_SCOPE_ALL_PROCESSES, PMU_MODE_CORE_FIXED, cpuid=4

Test 4 is very similar to Test 3, but we collect the results for any process running on core 4, not just the one configuring the library. This is a subtle difference with respect to Test 3, as we are mapping each thread in that core, but the objective of this test is to ensure that we can configure `PMU_MODE_SCOPE_ALL_PROCESSES` and collect the results we expect. Note that this test requires root privileges, as it is reading other process data.

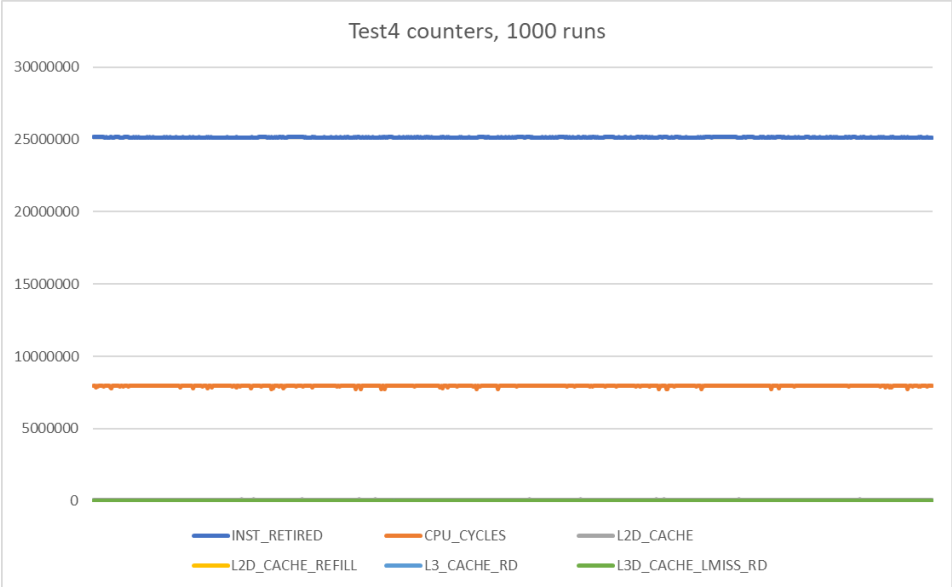

Figure 21 - Events collected with per-process scope, all processes, and fixed core.

As expected, Figure 21 reports the same results as in Test 3.

4.2.1.5 Test 5: PMU_MODE_PER_PROCESS, PMU_MODE_SCOPE_ALL_PROCESSES, PMU_MODE_CORE_FIXED, cpuid=11

This test complements Test 4, and its objective is to check if the PMULib reads low values when a core which is not running any thread is configured. We have set the same flags, so we are collecting system wide events (count for any process), and we require root access. We map our threads as in the rest of the tests to cores 4 and 5, and we collect PMULib results for core 11.

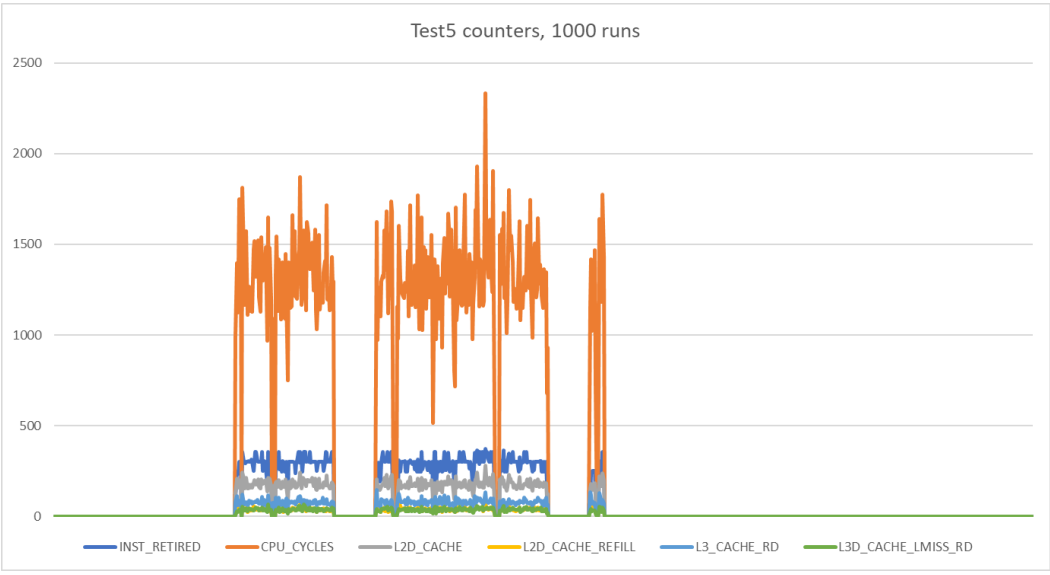

Figure 22 - Events collected with per-process scope, all processes, and fixed core (unused core).

In Figure 22, we can see that most of the time core 11 is idle, sometimes a sporadic task seems to be mapped there (likely an OS thread), but the values are extremely low (always less than 500 instructions retired).

4.2.2 Conclusion

We have tested the most useful PMULib configuration parameter combinations and we have proven that the outcome matches the expectations. In the figure below we see a summary of all tests, showing the CPU_CYCLES accounted in each of them. We see tests 1 and 5 counting very close to zero, as they are not counting any thread activity; we see test 2 counting activity for both threads; and we can see tests 3 and 4 counting the same activity, for a single thread.

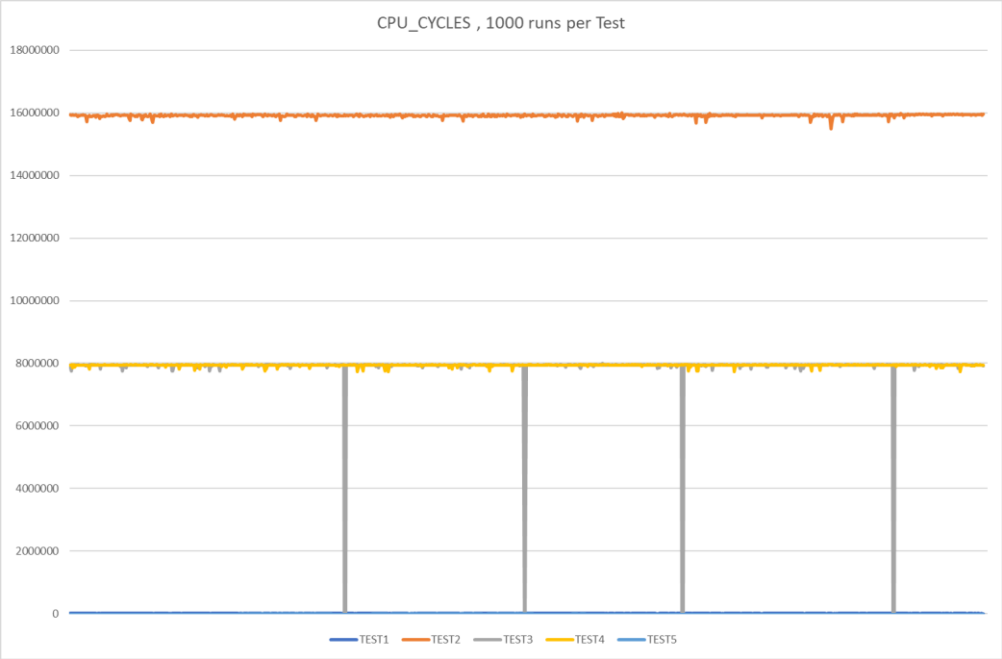

Figure 23 - Recap and overview of test results.

4.3 PMULib Integration

Additionally, in this phase of the project, we have focused on achieving the necessary degree of integration to support the case studies and to support the deployment of other technological elements in the SAFEXPLAIN ecosystem. First of all, the PMU Library has been closely integrated with the Middleware and in particular with the `BaseApplication` class (representing the baseline middleware component) to enable automatic monitoring of the component functionality. This integration has been designed to make the instrumentation transparent to the application designer and simply configurable (enabled/disabled) at deployment time. Furthermore, the integration also allows the user to explicitly and manually select which part of the application must be monitored and what hardware events are deemed relevant.

4.3.1 PMULib configuration for use in the middleware

Before using the library in the SAFEXPLAIN middleware, the user has to configure it in the machine that will run the application. The PMULib can be deployed as a shared library (`*.so` file), or using the source files directly. While implemented in C, the library also supports Python implementation of the middleware itself and applications. In this case, the user must set an environment variable (as exemplified below) so the library's Python wrapper will be able to use the PMULib.

```
export A78AE_PMU_LIBRARY=/path/to/the/lib.so
```

The monitoring library sends the values collected from the PMCs to a specific Middleware topic *a la* ROS2 (that the user can configure). The collected values can be read/accessed by the user using the `PMULogger`, a ROS2 compatible Middleware node, that receives the messages from the monitoring library, shows them in the console and stores them in a memory location. To set up the file's location, the user should modify the following environment variable:

```
export A78AE_PMU_OUTPUT=/path/to/the/output.txt
```

4.3.2 Enabling node-level monitoring

The Middleware-level integration provides the capability to monitor one node without setting additional code or pragmas but just by setting a configuration value. The library is included in all instances of the `BaseApplication` class (i.e. all nodes in the Middleware and Application), but the monitoring is disabled by default. To enable it, the user needs to create or modify the existing node's configuration file to set the following parameters:

- `enable_performance_monitoring`: enables or disables the node's monitoring.
- `performance_monitoring_mode`: selects which monitoring mode to use. We currently support three modes:
    0. *Timing*: collects basic timing metrics (CPU cycles).
    1. *Contention*: collects contention metrics
    2. *Manual*: collects custom set of metrics.

Mode 0: is the default mode, collecting instructions executed and cycle counts, which are the baseline for measurement-base timing analysis (See Section 5).

Mode 1: is the specific mode used to intercept and model contention impact. The tracked events are those hardware events with high correlation with interference. See Section 5.3 for the definition of the concept of hardware event correlation and how this is exploited for contention modelling and prediction.

Mode 2: is the custom model, fully configurable by the user. The specific set of tracked hardware events can be modified to allow the collection of all supported events in the PMULib. The user is responsible for selecting which area of the code wants to monitor using the libraries directives, already explained in [1].

Below we report an illustrative example of PMULib configuration for enabling the monitoring of a middleware node. The example shows how the *dl_02* model node from the *smw_ml_constituent* is configured to enable the timing monitoring. The file providing the configuration is a simple `yaml` file.

```
/**:
    dl_02:
        ros__parameters:
        reference_cycle_ms: 1000
        enable_alive_monitoring: true
        enable_health_monitoring: true
        enable_performance_monitoring: true
        performance_monitoring_mode: 0
        model_file: "path/to/the/model "
```

The most relevant entries in the configuration snippet are those setting the PMULib mode to 0 and setting to true the monitoring flag. Other entries in the configuration, which will be exhaustively covered in the middleware user manual, are instead related to configure basic execution features (i.e. execution frequency), platform level features (i.e. health and alive monitoring), or parametric component (i.e. ML models path).

4.3.3 Manual monitoring

The manual monitoring allows the user to monitor a particular piece of code of the application and not the full functionality provided by the node (as in the node-level monitoring). This mode is normally paired with PMULib mode 2. To enable this PMULib usage scenario, the user has to enable performance monitoring and set the mode to 2 (*manual*). Then, in the code, the user has to set which parts of the code to monitor. Next, there is an example of how to use the library to set the starting and end points of the monitoring:

```
def run(self) -> bool:
    # ...
    if self.pmu:
       self.pmu.reset_and_start()
  # ...
    if self.pmu:
        read = self.pmu.stop_and_read()
        self.pmuPublish(read)
    # ...
```

The `self.pmuPublish(read)` directive is used to feed the PMU Logger (see below), but it is not mandatory if the user does not plan to use it as the output can be processed similarly to any ROS2 topic.

4.3.4 PMU Logger

The PMU Logger is the node that receives the messages from the PMU library with the values collected for the tracked events. Its main functionality is to print the collected values to the terminal (e.g., the number of cycles each `run()` instance takes from start to end), and, by default, stores the values in the file located in `$A78AE_PMU_OUTPUT`. There is a `PMULogger` prototype

already configured for the `smw_ml_constituent`. The user can execute it by launching it as a ROS2 node, running the following command:

```
ros2 launch smw_util pmu_logger.launch.py
```

The topic to which the logger is subscribed and the logger's output path can be changed through parameters (`pmu_topic` and `output_path`, respectively) within the launch. The monitoring of multiple applications simultaneously is also supported, so multiple loggers reading and writing from and to different sources can be deployed.

4.3.5 PMULib and CGuard for contention control

PMULib is also at the art of CGuard, the contention monitoring and control solutions designed and deployed in SAFEXPLAIN. While the mechanism design and principles are discussed in Section 5.3, in the following we discuss the requirements imposed on PMULib and how the PMULib integration meets them.

As discussed in Section 5.3, CGuard mechanism builds on monitoring a certain set of hardware events that are strongly correlated to multicore timing interference and use these events to model the impact of non-critical functions on the timing behavior of critical ones. The modelling is exploiting the correlation between events and the potential delay suffered by the critical task. A linear formula for modelling such impact can be derived from empirical observations and applying the methodology described in Section 5.2.

In this context PMULib is used both at analysis time and run-time. At analysis time, PMULib is used to collect a wide set of events on specific multicore execution scenarios where the critical task is executed in parallel with a set of contenders to assess its sensitivity to contention and to derive the linear contention formula by exploiting the observed correlation between events and contention impact. The target operational scenario for PMULib in this case is thread level and does not require tailoring PMULib integration as it is easily achieved by configuring the PMULib accordingly (we can assume thread level events as we are controlling the mapping of threads to core in the middleware configuration).

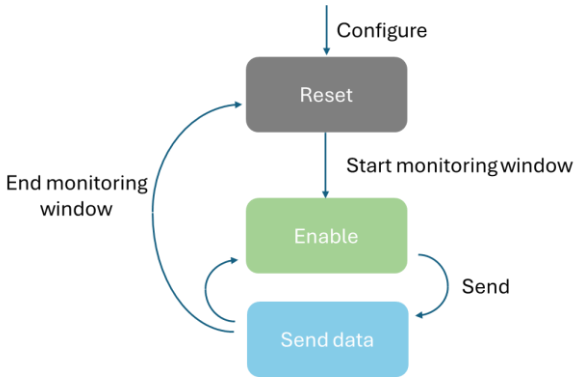

Figure 24 - PMULib state transitions in CGuard.

At run-time, PMULib is used to monitor the non-critical tasks by collecting the selection of hardware events (correlated to contention impact) and sending them to the CGuard module, which is responsible for processing the information and reacting in case the predefined threshold for the predicted contention is exceeded (see Section 5.3). In this case, the PMULib is pre-configured within the non-critical application class in the middleware, extending the base application class with specific functionalities to support the CGuard logic. The PMULib is involved

in the gathering and sharing of the set of events happening on the middleware node executing the non-critical task. In this case, the selected scope is the Core-level one as we enforce by construction that if one core is executing a non-critical node then it can only be executing non-critical nodes. This is justified by mixed-criticality considerations and to guarantee minimal separation between critical and non-critical functionalities. In this case, we are therefore interested in capturing all the impact coming from the core executing non-critical nodes in the contention monitoring interval. The PMULib is pre-configured accordingly, taking care of the set of events that need to be tracked and supporting CGuard logic on when to enable, disable and reset hardware event counters. The PMULib state transitions within the scope of CGuard is provided in Figure 24.

5 Timing Prediction Methods and Tools (T4.3)

The overall approach for timing validation was identified in the previous phase of the project and consisted in the deployment of a probabilistic timing analysis based on Markov's inequalities that resulted to be well equipped to deal with the complexity of the heterogenous MPSoC target and the type of execution time distribution exhibited by AI-based complex functionalities. We also identified the need for run-time monitoring of some non-functional aspects, with particular interest in the impact of multicore timing interference.

During the third phase of the project, this task has mostly focused on the integration of the timing analysis solutions and on supporting the deployment of a comprehensive, consistent strategy for timing verification and validation. In particular, the efforts have been devoted to three main objectives:

1. tailoring of the timing analysis approach to better capture the inherent traits of the type of systems we are addressing in SAFEXPLAIN;
2. complementing the timing verification strategy with implementation and integration of CGuard, a contention monitoring and control mechanism; and
3. improving on the automation of the timing characterization approach with the middleware environment, to allow a streamlined application of the RestK [1] timing analysis tool.

5.1 Detecting Low-Density Mixture Component Distributions in High-Quantile Tail

The execution of complex AI-based functionalities on top of cutting-edge heterogeneous MPSoC devices leads to highly variable execution times which are complex to analyse and to exploit for resource allocation and optimization. Those highly variable execution times are challenging the application of consolidated timing analysis approaches [15] and have justified an increased interest in probabilistic timing analysis approaches. Within the spectrum of probabilistic timing analysis approaches, the complexity of the analysed distribution may affect and impair the obtained results.

One challenge we addressed in this task is the presence of low-density mixtures in the tail of the execution time samples and their potential impact on probabilistic WCET (Worst-Case Execution Time) estimation. These mixtures arise from intricate interactions between cutting-edge hardware and software, leading to complex execution time distributions. Hardware-related variability patterns (due to uncountable stateful resources like multi-level caches, interconnects, on-core resources, ...) and coexistence of multiple operational scenarios contribute to this variability.

To conduct a detailed analysis of this problem, we studied several parametric mixture models, proving that WCET estimation can be misleading if the mixtures are not properly detected, exemplified by a Gaussian mixture with one component having $w_1 = 0.99$ of the weight with mean $\mu_1 = 10$ and standard deviation $\sigma_1 = 1$ and the second component having weight $w_2 = 0.01$, mean $\mu_2 = 20$ and standard deviation $\sigma_2 = 3$.

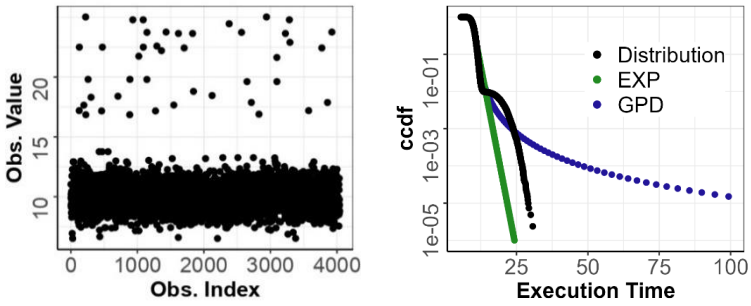

Figure 25 - Example mixture distribution and EXP, GPD results.

As exemplified in Figure 25, in the case of the GPD (General Pareto Distribution), it produces heavy tails, as the threshold selection detects that the tail begins before the second component of the mixture, which produces a heavy tail because the rest of the extreme points are further apart, leading to pessimistic upper-bounds. In the case of the EXP, the threshold has been selected before the second component, so the pWCET (probabilistic Worst-Case Execution Time) estimate is optimistic.

As addressing the presence of mixture distributions in high-quantiles does not fall within the specific scope of any existing IID (Independence and Identical Distribution) test in the state of the art, we developed an automatic algorithm for testing ID (Identical Distribution) on the tail.

This algorithm, named *TailID*, is based on the confidence interval (CI) computation for the EVT (Extreme Value Theory) EVI (Extreme Value Index) parameter ξ. Assuming that the excesses of the execution time observations for a given threshold follow a GPD, the Maximum Likelihood method can be conducted to find an estimator for the parameter ξ. To assess the uncertainty around this estimation, the Central Limit Theorem shows that the difference between the true value ξ and the parameter's estimator $\hat{\xi}$ converges in distribution to a Gaussian, this is $\sqrt{n}(\hat{\xi} - \xi) \rightarrow^{d} N(0, \xi^2)$. The CI can be derived from this result.

TailID algorithm analyses iteratively the extreme values that may be triggering a change in the EVI, called candidate points. To achieve this, the algorithm first computes the CI for the EVI parameter without any candidate points, then reintroduces the first candidate and recalculates. If the recomputed EVI falls outside the CI, the candidate point (along with all more extreme candidates) is flagged as an inconsistent point, violating the ID assumption. If the recomputed EVI remains within the CI, the process continues with the next candidate until all candidates are examined or one is detected as inconsistent.

Given the number of Inconsistent Points Detected (IPD) we defined 3 scenarios depending on IPD's relation to the minimum number of samples (MoS) required for a proper EVI estimation.

1. Scenario 1. $IPD = 0$ *TailID* outcome provides more evidence on the robustness of the tail estimations given the stability of the tail.
2. Scenario 2. $IPD > MoS$ We are in the presence of multiple tail behaviours, then for performing pWCET estimations we consider the threshold of the tail at the first inconsistent value found with *TailID*, which would be on the last mixture component.
3. Scenario 3. $IPD \leq MoS$ More samples are needed to make a better estimation. If after performing more runs $IPD > MoS$ we are in Scenario 2, while if one reaches the

maximum number of runs that can be performed and still $IPD \leq MoS$, tail prediction should not be done, otherwise the uncertainty in the estimation can be high.

TailID has been evaluated not only on synthetic mixtures but also on real data collected from an AI-based application running on an NVIDIA AGX Orin, a reference embedded heterogeneous platform for AI applications in the automotive domain. Figure 26 below illustrates an example of *TailID*'s performance on a specific workload.

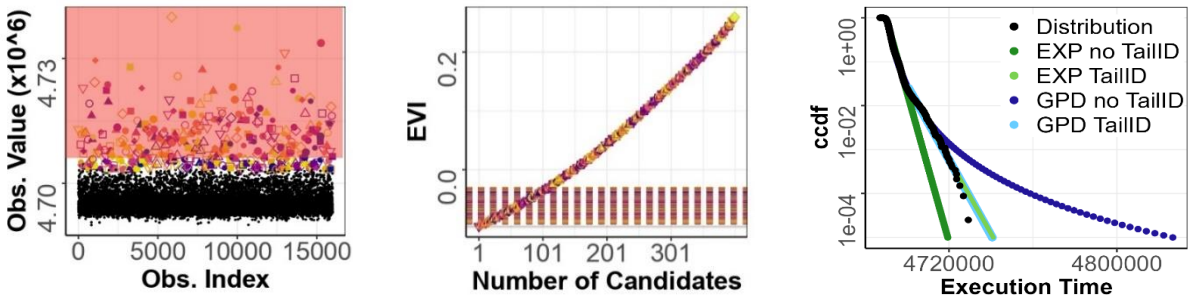

Figure 26 - TailID improvement over EXP and GPD.

The leftmost diagram shows the sample where points targeted as candidates by the algorithm are marked with distinct shapes and colours and the red-shaded area highlights the points ultimately detected as inconsistent. As many inconsistent points are detected, we fit Scenario 2.

In the plot in the middle of Figure 26, the horizontal grey line represents the GPD EVI of the sample's extreme set without including the candidate points, while the dashed line indicates the upper bound of its CI. Each point corresponds to the GPD EVI when the candidate point (represented with the same colour in the first figure) is added to the extreme set, with each dashed line representing the upper bound of the CI. *TailID* iterates this process until a candidate point is found to be above the CI, which will be used as the new threshold from where to estimate the pWCET.

Finally, the pWCET estimates are shown in the rightmost plot, confirming that detecting a low-density mixture in the tail and, hence, choosing the appropriate threshold, produces a tighter upper-bound with the GPD and EXP models.

5.2 Contention Modelling with Linear Regression

During this project we have gathered enough evidence of the correlation between HEMs and the Execution Time of an Analysis Task under contention, as seen in the Table 3. With that information we can construct a linear regression to model the contention as a function of the HEMs. Said models can incorporate more than one HEM if needed to decrease modelling error. However, the evidence points towards heterogeneous relationships between HEMs and ET depending on the AT, i.e. a fixed HEM can have a different correlation with the ET depending on the AT. Therefore, a single general linear regression cannot cover satisfactorily the modelling of contention for any AT. This creates the necessity of building a linear regression model for each AT. Our experimental setting is a set of AT kernels and a set of contending benchmarks. An experiment consists of the tuple (AT, CT, CTn, CPUMap), where AT is the task under analysis, the CT is the contender, the CTn is the number of contenders (all of which are copies of CT), and CPUMap indicates in which core these contenders are located. To construct a model for each AT, we gather all possible combinations of CT and CTn, with the CPUMap indicating that all contenders are within the same cluster.

Table 3 - Correlation, Magnitude and Variability of selected HEMs.

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

5.2.1 Modelling Approach

In our experiments we read 119 HEMs including the Execution Time. For a fixed AT, we can reduce the number of variables by filtering through low correlation HEMs with the Execution Time. Example classification of HEMs wrt correlation is reported in Table 3 where we summarize Correlation, Variability, and Magnitude for the L2 Write benchmark. Even after the filtering, we still need to find out which combination of HEMs produces the best contention model with the linear regression. In order to exhaustively find the best model, we perform the next two steps.

1. **LASSO**: standing for Least Absolute Shrinkage and Selection Operator, LASSO is a variable selection algorithm suited for linear regression models. LASSO is a more sophisticated approach than simply filtering through correlation, LASSO forces the sum of the coefficients in a linear model to be less than a fixed value, therefore, in an effort to construct the best model, only the most important variables will have a coefficient different than 0.
2. **Best Subset Selection**: once the most important HEMs are selected, we still need to find out which combination of HEMs produces the best model. In order to ensure finding the best one, the Best Subset Selection algorithms iterates through all possible combinations of HEMs, starting from a one HEM model, up to *p* HEMs, outputing for each case the best model, that is the one with highest R^2, the coefficient of determintation, which indicates the variability explained by the model. In our experiments, setting the maximum amount of HEMs in a model to *p = 5* is enough to produce satisfactory models.

5.2.2 Data Gathering

HEM data readings are limited to the number of available PMC in the platform. In the Orin, we are limited to 6 HEM readings at once, although we are able to read both the HEMs of the AT and the contenders. The contender HEMs are aggregated upon reading, thus only one aggregated count

per HEM is used on the models. It is crucial to remember that two groups of HEMs read in two different runs cannot be compared, due to different execution conditions. Therefore, constructing linear regressions with the Best Subset Selection algorithm requires all HEMs to be read with every other HEM in order to input them to the algorithm. This is too time consuming, therefore a more sophisticated approach is used in this project. We resort to using HRM [16] (Hardware Reading and Merging) which merges different HEM readings using an anchor HEM as reference. In this case, because we want to maintain the relationship between ET and the rest of the HEMs, we use ET as an anchor. The output of HEM is a dataset where for each execution time value, we have the corresponding values of HEMs as if they had been read together by using order statistics.

5.2.3 Results

Let us show a snippet of the results we generated on the Orin. In Figure 27 below we show an example of the best model found by the algorithm for the AT *matmulbasic_mem*. In this case, the Best Subset Selection algorithm found that 3 HEMs were enough to model the contention, with an $R^2 = 0.976$. In the picture we can observe the actual formula of the linear model, with the regression coefficients for each HEM.

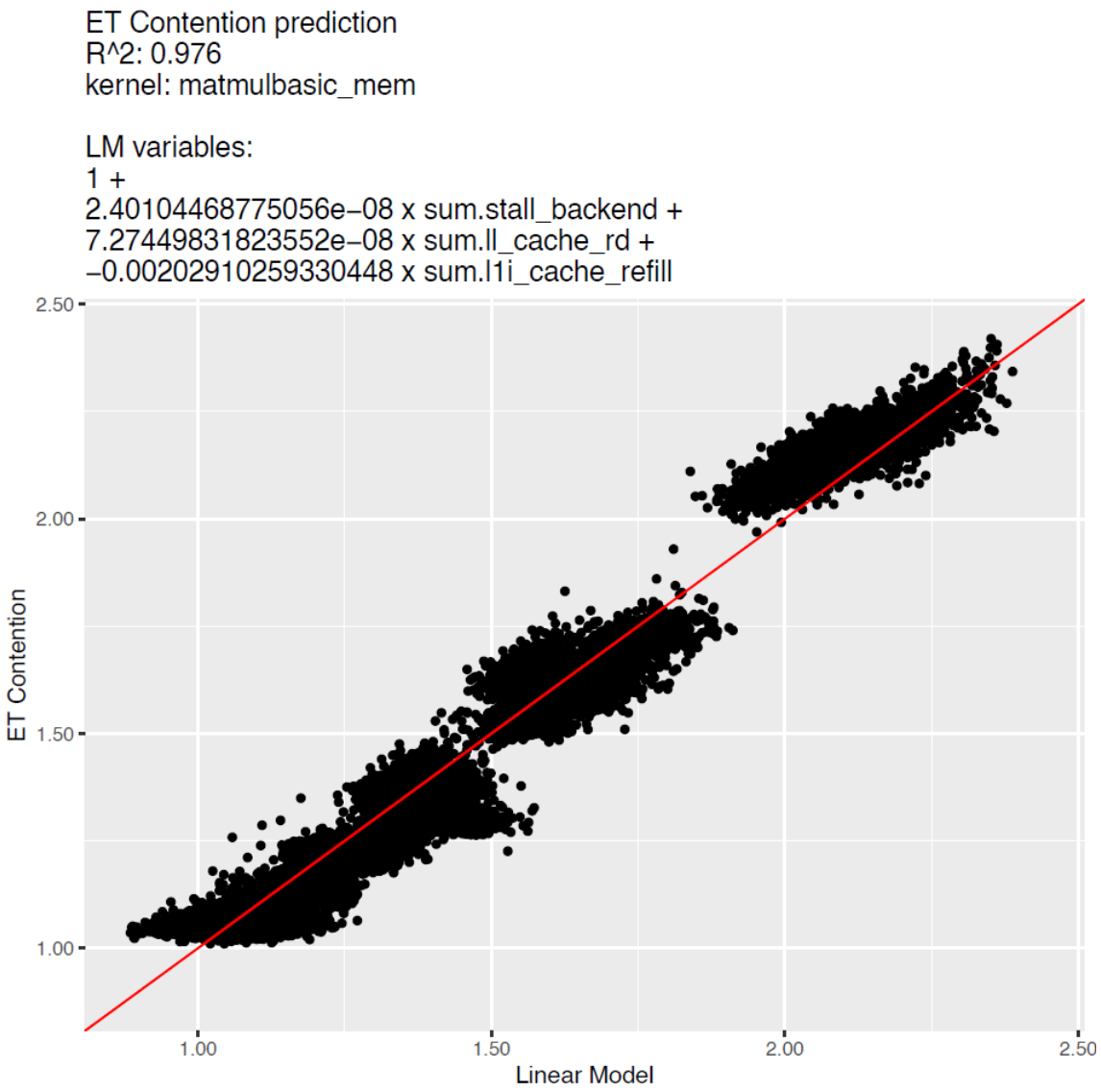

Figure 27 - Example correlation model for mamulbasic_mem benchmark.

In the Table 4 below, we observe the resulting performance of the models for each AT with the R^2. Note that the majority of the models have good performance, with 43% of the models having an $R^2 > 0.9$.

Table 4 - Correlation model performance expressed as R^2.

Kernel	R^2	Kernel	R^2
im2col_l2	0.874	**matmulttransp_l4**	0.664
im2col_l3	0.848	**matmulttransp_mem**	0.634
im2col_l4	0.552	**quicksort_l2**	0.4
im2col_mem	0.69	**quicksort_l3**	0.893
matmulbasic_l2	0.737	**quicksort_l4**	0.924
matmulbasic_l3	0.901	**quicksort_mem**	0.94
matmulbasic_l4	0.975	**relu_l2**	0.783
matmulbasic_mem	0.976	**relu_l3**	0.939
matmultiled_l2	0.122	**relu_l4**	0.955

matmultiled_l3	0.789	**relu_mem**	0.95
matmultiled_l4	0.859	**vector_dotprod_l2**	0.644
matmultiled_mem	0.962	**vector_dotprod_l3**	0.923
matmulttransp_l2	0.733	**vector_dotprod_l4**	0.924
matmulttransp_l3	0.689	**vector_dotprod_mem**	0.966

5.3 Timing interference control with CGuard

Multicore timing interference identifies the timing penalty or impact from contending requests on the same hardware component, arising on multicore execution platforms with an increasing number of shared resources. Timing interference is a well-known concern for timing analysability and predictability. As such, it has been explicitly addressed in domain specific standards and regulations [17] [18].

As timing interference is stemming from specific shared hardware components, the so-called interference channels, it can be in principle avoided or at least mitigated by controlling the way resources are shared across applications. The main sources of interference in the AGX Orin platform have been identified and explained in [1]. Available mechanisms and configurations that can be exploited to limit the amount of interference in the system have been discussed in Section 3.2.

However, it is important to observe that it is practically impossible to remove all sources of interference altogether unless we opt for a fully segregated setup, with evident drawbacks on performance. More pragmatic approaches resort to a more holistic approach where the potential amount of interference is minimized by resorting to the available support at platform and system software level, and the residual potential interference is eventually captured at operation by deploying a contention regulation mechanism [10] [13] [14]. The problem of guaranteeing *freedom form interference* therefore is divided into two steps: minimization by design, and protection at run-time.

In SAFEXPLAIN we embraced the same philosophy, and complemented the platform level solutions, at design and configuration time, with a highly modular and portable on-line monitoring mechanism for contention control. The tool we designed and developed in SAFEXPLAIN is called *CGuard*, which stands for contention guard.

5.3.1 CGuard design

The CGuard mechanism is inspired by existing bandwidth regulation approaches [10] [13] [14] to monitor and control the activity of non-critical tasks, which do not require protection from timing interference, and prevent them from exceeding predefined utilization thresholds for shared hardware resources (the source of timing interference). CGuard is used to temporarily pause the execution of non-critical tasks when they generate more interference than allowed, hence protecting the critical tasks to terminate its execution within the allocated timing budget.

CGuard differentiates from existing approaches in the fact that it does not rely on specific hardware or system software support (with enhanced portability and modularity) while still providing comparable performance. Further, the CGuard can deploy different contention modeling approaches, including the one based on linear regression (see Section 5.2).

Below we describe the process and mechanism we have defined to control the contention critical tasks (CT) may suffer from non-critical tasks (NCTs). The process is partially automated, and it encompasses a pre-operation phase and an operation phase. Each phase carries several steps.

In the pre-operational phase, the contention thresholds must be identified for each CT. Such thresholds can be derived based on the quasi-final deployment scenarios or, alternatively, with augmented scenarios, relying on synthetic benchmarks [19] to generate a configurable amount of contention. The pre-operational phase may also include the test batches required for the derivation of the linear contention formula (see Section 5.2).

In the operational phase, instead, the mechanism is instantiated by a set of specialized entities (or components) that are deployed at the same time on the execution platform.

The CGuard mechanism builds on ROS2 support for `callback_groups` and `executors`, available starting from ROS2 Humble version. These features allow two (or more) executors with different priorities to be mapped to the same core. In practice, we can define a high-priority thread that can always pre-empt the standard (low-priority) thread responsible for the node nominal behavior of an NCT. The high-priority thread is associated to a callback that can be programmed to perform critical and urgent actions such as sharing the core hardware event monitors (HEM) values read via Performance Monitoring Counters (PMCs) collected through PMULib or preempt the low-priority thread, obtaining the same effect as forcing the component into throttling. The latter state corresponds to pausing the NCT for the necessary time to terminate the execution of the CT, or at least till the next monitoring window in case the configuration allows.

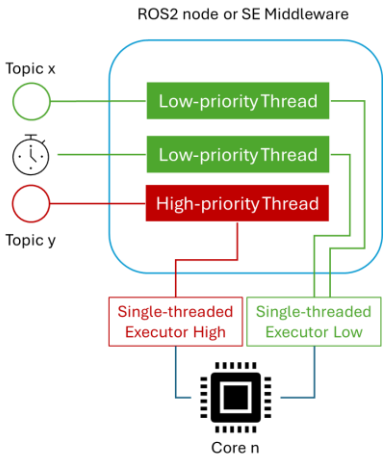

Figure 28 - Basic building block.

ROS2 concepts for multiple executors with different priorities on the same node are summarized in Figure 28, relative to an NCT. As shown in the figure, we can define multiple executors and associate them to different threads that, in turn, are associated to specialized callbacks in the NCT ROS2 node. This allows the NCT to perform three main functions: (i) its nominal behavior, (ii) sending the PMC data to the contention control module, and (iii) entering into a paused/throttling mode.

The contention control mechanism is implemented by a dedicated node on a dedicated core. The CGuard control node is responsible for:

- Collecting the NCTs PMCs in a centralized place
- Applying the formula or model to detect threshold exceedance
- React consequently by triggering the throttle state
- Re-enable the NCT nominal behavior when needed

We detail below the main functionalities required from the different entities in the CGuard mechanism.

Enabling monitoring and control

The control is always on for NCTs but the controller needs to be enabled (by the CT upon each activation) by publishing a special value to a dedicated topic (on/off on control node). Once enabled, the controller can enable and disable the monitoring by publishing a special message to

a critical topic for NCTs, associated to a high-priority callback executor in charge of setting the status variable to ON. One topic can be used for all NCTs monitored by the CGuard module.

HEMs collection

When monitoring is enabled, each NCT provides values of relevant HEMs periodically by publishing them to a topic to which the controller node is subscribed. NCTs periodic callback for PMCs is not enabled if the monitoring is disabled. When enabled, the PMCs for each NCTs are reset or taken as a reference for the next PMC read.

Contention model

When enabled, the controller applies the contention model (e.g. the linear formula defined according to Section 5.2) to detect threshold exceedance periodically or after he receives all updates from all NCTs (this is modelled with critical and non-critical topics or periodic activation of the controller node). Updates will provide cumulative values for each HEM or the values for the last monitoring period only. This is a design decision and can be configured.

Reaction

If the formula threshold is met, the controller publishes a special message to a critical topic on NCTs. The topic is different from the one for enabling/disabling monitoring. One topic can be used for all NCTs. The callback associated to this topic has high priority in the node and implements an endless empty loop polling for the status variable to THR. The local variable is reset to OFF by another high-priority callback, triggered by the controller node when the CT is done so that the throttling task terminates and even the NCT resumes its nominal execution.

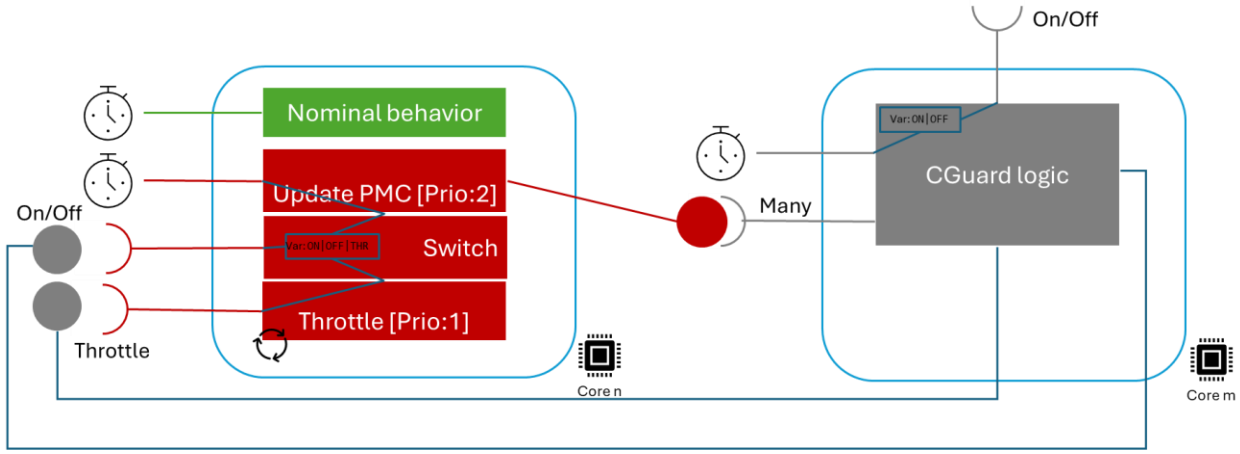

Figure 29 - CGuard mechanism overview.

Configuration parameters

The CGuard mechanism offers several configuration parameters that have direct impact on the effectiveness and overhead of the mechanism. The main parameters comprise the PMC callback frequency (impacting the granularity at which the monitoring is enforced) and the monitoring window (setting a trade-off between protection of the CT and progresses on the NCT).

5.3.2 CGuard integration

The CGuard mechanism has been prototyped in a clean ROS2 environment and later ported and integrated in the SAFEXPLAIN middleware, for easy integration in the use cases.

The integration of CGuard has been implemented by extending the `BaseApplication` class in the middleware to provide the necessary control in terms of thread mapping to callbacks and their priority of execution. The `BaseApplication` is extended to obtain specialized middleware nodes for:

- the `ContentionGuard` class, responsible for the implementation of the monitoring and control logic, including the contention modelling. The CGuard node keeps the functionalities of a `BaseApplication` with respect to system-level Diagnostic and Monitoring functionalities, including Alive monitoring, Health manager and Status manager.
- the `CriticalApplication` class, which extends the standard middleware node with specific support to set and reset the monitoring window and trigger the contention guard mechanism. It also keeps the functionalities of a `BaseApplication`, such as Alive monitoring, Health manager and Status manager.
- the `NonCriticalApplication` class, which models the NCT task by definingthe set of callbacks to support the CGuard implementation, using PMULib to read PMCs and share them with the `ContentionGuard` node. As for the other specialized node, the diagnostic and monitoring features are still supported. A slight modification has been applied to make sure that the Alive monitoring is not triggered when the NCT enters in throttling mode.

Figure 30 below provides a schematic view of the main integration concepts for an NCT node, which is the most complex specialization node due to the additional number of callbacks and associated threads.

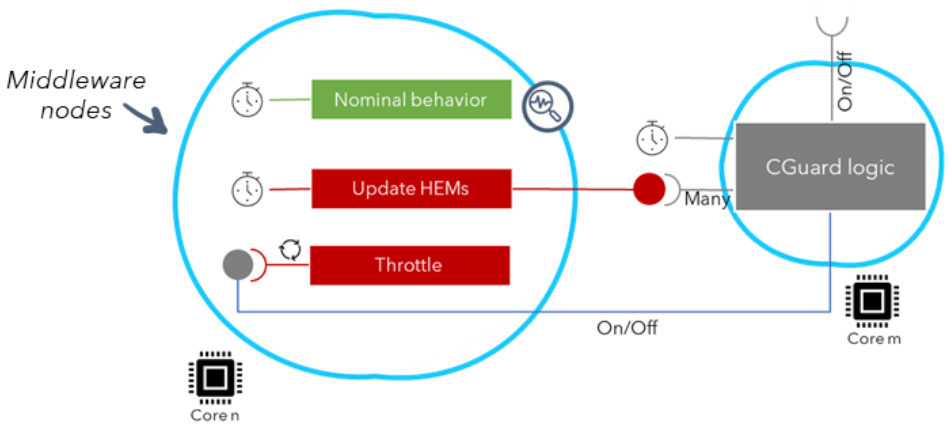

Figure 30 - CGuard integration with Middleware.

5.3.3 CGuard validation

We have performed an initial validation of the CGuard mechanism to assess its effectiveness and accuracy in capturing the timing interference. To this extent we focused on high contention scenarios as they represent the most critical operational conditions for the CGuard. In the scope of the SAFEXPLAIN setup, we must consider that CGuard will be deployed in a controlled contention scenario, thanks to the hardware and system software configuration. In that case, CGuard will be instrumental to capture residual interference that may happen at operation and that exceeds the interference thresholds considered acceptable at analysis time.

Experiments have been performed to assess CGuard overhead, effectiveness in protecting the CT, and impact (slowdown) incurred on the NCT. As CT we used the same matrix multiplication

function used to assess the impact of platform configuration in Section 3.2.1. As NCT, instead, we used an aggressive benchmark [20] trying to saturate the memory bandwidth, resulting in a more challenging scenario for CGuard.

PMC callback overhead

We measured the overhead of the PMC callback. We measured the cost of executing the PMC callback on the NCT under different PMC callback frequencies. While we do not expect this to largely affect the execution, we wanted to rule out hidden dependencies. Results, reported in Table 5, show the PMC callback is incurring a negligible overhead, averaging 16 micro-seconds.

Table 5 - PMC callback overhead.

PMC Callback [us]	200 us	500 us	1000 us	Overall
average	14.02	17.47	18.42	**16.64**
median	14	17	18	**14**
min	5	5	8	**5**
max	30	36	32	**36**

PMC callback cumulative impact on NCT

The cumulative overhead brought by the PMC measuring and callback on the NCT is reported in Figure 31, showing the slowdown incurred on NCT execution in relation to the frequency of the PMC callback execution. Clearly the cumulative impact is higher with higher frequency (left) and diminishes as long we decrease the frequency at which the PMCs values are sent to the CGuard module. It is worth nothing that starting from 1 ms, a reasonable target frequency on our setup, the overhead on the NCT becomes negligible.

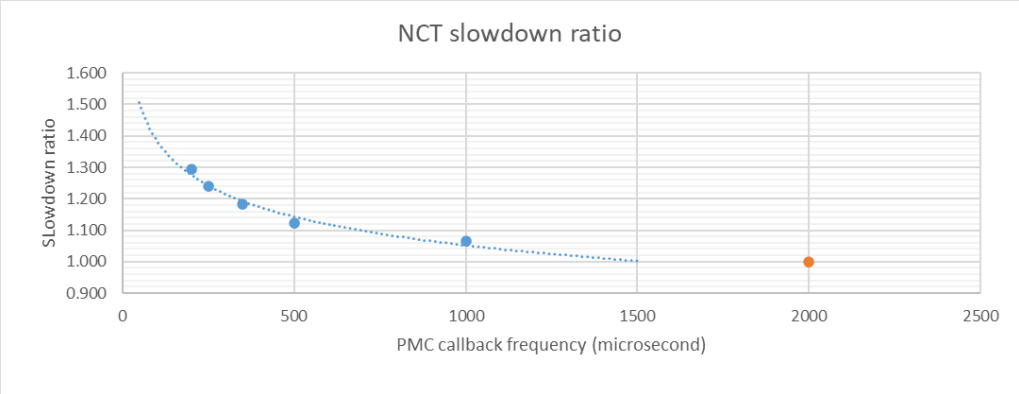

Figure 31 - Cumulative impact of CGuard monitoring on NCT.

Effectiveness of CGuard

Despite the aggressiveness of the benchmark used as NCT, CGuard is extremely effective in protecting the execution of the CT from unwanted interference arising from shared memory accesses. It is worth noting that, to make things even worse, we deployed the NCT on the same Cortex A79 core cluster in the Orin [3].

Results in Figure 32 show the relative increase in execution time of the CT (where 1 represents the performance of the CT in isolation) with varying frequency of the monitoring, as implied by the PMC callback frequency. Even under the looser frequency scenario (1 ms) the CT only incurs a 4%

increase in execution time, which is in fact pretty low considering the aggressiveness of the deployed NCT.

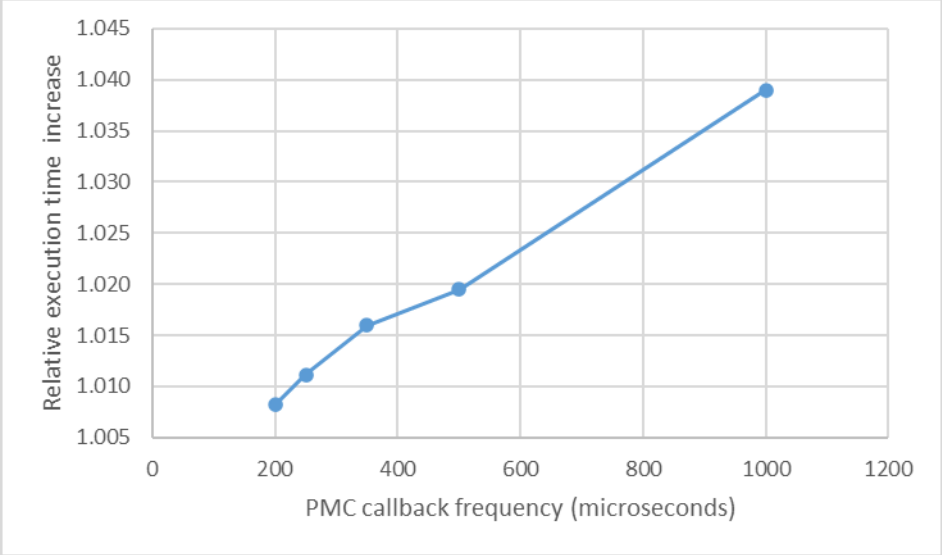

Figure 32 - CGuard effectiveness in protecting CT execution form interference.

5.4 Timing characterization integration and automation

In this third phase of the project, we worked towards an improved integration of the probabilistic timing analysis solution. In particular, we focused on increasing the level of automation support on the middleware. We aimed at providing a streamlined flow for collecting the (timing) profile of a generic middleware application for which we want to compute an execution time upper bound.

We leveraged the available main elements in the PMULib integration (see Section 4.3) to support not only the transparent collection of timing information on a target middleware node but also to automate the use of the RestK probabilistic timing analysis script to derive probabilistic WCET bounds.

As a main constraint, we wanted to ensure the timing analysis tool is not executed on the target platform but on an external device, in order not to overload the execution on the Orin. In fact, while the script itself is not excessively onerous in terms of computation, it also generates graphical representations of the analysed sample, fitted distribution and alike.

We built on top of ROS2 DDS functionality to implement a standard ROS2 remote note that can connect to the Middleware `PMULogger` to retrieve locally all the collected information (either on-line or off-line at specific execution stages). The remote node is then responsible to filter and preprocess the data obtained from the logger and call the RestK R script. A wrapper has been provided for the RestK script that takes care of generating different statistical plots for the computed distribution and the respective bounds. The plots are automatically rendered on screen in the remote node.

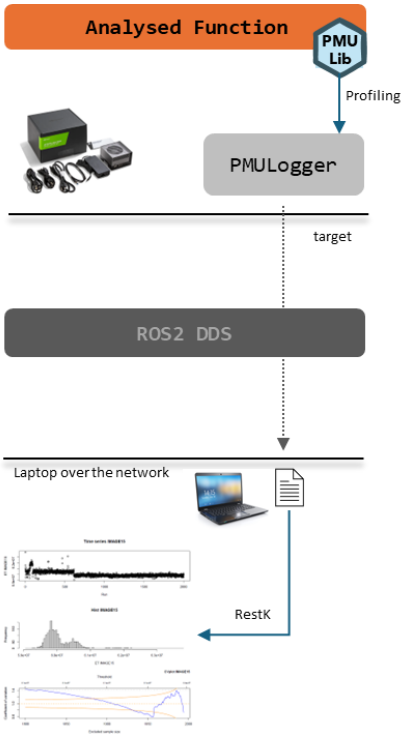

Figure 33 - Timing analysis automated flow with remote node.

Figure 33 shows the local and remote infrastructure for the analysis automation. The PMULib automatically and transparently collects measurement data from the execution of the analysed function on the middleware and logs the information on the `PMULogger`. The latter is directly accessible from a remote ROS2 node through a VPN. Data is then processed locally to obtain the timing distribution and bound for a reference exceedance threshold (see [1] for more details).

The automated framework has been already deployed to support the SAFEXPLAIN open demonstrator, introduced in Section 6.3.

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

The main objectives of this task during the third phase of the project were mainly related to finalizing the implementation of already agreed and emerging middleware required functionalities, as well as improving the support for automated V&V tasks. Besides, we also provided continuous support to facilitate the mapping of FUSA architectural elements (and Safety Patterns) to middleware nodes and features, and to facilitate the porting of the case studies on top of the middleware and platform. In line with its overall objective, the task aimed to provide comprehensive support to all platform level requirements.

In this third phase we kept consolidating our strategy around the concept of SAFEXPLAIN Middleware as a common ground for deploying and integrating all technological elements developed to support analysis and platform level concerns in general. Figure 34 provides a high-level view, on an architectural example, of the large and diverse set of platform-level elements and tools that are supported by the SAFEXPLAIN execution platform.

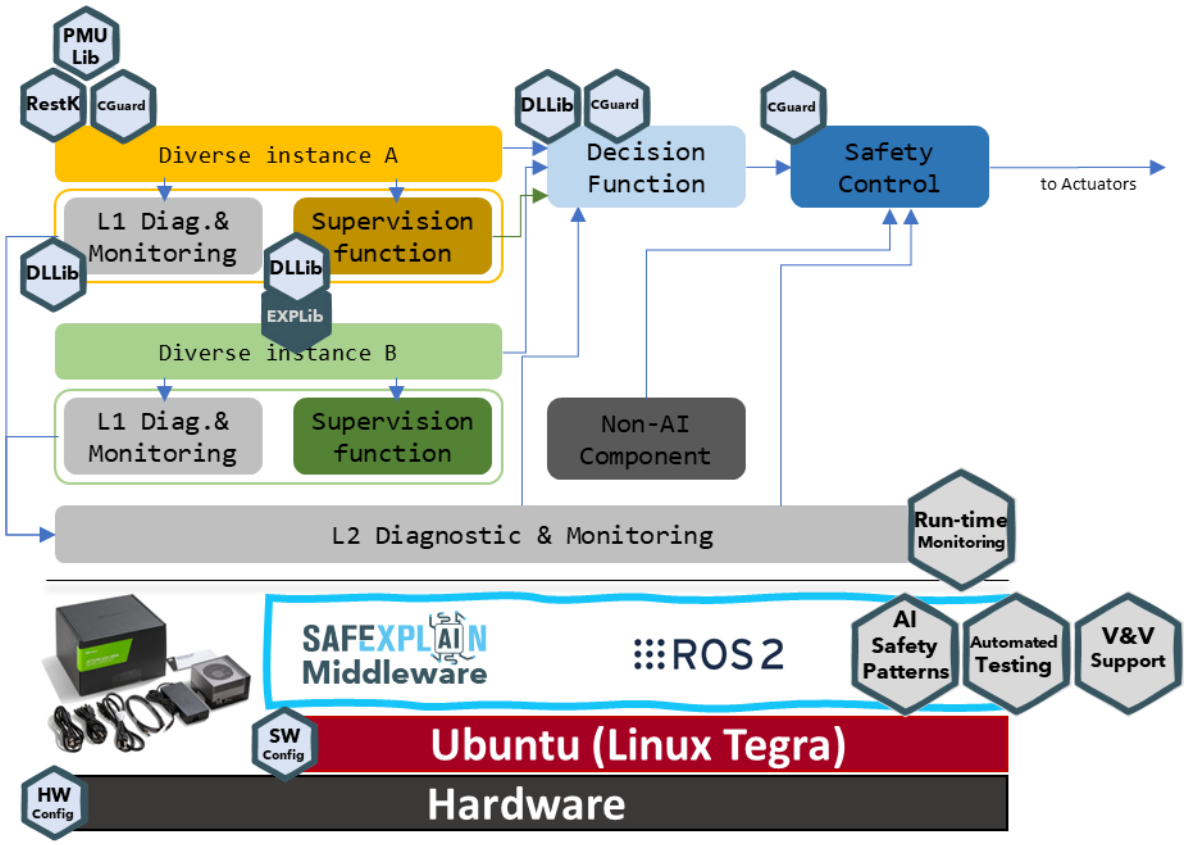

Figure 34 - Summary of platform-level support.

All the elements in Figure 34 have been already introduced either in the previous Sections (e.g. PMULib and CGuard) or are described in other project deliverables (e.g. RestK in [1]). The interesting information we can draw form the picture is that the SAFEXPLAIN ecosystem includes a set of tools and methodologies that operate on the platform configuration at hardware (HW Config) and OS (SW Config) level and enable SAFEXPLAIN building blocks (FUSA Safety Patterns, Explainable AI, V&V support, Predictability, and Performance). The concept of Middleware is a critical enabler for the whole ecosystem.

Besides improvements and refinements to the middleware to support SAFEXPLAIN tools and methodologies, efforts have been devoted to realizing a concrete instantiation of the methodology in the form of an *open demonstrator*, that can be used to showcase the methodology beyond the scope of the consortium.

In the following, we report on the progress achieved in the Middleware functionalities to support application requirements, FUSA patterns, and V&V task. Finally, we report on the specific configuration and additional features deployed to support an open demonstrator, showcasing all relevant SAFEXPLAIN results.

6.1 SAFEXPLAIN Middleware concept

The SAFEXPLAIN middleware has been designed to serve as a common abstraction layer to accommodate all platform level requirements on top of the execution platform (hardware and system software). This provides a consistent execution environment where all SAFEXPLAIN tools and methodologies are naturally integrated and can be straightforwardly exploited by the platform user at development, analysis, and operation.

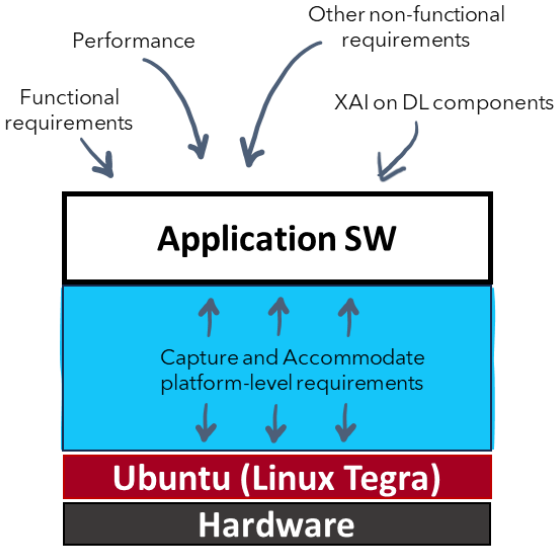

Figure 35 - SAFEXPLAIN Middleware overview.

The middleware has been designed to capture both self-imposed requirements and all sorts of requirements emerging from the other work packages on the execution platform. It positions in between the execution platform and the user application, see Figure 35.

The Middleware has been instrumental to:

- *Ensure compliance with AI-FMS and FUSA architectural patterns by design*, avoiding the complexities and the pitfalls of delegating the full design to the end user, and ensuring the system is deployed following the selected safety pattern.
- *Facilitate the integration of the use case* on top of a simplified ROS2-based framework where low-level configurations become transparent to the user, and the porting effort is often reduced to move code snippets within predefined placeholder nodes in the middleware.
- *Provide automation support for recurrent V&V tasks*, with special focus on providing an efficient environment for SAFEXPLAIN verification and validation requirements and tools

- *Seamlessly support traditional software-level monitoring solutions*, by collecting functional and non-functional metrics at operation and support state-of-practice life-cycle monitoring and management.

The Middleware has been designed and developed 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 36 below represents how the middleware, on top of the combination of hardware and OS layer, integrates a set of tools (PMULib, CGuard, RestK, and DLLib) and methods (HW and SW Config) to support a wide and diverse set of requirements and features, ultimately supporting the whole SAFEXPLAIN methodology.

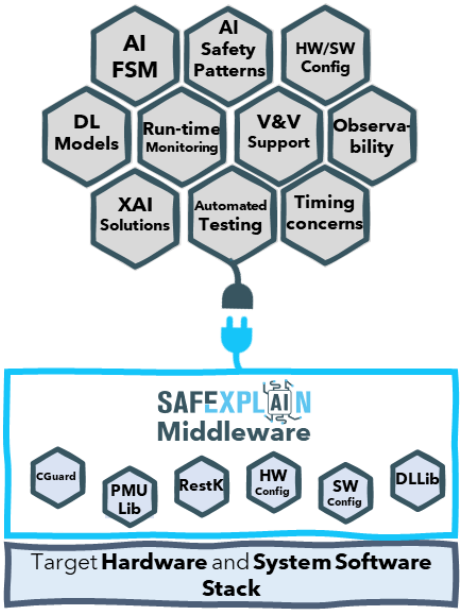

Figure 36 - SAFEXPLAIN middleware embedded tools and methodologies.

PMULib, CGuard and RestK are specific tools developed within this same work package. DLLib is the software library providing a concrete instance of EXPLib solutions; both have been developed in WP3 and are described in [21].

In the next sections we will focus on the main updates and refinements to the Middleware layer and will introduce the main features in the SAFEXPLAIN open demonstrator.

6.2 SAFEXPLAIN Middleware support

As introduced in the second phase of the project, the SAFEXPLAIN middleware provides several libraries and services to standardize access to the core functionalities of the platform and meet a wide set of platform-level requirements, as represented in Figure 35 and Figure 36. During the third phase of the project, we devoted our efforts, on one side, to enriching the functionality (and solve issues and bugs) of the already present features, and on the other side, to complete the missing parts with respect to the requirements specified in WP1 and emerged during the project.

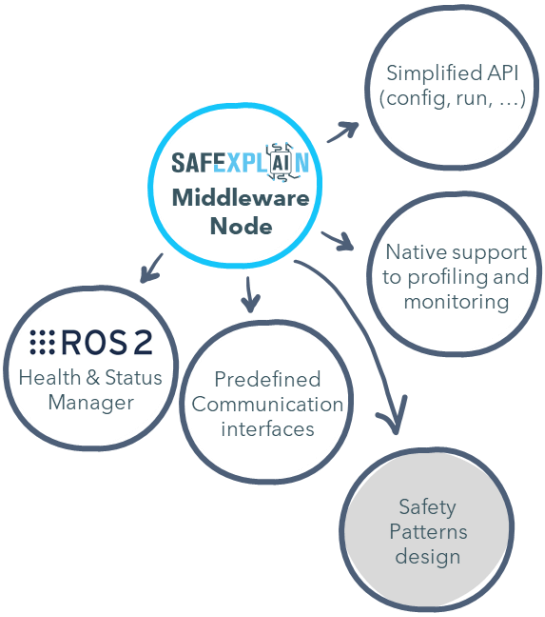

Figure 37 - Middleware relevant features.

In the following discussion, we aim to present the main updates to the software packages developed in the context of the SAFEXPLAIN project. To provide a complete picture of the platform developed, the package description will recap and update the previous discussions developed during the second phase of the project.

Base Application Class. In the SAFEXPLAIN Middleware, the safety-related applications shall be ROS2 nodes derived from the `smw_base_application::BaseApplication`, which is a highly specialized node that gives access to the user support to the platform features, such as IPC communication, logging, reliable data persistency, timing analysis instrumentation, supervision, etc. The main relevant features offered by the middleware `BaseApplication` are summarized in Figure 37. All these features concur in the realization of a simplified design and deployment environment for complex and modular AI-based applications.

Considering the typical need of a safety application of meeting strict timing constraints, the `BaseApplication` sets up (out-of-the-box) a function `initialize()` called once to perform setup of the application, callback function called `run()` that's periodically called upon a user-defined time expiration, and a `terminate()` function called to perform housekeeping / clean-up during application shutdown. The effort of the user that aims to write a `BaseApplication` is mostly towards defining those three functions.

The state of the `BaseApplications` (which can be Unconfigured, Inactive, Active and Finalized) evolves over time thanks to the coordination of the `lifecycle_manager` (within `smw_lifecycle_manager` package). The overall state of the platform instead is kept by the `state_manager` (within `smw_state_manager` package).

Lifecycle management. The `lifecycle_manager` ensures synchronization across all tasks within the platform, evaluating the preconditions required to configure, activate, and finalize an application. During the initialization phase, it verifies that all processes are spawned by the platform launcher. Before execution, it ensures that supervision mechanisms are up and running.

Health management. The platform health manager (implemented as the `health_manager` node in the `health_manager` package) oversees the supervision of all `BaseApplication` instances, providing both temporal and health monitoring. Each `BaseApplication` reports its status during

the `run()` cycle, publishing an "*alive*" and "*health*" message. Upon publishing, an internal timer within the ROS 2 DDS layer is triggered. If the elapsed time between two consecutive status reports exceeds a user-defined threshold, DDS—via the ROS 2 RMW and RCL layers—activates a debounce counter within the health manager to track the violation. If this violation persists for a predefined number of `run()` cycles (also configurable by the user), the health manager initiates a predefined reaction based on its configuration.

Given the target maturity of the platform software, two response mechanisms have been implemented:

1. Triggering a platform state change – This instructs the `state_manager` to transition to `SAFE_STATE`. Applications managing platform control output variables recognize this state change, helping to prevent unpredictable behavior.
2. Terminating an application – This allows the supervision node to terminate a process that may be disrupting resource access for other platform components or interfering with their functionality.

Although not yet implemented, the current architecture is designed to support more advanced response mechanisms in the future, such as coordinated actions across multiple applications or automatic process restarts.

Data persistence. To enhance overall safety, two mechanisms have been implemented to improve the detection of corrupted persistent data.

The first mechanism is based on *image hashing*, designed to protect and uniquely identify recorded camera input data. During the build process, a hash is computed for each input frame using a standard algorithm and stored in a text file. At runtime, whenever an image is broadcast, it is sent along with its precomputed identifier. Upon reception, the hash is recomputed and compared to detect any corruption.

The second mechanism ensures the integrity of configuration parameter files used by safety-critical applications. During the build process, a *Cyclic Redundancy Check* (CRC) is computed for each configuration parameter file and stored in a .crc file alongside the original file. At runtime, during application startup, the user can invoke the `safe_load_parameter` method from `smw_persistency`, which loads the parameter file, computes its CRC, and compares it against the stored CRC value. If a mismatch is detected, an error is triggered.

Data communication support. Additionally, in the third phase of the project, the communication library has been enhanced with the introduction of the `SynchronizerServiceProxy`. This service enables the synchronization of multiple messages from different sources based on a numerical attribute. It has been extensively deployed, with different objectives, across various applications, such as to synchronize:

- Left and right camera frames according to their timestamps in the railway use case.
- Outputs of two redundant nodes according to the input image ID, as in the demonstrator.

6.2.1 Support to FUSA Architecture

Supporting FUSA architectural patterns by design is one of the main features of the SAFEXPLAIN middleware. In this third phase of the project, we devoted our effort to improving and refining the

modeling of the Safety Patterns in terms of components and connections between them, data types, and functionalities.

In the following, we provide a high-level mapping between architectural components and middleware nodes in a reference implementation. Considering that the functional-safety scenario of the use cases closely matches that of Safety Pattern 2 [2] in terms of functionality and role of the AI components, we will focus our review on the instantiation of that pattern.

We consider a reference setup which includes instantiations of the required software packages, communication interfaces, instrumentation, and a standardized parameter configuration, ensuring compliance with the selected safety pattern.

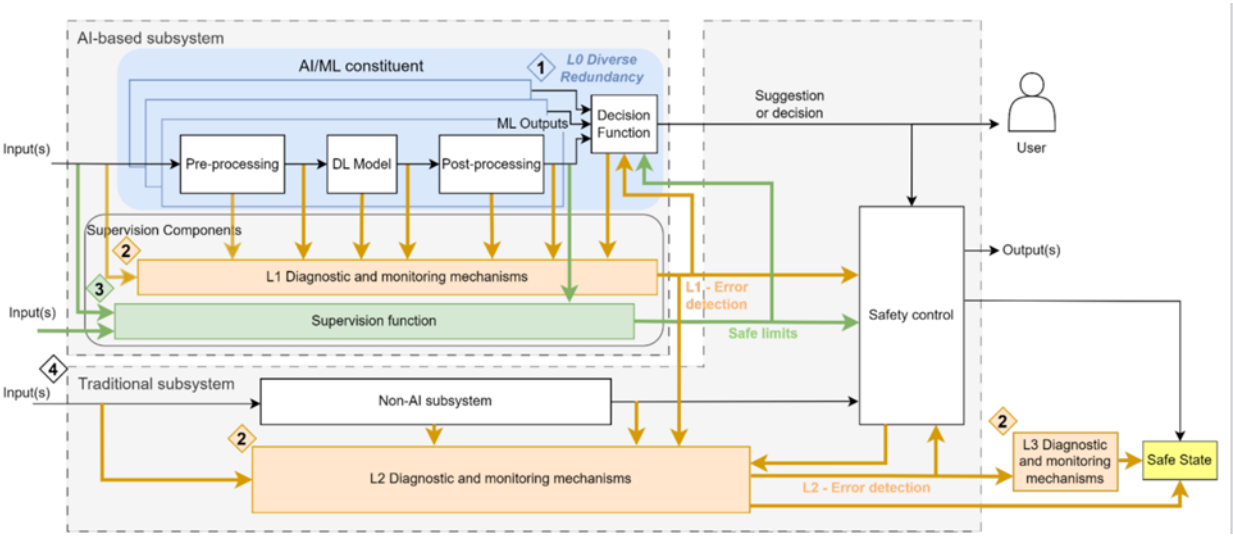

Figure 38 - Reference safety architecture pattern for Safety Pattern 2 (from [2])

Figure 38 illustrates the reference safety architecture for Safety Pattern 2. The architectural components (blocks) in the architectural design correspond to software packages and classes in the middleware deployment view.

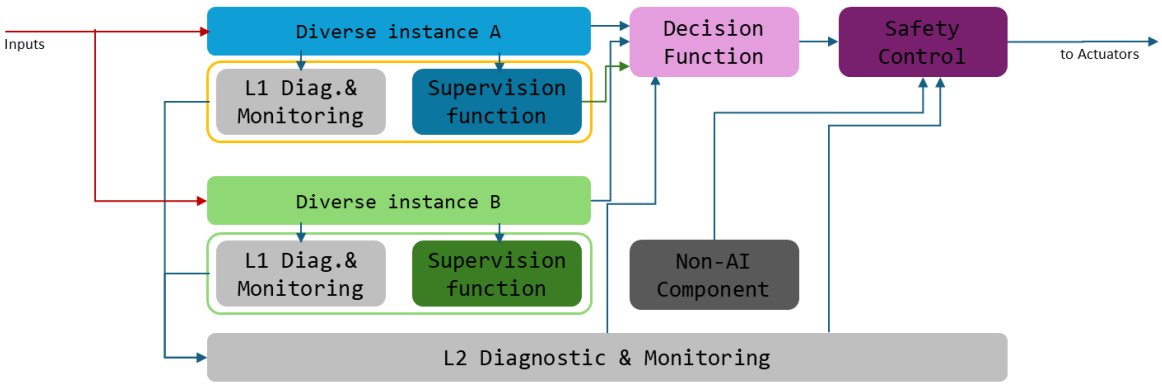

Figure 39 - Reference template instantiation for Safety Pattern 2.

In the reference Safety Pattern template instantiation (see Figure 39), these packages and classes can be considered as partially empty boxes that only need to be augmented with the specific semantics of the concrete system. The main software elements in the middleware are the following:

- `smw_ml_constituent`: contains the code for all AI/ML constituents, with multiple instances meeting the L0 diverse redundancy requirements. Each AI/ML constituent (represented by the classes `DLConstituent_01` and `DLConstituent_02` in the provided example) extends the ROS2 node concept with specific SAFEXPLAIN features and functionalities, inheriting from `smw_base_application::BaseApplication`. The user is expected to provide the semantics for the DL model, as well as the data pre-processing and post-processing modules, within this package.
- `smw_decision_function`: contains the `DecisionFunction` class, that shall be customized according to the use case; it takes the output from all redundant AI/ML constituents and supervision components to provide a unified decision to the safety control component.
- `smw_l1_monitor` and `smw_supervision_function`: accommodate the supervision components that complement the DL model's diverse redundancy. The latter component comprises AI-specific diagnostic functions, whereas the former provides standard FUSA diagnostic and monitoring support. These components detect runtime errors, model insufficiencies, anomalies in models and data, and other potential issues.
- `smw_health_manager`: Implements platform-level diagnostics and monitoring (L2 mechanisms) through the `health_manager` daemon. The only requirement on the end user is the definition of a configuration file for determining triggering conditions and reactions to detected issues. Once configured, the `health_manager` is responsible for triggering the necessary reactions to ensure system integrity.
- `smw_safety_control`: Contains the code for implementing system control and actuation logic. In the reference implementation, it simply forwards the output from the node that spawns the `DecisionFunction` class. In a more complex instantiation of the pattern, the safety control package is eventually connected to the traditional Non-AI subsystem, which is use-case dependent.
- `Non-AI Subsystem`: this component is stubbed in the reference middleware template for Safety Pattern 2 but it will simply include the deployment of a set of nodes extending the `smw_base_application::BaseApplication` and embedding a use case specific semantics.

6.2.2 Support to Verification and Validation

The Middleware software architecture accommodates also standard Verification & Validation concepts, in a consistent way with the AI-FSM concepts (see Figure 40). As already consolidated in the second phase of the project, the middleware software architecture supports standard verification and validation tests, classified as:

- **Unit tests**: Focus on individual code components, such as functions or objects.
- **Component tests**: Ensure that a module (e.g., a ROS2 node) behaves correctly by testing its interfaces without examining internal implementation details.
- **Integration tests**: Validate that multiple system modules work together, ensuring that applications can effectively utilize the platform's core functionalities.

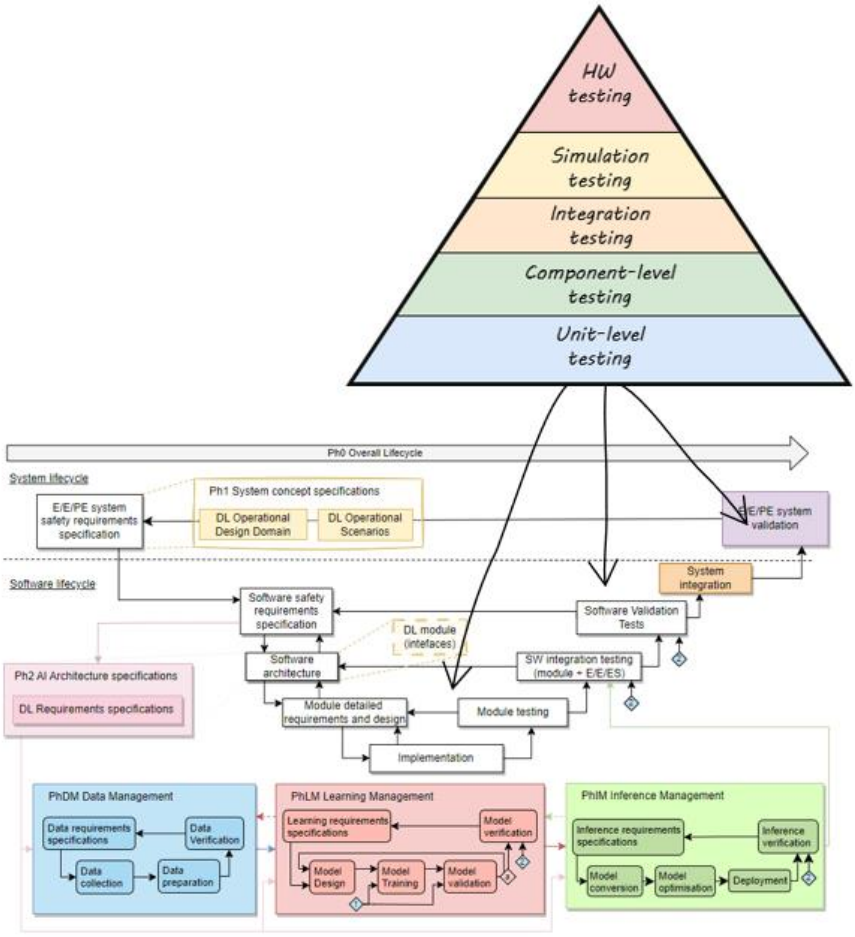

Figure 40 - Overview of the verification and validation process for the platform development (from [1]).

During the third phase of the project, we have increased the test coverage for all platform libraries, but complete coverage was not feasible due to time constraints, and therefore we prioritized the most critical components. Tests have been set up to be executed automatically within Docker containers as part of the CI/CD pipeline before merging updates into the main stable branch. Additionally, we conducted regular testing on the target hardware (NVIDIA Orin) following each major update to ensure smooth operation. This is crucial since behavioral differences may arise between the platform running on a standard desktop PC (x86-64 architecture) and the NVIDIA Orin (ARM64 architecture).

From a methodological perspective, we employed three primary types of testing: static code analysis, requirement-based testing, and fault-injection testing.

Static code analysis involves reviewing the source code without execution to detect potential issues and enforce coding standards early in development. An example of tool we've developed and used for static code analysis was the smw_apicheck.py (located in the scripts/ folder), which ensures that API definitions remain consistent between Python and C++ implementations. This verification helps prevent integration issues arising from function signature mismatches, given that C++ and Python applications run concurrently on the platform.

Requirement-based testing was the primary methodology used during the project to verify whether the libraries met their specified requirements. To enhance traceability, we've added a

detailed description, expected results, and a reference to the corresponding requirements for those kinds of tests.

To support application developers in testing their packages within the platform context, the `smw_testing` package provides valuable mocking tools and tailored testing utilities, such as:

- `topic_statistics_listener`: Verifies the frequency of message publishing by an application to ensure expected behavior and performance targets have been achieved.
- `param_update_tracker`: confirms that a specific application parameter holds the correct value, useful, for example, to verify if the applications have been correctly configured by the user and if the transitions into a safe state take place within a defined tolerance time interval.
- `utilities_from_cli`: Exposes various command-line interface (CLI) utilities within test scripts, facilitating testing automation.

Finally, fault-injection testing was conducted to assess the system's error-handling capabilities, particularly focusing on the Platform Health Manager. By simulating faults, we evaluated the platform's response to availability issues and health-related reporting, thereby increasing confidence in the robustness of software module integration.

In summary, during the third phase of the project, we focused on enhancing the SAFEXPLAIN middleware by expanding its functionality, addressing existing issues, and implementing the remaining features specified in WP1. Key additions include improved supervision and health monitoring mechanisms, enhanced data integrity validation through image hashing and CRC checks, the introduction of the `SynchronizerServiceProxy` for efficient message synchronization, and extended testing tooling. With these developments, it has successfully met all the requirements established during WP1 for the platform.

6.3 Demonstrator

The demonstrator aims to provide a simplified yet comprehensive representation of a reference AI-based system, highlighting SAFEXPLAIN technologies and tools, together with the platform's key resources. It was initially conceived as Proof of Concept (PoC) to facilitate the identification of potential issues, strengths, and areas for improvement, and to guide the implementation of platform features across three application domains: aerospace, automotive, and railway. During the second and third phase of the project, it matured into a more robust and consolidated example of application of the SAFEXPLAIN approach and is now considered as an open demonstrator that can be exploited to share and promote the project results outside the consortium.

The demonstrator models an application scenario that meets Safety Pattern (SP) 2 [2] specifications where the AI component is not exclusively responsible for the system behavior but can contribute to the decision process and ultimately on the operation. The reasons for going for SP2 were: first, that SP2 is the simpler pattern where the AI component inherits criticality concerns; second, the SP2 was closely matching the operational scenarios of most of the use cases. It was therefore natural to consider SP2 as the initial step for providing a reference setup. This reference setup includes instantiations of the required software packages, communication interfaces, instrumentation, and a standardized parameter configuration, ensuring compliance with the selected safety pattern.

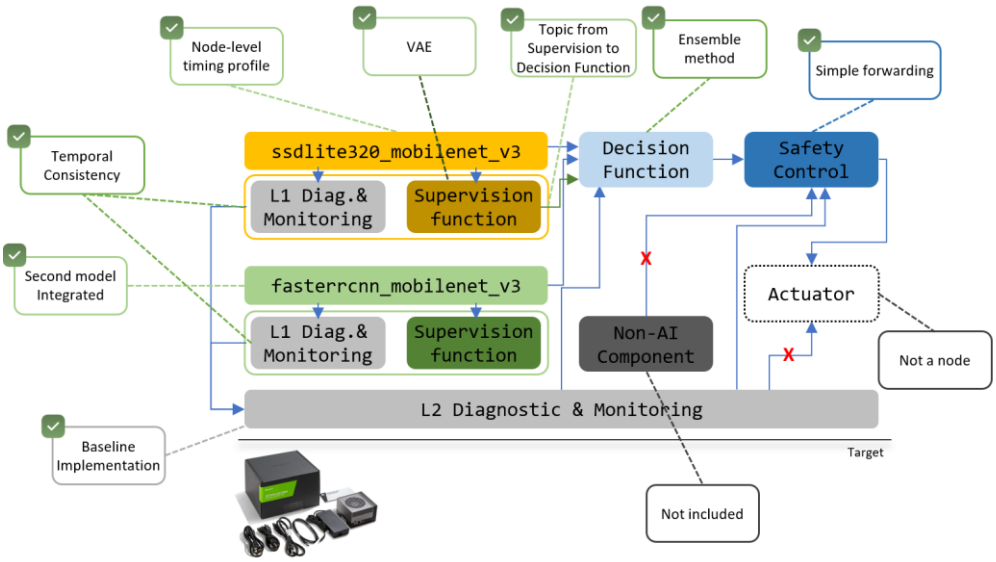

Figure 41 - Overview of SAFEXPLAIN open demonstrator.

The demonstrator includes a representative instance or prototype implementation of all the building blocks of SP2 that are also tailored to the specific functional behavior. The demonstrator was developed in close collaboration with WP2, WP3, and WP5. The resulting system, illustrated in Figure 41, models a satellite position tracker sub-system, featuring the following components:

- Two diverse and redundant AI neural networks for object identification.
- A supervision function based on Variational Autoencoders (VAEs) to detect anomalies in both input-output data and internal model behavior.
- An L1 Diagnostics & Monitoring module to verify the temporal consistency of input and output data.
- A decision function implementing an ensemble method that synchronizes data from different models, aligns detections, and produces a confidence-weighted output.
- A safety control module, which is meant to use the outcome of the decision function, together with diagnostic information, and ideally the result of a non-AI algorithm, to elaborate the command of be set to the system actuators (thrusters, gyroscopes, etc.). In the absence of a real or simulated actuator, the control module elaborates the decision function's results as a simple validation step.

Details on the implementation of each component — particularly the AI-related modules and supervisors — including an assessment of their maturity, obtained outcomes, and limitations, are provided in the deliverables for WP3 [21] and WP5 [22].

The system has been configured to execute under PM0 and to deploy a mapping that enforces a clear separation between critical components (decision function and safety control operating within a functional pipeline), monitoring components, and (segregated) diverse and redundant instances of the ML component.

Clearly, all custom software modules within the demonstrator rely on platform resources for execution control and monitoring. In particular, the L2 Diagnostics & Monitoring layer has been configured to perform real-time monitoring of all safety-relevant platform nodes. This setup

demonstrates how a monitoring violation can trigger a system-wide response designed to transition the platform to a Safe State.

6.3.1 Demonstrator functionalities

The baseline system deployment and operation have been also extended to provide a set of useful functionalities to support the showcasing of the internal information and events happening in the system to an external audience.

To support the execution of the demonstrator, the following software components have been developed:

- *Camera Node* (part of the `smw_sensors` package): This software component is designed to replay data for the case studies, specifically tailored for use in the demonstrator. It enables the platform's software modules to receive a continuous or step-by-step stream of input data from a camera, with the ability to pause playback as needed. Additionally, the tool allows users to switch the data source folder, which is particularly useful for automated fault-injection testing and performance evaluation. Furthermore, applications can utilize the `ros2bag` package for data recording, and other types of data may also be reproduced as needed.
- *Visualizer Node* (part of the `smw_visualizer` package): This component is responsible for displaying output and diagnostic data from the platform during execution. The visualizer node can operate either locally on the AGX Orin board or remotely on a desktop connected to the board through a private network. In the latter case, it leverages ROS 2's distributed communication framework to minimize interference with the platform's software execution, particularly in terms of CPU load and memory consumption.
- *Remote controller*: remotely connects via VPN to the camera node and allows to control the execution from a remote ROS2 node.

The visualizer node has been further generalized to allow overhearing all the information shared among nodes through topics in the middleware, which is especially useful when deployed on the remote host. The visualizer can be straightforwardly instantiated just by providing the topic identifier.

We also developed a visualizer specialization for supporting the timing analysis flow in an automated way, as discussed in Section 5.4, grouping on the same remote window both measurements from the `PMULogger` and automatically generated plots for the probability distribution and probabilistic bounds.

As an example, Figure 42 illustrates how the visualizer node aids in showcasing the demo results while also providing insights into the platform software's operation. This figure presents the synchronized output of the two redundant AI neural networks for a given input image, which is subsequently used to elaborate the final result in the decision function. Clearly, the visualizer can be tailored to each use case to display any internal variable of interest.

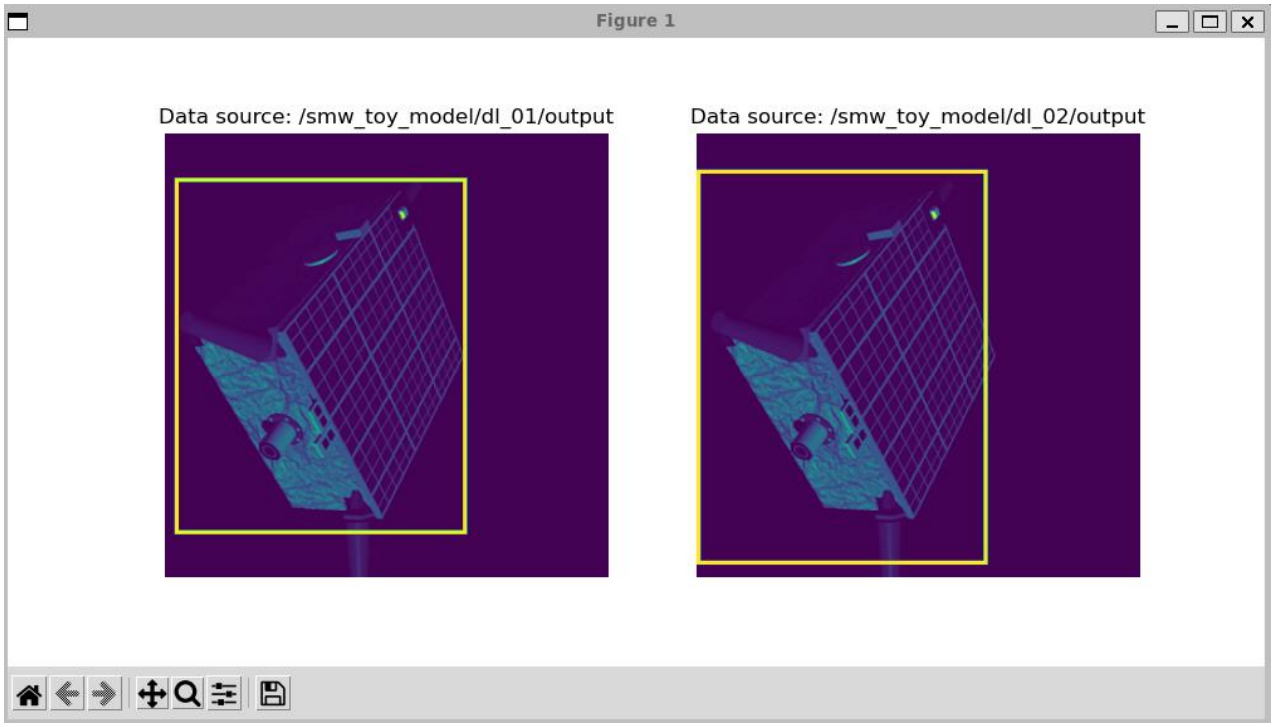

Figure 42 - Visualization node for the outputs of the redundant models.

7 Acronyms and Abbreviations

COTS	Commercial Off The Shelf
CUDA	Compute Unified Device Architecture
FUSA	Functional Safety
GPU	Graphics Processing Unit
HEMs	Hardware Event Monitor
HW	Hardware
OS	Operating System
PMC	Performance Monitoring Counter
PMU	Platform Monitoring Unit
ROS2	Robotic Operating System version 2
SW	Software
V&V	Verification and Validation

8 References

[1] SAFEXPLAIN, “D4.1 Platform Technologies Report,” 2024.

[2] SAFEXPLAIN, “D2.2 DL Safety Architectural Patterns and Platform,” 2024.

[3] 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 .

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

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

[6] The Linux Foundation, “Real-Time LInux,” 2024. [Online]. Available: https://wiki.linuxfoundation.org/realtime/start.

[7] L. K. J. A. F. C. J. Barrera, “Assessing the use of NVIDIA Multi-Instance GPU in the Automotive Domain,” *IEEE Embedded Systems Letter (to appear) ,* 2025.

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

[9] C. H. J. A. E. M. F. J. C. Jordi Cardona, “Accurately Measuring Contention in Mesh NoCs in Time-Sensitive Embedded Systems,” *ACM Trans. Design Autom. Electr. Syst. ,* vol. 28, no. 3, p. 34, 2023.

[10] H. Yun, R. Pellizzoni, M. Caccamo and L. Sha, “Memory Bandwidth Management for Efficient Performance Isolation in Multi-Core Platforms,” in *IEEE Transactions on Computers, vol. 65, no. 2, pp. 562-576*, 2016.

[11] A. Agrawal, G. Fohler, J. Freiteg, J. Nowotsch, S. Uhrig and M. Paulitsch, “Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study,” in *Proceedings in Informatics (LIPIcs), Volume 76, pp. 2:1-2:22*, 2017.

[12] I. Izhbirdeev, D. Hoornaert, W. Chen, A. Zuepke, Y. Hammad, M. Caccamo and R. Pellizzoni, “Coherence-Aided Memory Bandwidth Regulation,” in *IEEE Real-Time Systems Symposyum*, 2024.

[13] P. Sohal, R. Tabish, U. Drepper and R. Mancuso, “Profile-driven memory bandwidth management for accelerators and cpus in qosenabled platforms,” in *Real-Time Systems, vol. 58, no. 3, p. 235–274*, 2022.

[14] A. Zuepke, A. Bastoni, W. Chen, M. Caccamo and R. Mancuso, “Mempol: Policing core memory bandwidth from outside of the cores,,” in *IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, 2023.

[15] 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.

[16] S. Vilardell, I. Serra, R. Santalla, E. Mezzetti, J. Abella and F. J. Cazorla, “HRM: Merging Hardware Event Monitors for Improved Timing Analysis of Complex MPSoCs,” *IEEE Trans. Comput. Aided Des. Integr. Circuits Systems,* vol. 39, no. 11, pp. 3662-3673, 2020.

[17] International Organization for Standardization, “ISO/DIS 26262. Road Vehicles -- Functional Safety,” 2009.

[18] EU Aviation Safety Agency (EASA) and Federal Avionics Administration (FAA), “General Acceptable Means of Compliance for Airworthiness of Products, Parts and Appliances (AMC-20). Amendment 23. Annex I to ED Decision 2022/001/R. AMC 20-193 Use of multi-core processors,” 2022.

[19] M. Bechtel and H. Yun, “Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention,” in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[20] M. NIcolella, S. Roozkhosh, D. Hoornaert, A. Bastoni and R. Mancuso, “RT-Bench: an Extensible Benchmark Framework for the Analysis and Management of Real-Time Applications,” in *Real-Time Networks and Systems (RTNS)*, 2022.

[21] SAFEXPLAIN, “D3.3 Final DL components, libraries, and the DL interface,” 2025.

[22] SAFEXPLAIN, “D5.2 Case study porting and integration,” 2025.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] D. Gordon, “Covering Designs,” [Online]. Available: https://www.dmgordon.org/cover/.

[29] L. V. e. a. Montiel, “Approximating Joint Probability Distributions Given,” *Decision Analysis,* 2013.

[30] SAFEXPLAIN, “D5.1 Case study stubbing and early assessment of case study porting,” 2024.

[31] 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.

[32] 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.

[33] 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.

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

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

[36] NVIDIA, “Xavier Series SoC Technical Reference Manual,” [Online]. Available: Xavier_TRM_DP09253002.pdf.

9 Annex 1 – Updated PMULib interface

In this annex updated 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_init()

pmu_result a78ae_pmu_init (void)

Initializes the library. Must be called before using any other library function. Must be called only once.

9.1.2 a78ae_pmu_configure()

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

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>.
mode	Contains the configuration mode. Should be the OR value of these three pairs of configurations: • PMU_MODE_PER_THREAD, PMU_MODE_PER_PROCESS: count per process or per thread. Default: per process • PMU_MODE_SCOPE_OWN_COUNT, PMU_MODE_SCOPE_ALL_PROCESSES: count own (thread or process) events, or all cores. Default: own. Note all requires root permission • PMU_MODE_CORE_ANY, PMU_MODE_CORE_FIXED: count events in any core or for a fixed core. Default: any core. Fixed core is configured in the lower bits, or'ing the core if to the configuration.

Returns

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

9.1.3 a78ae_pmu_counters_available()

unsigned a78ae_pmu_counters_available (void)

Number of counters available in the platform for simultaneous use.
Returns
The number of counters that can be used simultaneously in the platform.

9.1.4 a78ae_pmu_read_counters()

9.1.5

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.

9.1.6 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.

9.1.7 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.

9.1.8 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.

9.1.9 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.

9.1.10 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

The library provides definitions for event values and for configuration modes.

Macro	Value
PMU_MODE_PER_THREAD	0
PMU_MODE_PER_PROCESS	(1<<8)
PMU_MODE_SCOPE_OWN_COUNT	0
PMU_MODE_SCOPE_ALL_PROCESSES	(1<<9)
PMU_MODE_CORE_ANY	0
PMU_MODE_CORE_FIXED	(1<<10)

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 <stdio.h>
#include <stdlib.h>
#include "a78ae-pmu.h"

int main() {
	// The mask specifies if each counter will be used or not. 1 to set, 0 to reset.
	const unsigned int mask = 0b111111;
	// Event numbers in accordance with the A78-AE manual. You can also use the #defined
events in a78ae-pmu.h
	const unsigned int events[] = {0x3, 0x8, 0x11, 0x17, 0xA0, 0x400B };
	// Array which will save the values collected by the library
	int values[6];
	// First, initialize. Only once.
```

```
	a78ae_pmu_init();
	int core_id = 1;
	if(a78ae_pmu_configure(mask, events, PMU_MODE_PER_THREAD | PMU_MODE_SCOPE_ALL_PROCESSES|
PMU_MODE_CORE_FIXED | core_id) != 0) {
		exit(-1);
	}
	// Reset and start counting
	if (a78ae_pmu_reset_counters(mask) != A78AE_PMU_RESULT_OK) {
		exit(-1);
	}

	if (a78ae_pmu_start_global() != A78AE_PMU_RESULT_OK) {
		exit(-1);
	}
	volatile int tmp=0;
	for (volatile int i = 0; i< 100000000; i++)
		tmp += i;
	// Stop and read values
	if (a78ae_pmu_stop_global() != A78AE_PMU_RESULT_OK) {
		exit(-1);
	}
	if ( a78ae_pmu_read_counters(mask, values) != A78AE_PMU_RESULT_OK) {
		exit(-1);
	}
	printf("%d,%d,%d,%d,%d,%d\n", values[0],  values[1],  values[2],  values[3],  values[4],
values[5] );

}
```