Avionics use case description

**Table of Contents**

[1 The Avionics Use Case description 2](#_Toc129178091)

[1.1 Deployment 2](#_Toc129178092)

[1.2 Functional description of the FMS 2](#_Toc129178093)

[1.3 FMS Task decriptions and requirements 4](#_Toc129178094)

[2 Data collection with METrICS 7](#_Toc129178095)

[2.1 Performance measurement tools in a CPSoS context 7](#_Toc129178096)

[2.2 The METrICS framework 7](#_Toc129178097)

[2.2.1 METrICS architecture 7](#_Toc129178098)

[2.2.2 METrICS intrusiveness 9](#_Toc129178099)

# The Avionics Use Case description

## Deployment

The avionic use-case, is composed of a safety and security-critical application, the Flight Management System (**FMS)** from which we want to ensure the correct behaviour on the target iMx8 embedded hardware board depicted in Figure 1, as well as a sensor filtering application (**FMS FFT**)

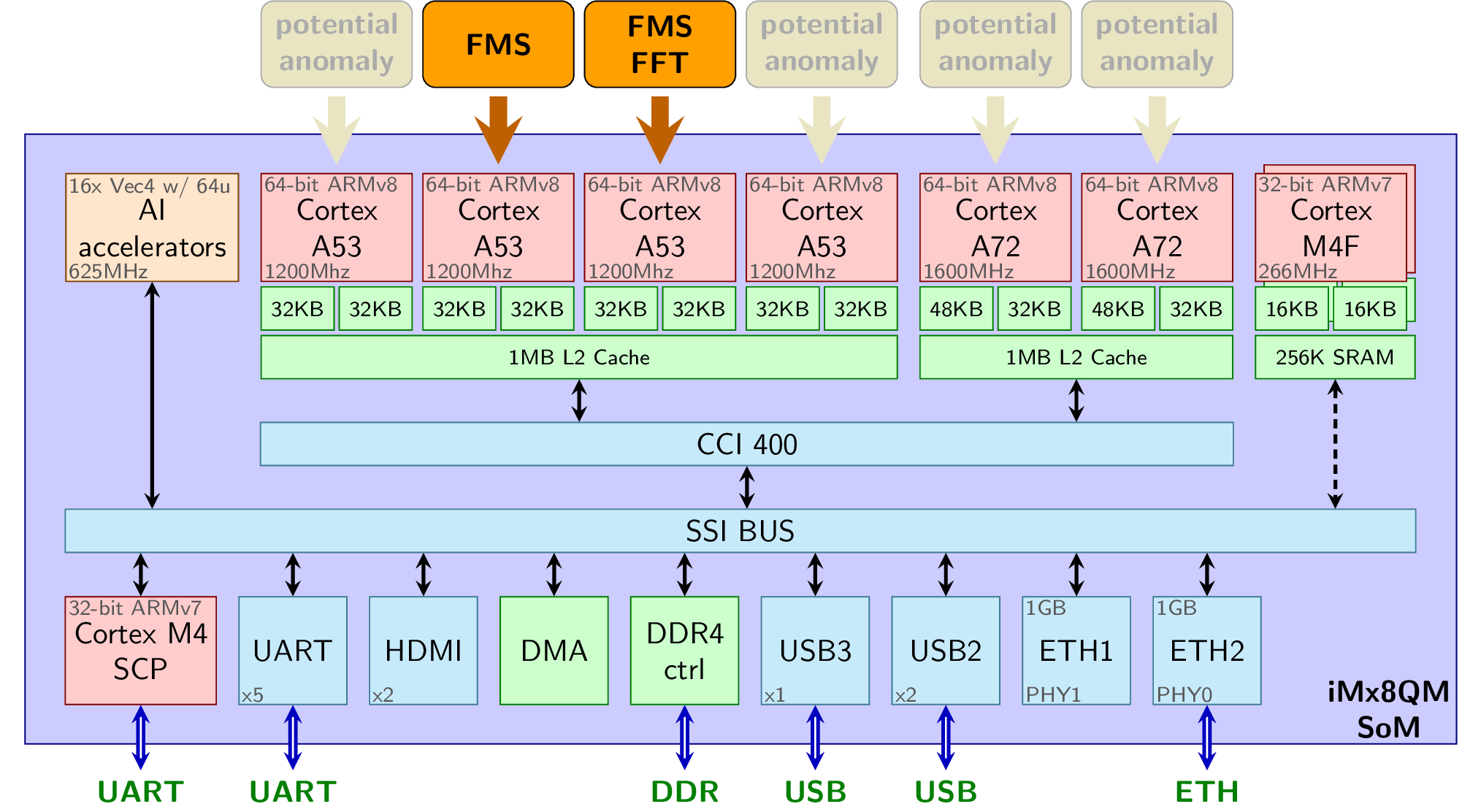


Figure 1 – ARMv8-based iMx8 hardware platorm

Safety requirements of the avionics domain introduce the **Design Assurance Level** (DAL) as the different safety levels defined by the avionics standards (DO-178, DO-254, DO-297). To ensure safety constraints, the industry relies on strict spatial partitioning and temporal partitioning, as well as the ability to compute the **Worst-Case Execution Time** (WCET) to respect real-time constraints.

A particular challenge for multi-core or heterogeneous architectures is to deal with **timing interference** of co-running independent applications sharing the same hardware resources (caches, interconnects, etc.). As a consequence, there is a tradeoff between performance and predictability. Hence, it is now required to identify and bound all the interference channels of the hardware architecture.

A second challenge, while running on top of an operating system such as Linux rather than a dedicater RTOS is to guarantee a partitioned runtime behaviour: While initially executing the flight management application, a large number of pre-emption were observed, as well as process migration from physical cores to other physical cores by the Linux scheduler.

To endure partitioning and prevent process/thread migration we dedicated a core to the FMS, a core to the FFT, and prevented the linux scheduler to use these cores for any other application, using the isocpus feature of the kernel. If effectively prevents the scheduler from staring nor migrating to the isolated cores.

The remaining cores are used to start attacks / anomalies as depicted in the Figure 1. Those anomalies are detailed with the datasets and as part of the avionic\_traces.docx documentation.

## Functional description of the FMS

Modern planes are equipped with Flight Management System (FMS). Its purpose is to provide the crew with centralised control for the aircraft navigation sensors, computer-based flight planning, fuel management, radio navigation management, and geographical situation information.

As depicted in Figure 2, the FMS is responsible for services that allow in-flight guidance of the plane. Throughout the pre-set flight plans, starting with the airport take-off and finishing with the airport landing, the FMS is responsible for plane localisation and trajectory computation. It is FMS that enables the plane to follow the flight plan, and reaction to pilot directives.

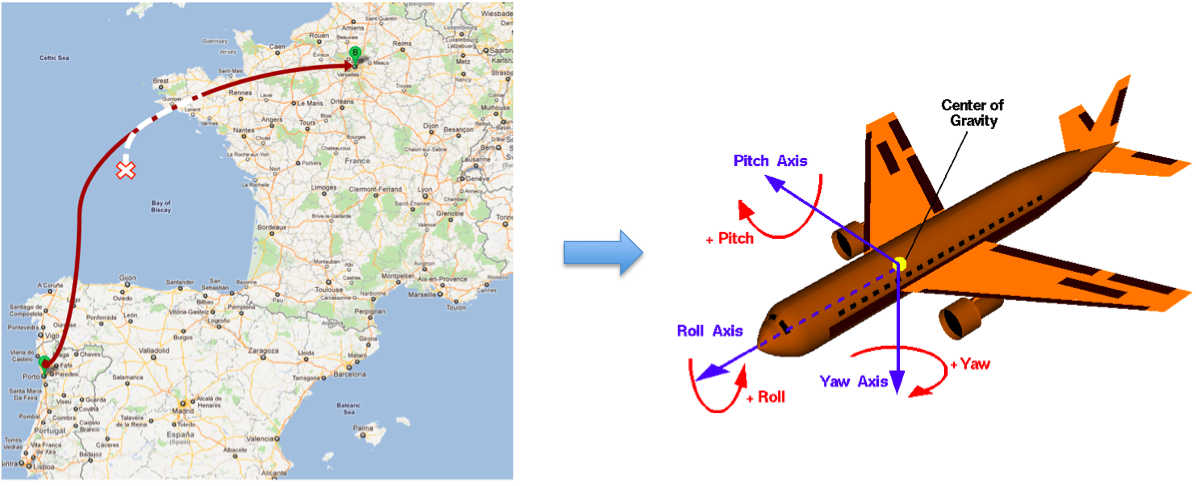


Figure 2. Flyance Flight Management System

The FMS application is constituted by 25 time-critical tasks that are regrouped in task groups (Sensors, Localisation, Flightplan, Trajectory, Nearest Airports and Guidance) as presented in Figure 3, that allow autonomous in-flight guidance of the plane, following a pre-set Flightplan from the take-off airport to the landing airport.

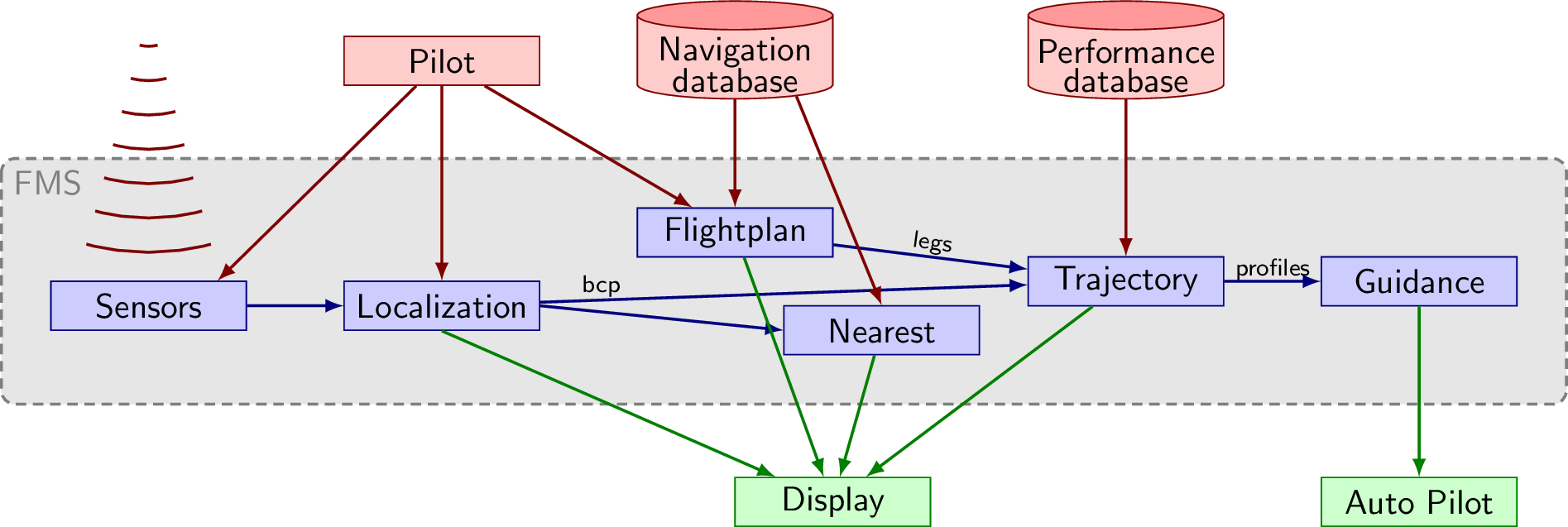


Figure 3. Flyance FMS - functional overview

The **Sensors task group** is in charge of generating all the localisation data from various sensors (Anemo-barometric sensors, Pure Inertia Reference System (IRS), Global Positioning System (GPS), Hybrid Inertia Reference System (HYB), Doppler sensor).

The **Localisation task group** is in charge of analysing outputs of sensors to generate the most probable position of the aircraft (BCP). This localisation data is composed of: Position (latitude, longitude, and altitude), Attitude (Pitch, Roll and Yaw angular rates), Velocity (Ground speed and Vertical Speed), Acceleration (lateral and longitudinal), and Wind-related data (speed and angle).

It must be noted that a single sensor may not provide the full localisation information. For instance, the Doppler sensor does not provide any position-related information, such as longitude and latitude. However, it provides very accurate velocity (speed-related) information. The role of the localisation task group is therefore to merge information from sensors with different trustworthiness levels.

The purpose of the **Nearest Airports task group** is to continually build a list of the nearest airports, during the flight. This information is useful if the pilot decides to land the plane impromptu. The tasks from this task group do not participate directly in flight management. The computed output is only sent to the display.

The **Flightplan task group** manages and processes modification requests on the pre-set flight plans. These are pre-set routes for plane guiding. Three different flight plans coexist concurrently in the system: 1) The active flight plan is the flight plan currently used to guide the aircraft; 2) The secondary flight plan is an alternative route toward the destination. It could consider an alternative landing runway at the destination airport, which has a significant impact on the target airport approach procedure; 3) The temporary flight plan is an intermediate flight plan, which allows the crew to enter a new flight plan and check for modification before applying. The flight plan task group is only composed of aperiodic tasks that correspond to the pilot’s modifications to the pre-set flight plans.

## FMS Task decriptions and requirements

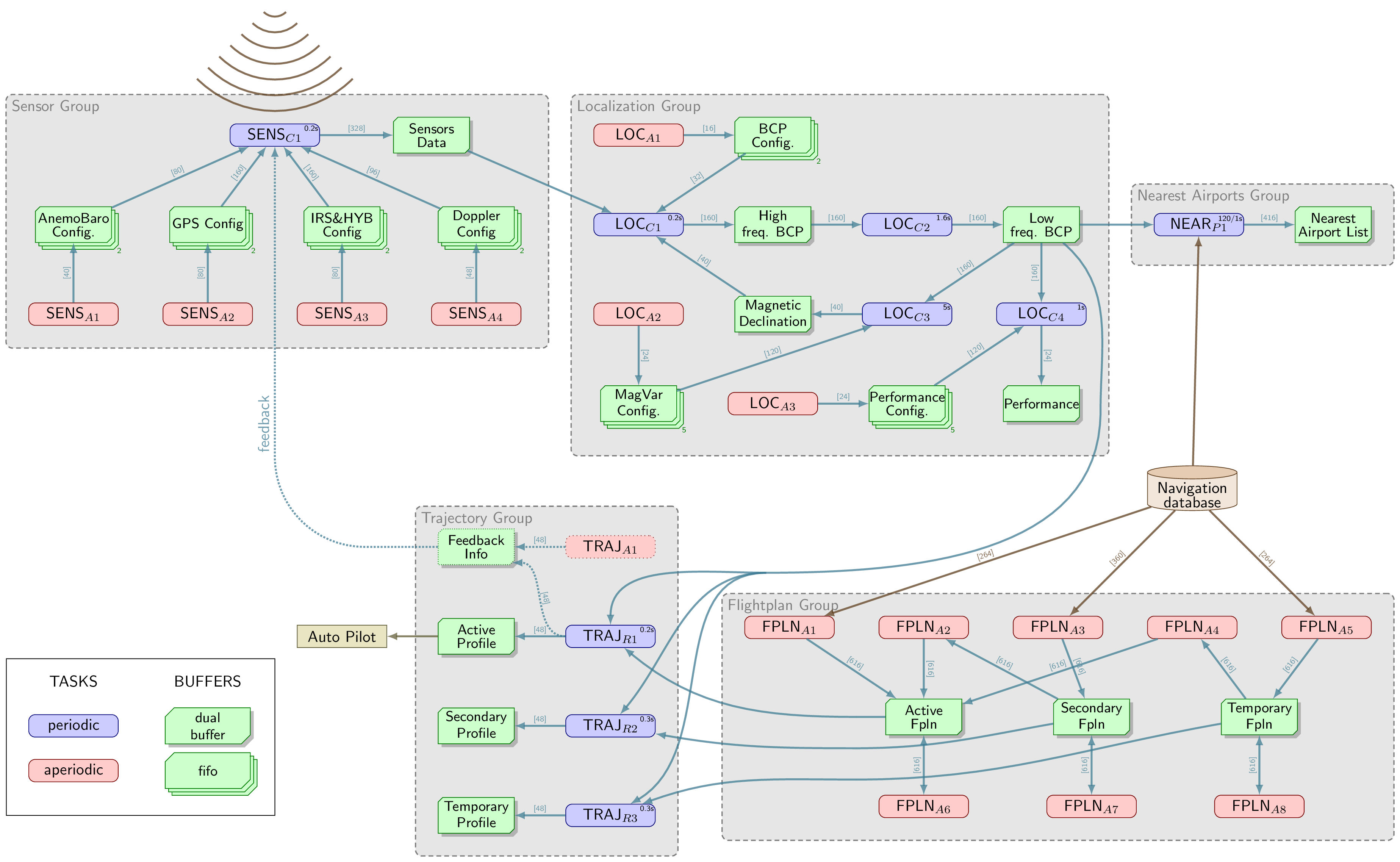
The Flyance FMS is a C++ application relying on object programming which overall taskgraph is presented in Figure 4. The data transmitted between the 20 periodic and aperiodic tasks are stored in dedicated, software-defined buffer structures, but the amount of data being transferred remains quite low (usually 3KB for inter-task data transfers).

Flyance also embeds a larger (10MB) navigation database, that is periodically accessed linearly to determine the list of the closest airports, computing loxodromic distances.

The most computation-intensive part of Flyance is its trajectory computation. However, the computation effort depends on the shape of the trajectory, with more computation during curves than during straight lines.

The tasks from the FMS application have stringent real-time requirements, most of them being periodic, meaning they do not start when their input data is ready as with regular dataflow applications, but are activated by the clock, even if the latest set of input data is not ready yet. Such hard-real time behaviour should not be altered by any HUMS / HIDS system ensuring the correct behaviour of the FMS.

The main periodic tasks composing the FMS application are the following:

* **SENSC1**: This periodic task is responsible for generating every sensor outputs to be passed to the Localisation task group. These data are self-estimated by the application instead of being received by real hardware sensors.
* **LOCC1**: Sensor management and high frequency BCP processing: This task is responsible for merging the valid sensor data, for performing data fixing on some of the sensor information, managing the localisation running mode based on the number of available sensor information, and for starting the computation of the best computed position performing a weighted mean of sensor information.
* 
* Figure 4 - FMS task graph
* **LOCC2**: Low frequency BCP processing: This task is in charge of the low frequency processing associated with the best-computed position computation.
* **LOCC3**: Magnetic declination computation: This task is in charge of computing the angular difference between magnetic north and true north (magnetic declination).
* **LOCC4**: Performance computation: This task is responsible for computing the actual aircraft navigation performance (ANP) according to dynamic aircraft performance models.
* **NEARP1**: Airport list computation. This task is responsible for the computation of the nearest airport list using the current localisation of the aircraft. Building this list implies querying the navigation database, and computing loxodromic distance to all the airports to figure out the closest ones. As a consequence, this task is both memory intensive and computation intensive.
* **TRAJR1**: Active trajectory. The active trajectory task performs the computation of the lateral and vertical profiles against the primary flight plan. This is a very critical computation-intensive task.
* **TRAJR2**: Secondary trajectory. The secondary trajectory task performs the computation of the lateral and vertical profiles against the secondary flight plan. This is a computation-intensive task.
* **TRAJR3**: Temporary trajectory. The temporary trajectory task performs the computation of the lateral and vertical profiles against the temporary flight plan. This is a computation-intensive task.

A set of pilot-triggered aperiodic tasks are also part of the application, but rarely triggered during the scenario. The Real-time requirements of the FMS application appear in Table 1.

Table 1 - FMS real-time requirements

|  |  |  |
| --- | --- | --- |
| **Task** | **Type** | **Period & Deadline** |
| SENSC1 | periodic | 200ms |
| LOCC1 | periodic | 200ms |
| LOCC2 | periodic | 5000ms |
| LOCC3 | periodic | 1600ms |
| LOCC4 | periodic | 1000ms |
| TRAJR1 | periodic | 200ms |
| TRAJR2 | periodic | 300ms |
| TRAJR3 | periodic | 300ms |
| NEARP1 | periodic | 1000ms |
| **Task** | **Type** | **Maximum activation** |
| SENSA1 | aperiodic | 2 times every 200ms |
| LOCA1 | aperiodic | 2 times every 200ms |
| LOCA2 | aperiodic | 5 times every 5000ms |
| LOCA3 | aperiodic | 5 times every 1000ms |

# Data collection with METrICS

We collected information on how the FMS software behave on the hardware with METrICS[[1]](#footnote-2), relying on performance monitoring counters. This collected information is then combined in an **aggregation** phase that generates traces of hardware events that will serve as learning material for the AI models, detecting anomalies with respect to the expected application behaviour.

## Performance measurement tools in a CPSoS context

For a long time, performance monitoring and profiling tools helped the HPC programmers with debugging their systems, optimizing their applications, or identifying bottlenecks. A wide variety of generic tools exists for non-time-critical systems[[2]](#footnote-3), such as gprof, valgrind, or atom. These tools rely on either OS features such as multi-threading, interrupts or timers, or either on pseudo-automatic code instrumentation to collect the required timing information.

However, in **real-time systems**, such features are either not available (with enforced static scheduling), restricted or prohibited due to their impacts on time determinism. This is especially true for safety critical software that is constrained by drastic limitations due to the safety standards[[3]](#footnote-4) [[4]](#footnote-5) [[5]](#footnote-6).

Performance Monitor Counters (**PMC**) are a way to collect both timing information as well as details on the hardware resource usage, making them suitable to be exploited in Health & Usage Monitoring Systems (HUMS).

The challenges can be summarized a. providing a way to 1) perform an accurate real-time runtime and resource usage measurement, 2) with a negligible impact on timing behaviour, 3) running outside of the operating system (avoiding system calls) to be able to profile both the OS and the running applications.

## The METrICS framework

**METrICS, a Measurement Environment for Multi-Core Time-Critical Systems**, a framework proposing accurate runtime and resource usage measurement with a negligible impact on timing behaviour of the critical application.

### METrICS architecture

METrICS consists of several core components appearing in green in Figure 5. On the left side, we present the components actually running on the target hardware board, and on the right side the METrICS server, running on a Linux host, and in charge of driving the experimental campaign to be run on the board and collect all the gathered profiling information.

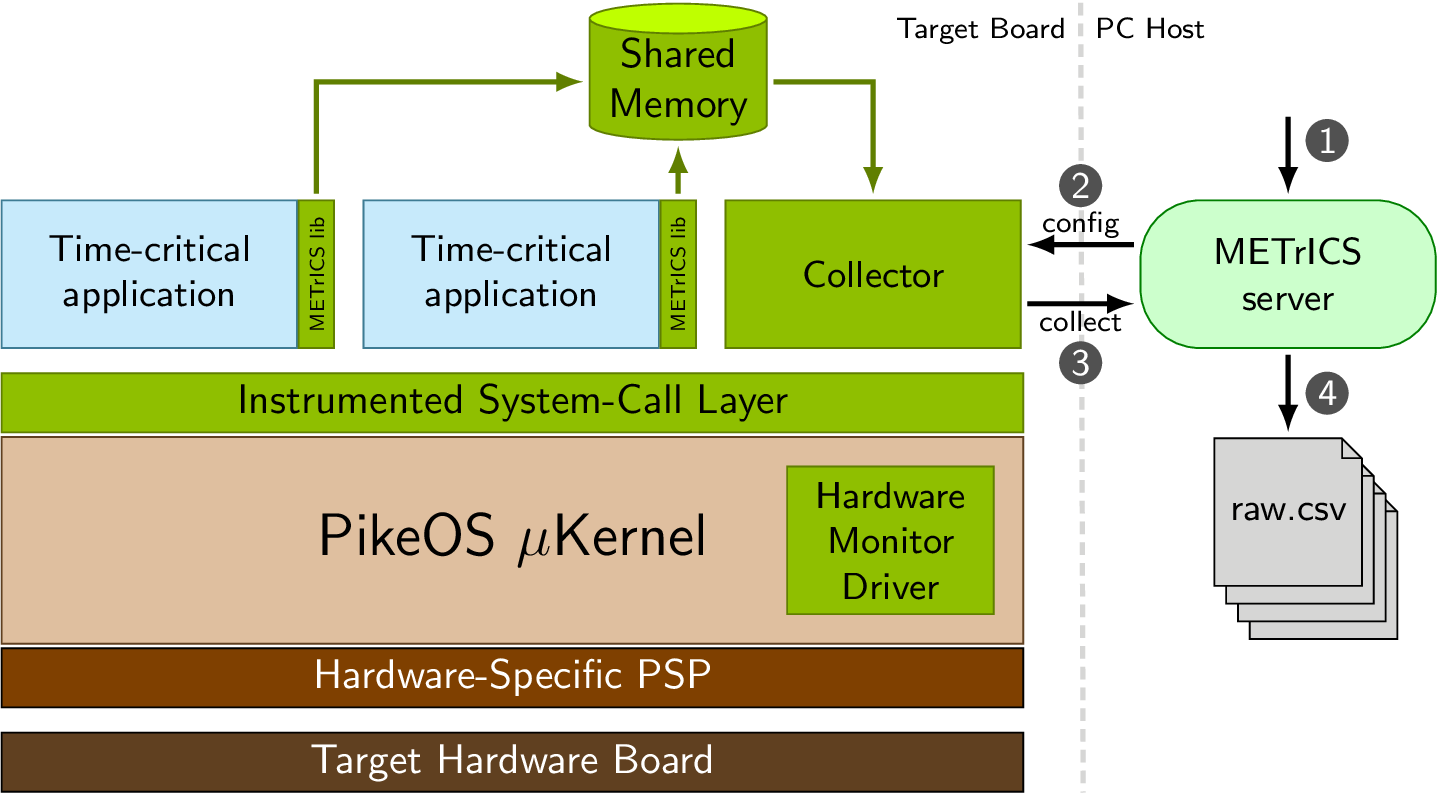


Figure 5: Architecture of the METrICS measurement tool

Software-related METrICS components include the **METrICS library** which allows insertion of measurement probes in the safety-critical software, the **syscall instrumentation** layer provides a way to automatically instrument each operating system call and the **collector process** that defines a shared memory space to collect measurements and configures specific measurement scenarios.

Hardware-related METrICS components include the **hardware monitor kernel driver** that provides the supervisor-level privilege necessary to access the hardware performance monitor counters (PMC) and, to some extent, the **collector process** that transfers the collected profiling information to an external Linux host.

* The **METrICS library** is meant to be linked with the target applications to provide them with an access to the measurement probes API, allowing the collection of time and resource access information.
* The **syscall instrumentation layer** provides a way to automatically instrument each APEX system calls for ARINC-653[[6]](#footnote-7) avionic applications.
* The **Hardware Monitor kernel driver** provides the supervisor-level privilege necessary to access to hardware performance monitor counters (PMC). Such counters[[7]](#footnote-8) allow us to count some hardware events, including the accesses to some shared hardware resource.
* The **collector partition** is in charge of 1) defining a shared memory space to collect measurements; 2) configuring specific measurement scenarios; 3) transferring the collected profiling information to the Linux host.
* Finally, the **METrICS server** running on the Linux host. It drives the experimental campaign and gather the collected profiling information.

### METrICS intrusiveness

A major challenge in profiling tools is its intrusiveness in the system it monitors. METrICS distinguish **execution time intrusiveness** and **code intrusiveness**. The former limits the accuracy of the measurement due to the monitoring overhead, whereas the latter requires an effort from the developer to instrument the code of the application, which could be an issue for legacy software. METrICS focuses into limiting **execution time intrusiveness**, to have a minimal impact on the timing interference phenomenon.

A time intrusiveness had been performed of a full METrICS probe consisting of: 1) retrieving the timing information thanks to the core-dedicated special registers; 2) retrieving the performance monitor counters, again through direct register access; 3) retrieving thread-specific information from the OS; and 4) storing the collected information into the shared memory. The results are presented in Figure 6.

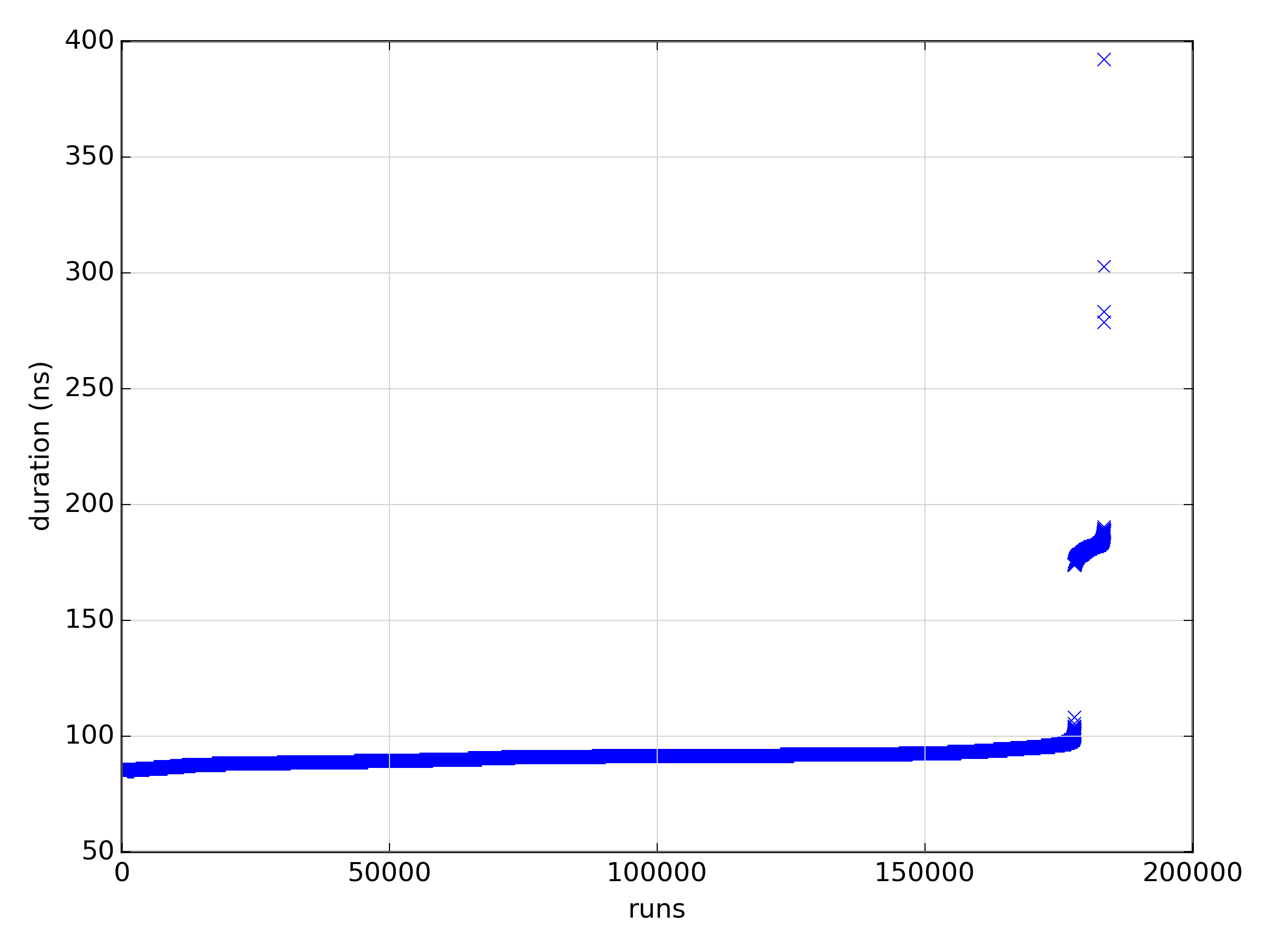


Figure 6: Completion time of a METrICS probe over 180000 runs

Over 180K runs, the probe time varies from 85ns up to 392ns. For 97% of the runs the overhead is below 110ns, and the overhead is above 191ns for only 0.002% of the cases. In comparison, the corresponding RTOS system call to only obtain current time (p4\_get\_time for PikeOS) requires 240ns, and it only get the current time and no PMC information. This is due to the fact that a system call involves at least two context switches, and possibly some privilege level changes. Therefore, the low intrusiveness of the overall METrICS probe makes it viable even for characterizing few micro-second long system calls of the OS.

## Trace format

The trace format and file structure organization of the collected data is described in details in avionic\_traces.docx and avionic\_traces.pptx

1. S. Girbal, J. Le Rhun and H. Saoud, “**METrICS: a Measurement Environment for Multi-Core Time Critical Systems**” in Embedded Real Time Software and Systems (ERTS’18), Toulouse France, 2018. [↑](#footnote-ref-2)
2. Survey of Software Monitoring and Profiling Tools. B Wun. 2006 [↑](#footnote-ref-3)
3. Functional safety and IEC 61508 – A basic guide. International Electrotechnical Commission (IEC), Geneva, Switzerland, Nov 2002 [↑](#footnote-ref-4)
4. ISO 26262: Road Vehicles – Functional Safety. International Organization for Standardization (ISO), 2011 [↑](#footnote-ref-5)
5. DO-297: Software, Electronic, Integrated Modular Avionics (IMA): Development Guidance and Certification. Radio Technical Commission for Aeronautics (RTCA) and EURopean Organisation for Civil Aviation Equipment (EUROCAE). 1992 [↑](#footnote-ref-6)
6. ARINC specification 653-2, “Avionics Application Software Standard Interface”. ARINC. December 2005 [↑](#footnote-ref-7)
7. The basics of performance-monitoring hardware. B. Sprunt. Micro, IEEE, 22(4):64–71, 2002 [↑](#footnote-ref-8)