ECSEL2017-2-783162

FitOptiVis

From the cloud to the edge - smart IntegraTion and OPtimisation Technologies for highly efficient Image and VIdeo processing Systems

Deliverable: D4.3 Monitoring, profiling, measuring and reconfiguration support for real time quality and resource management

Due date of deliverable: 31-05-2020

Actual submission date: 31-05-2020

Start date of Project: 01 June 2018 Duration: 36 months

Responsible WP4: Tampere University (of Technology)

Revision: draft

Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including the Commission Service	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (excluding the Commission Services)	

DOCUMENT INFO

Authors (alphabetical order)

Author	**Company**	**E-mail**
Francisco Barranco	UGR	fbarranco@ugr.es
Lubomír Bulej	CUNI	lubomir.bulej@mff.cuni.cz
Guillermo Amat	ITI	gamat@iti.es
Tiziana Fanni	UNISS	tfanni@uniss.it
Dip Goswami	TUE	d.goswami@tue.nl
Keijo Haataja	HURJA	keijo.haataja@hurja.fi
Pekka Jääskeläinen	TUT	pekka.jaaskelainen@tuni.fi
Jiří Kadlec	UTIA	kadlec@utia.cas.cz
Francesca Palumbo	UNISS	fpalumbo@uniss.it
Jukka Saarinen	NOKIA	jukka.saarinen@nokia.com
Raúl Santos de la Cámarra	HIB	rsantos@hi-iberia.es
Pablo Sánchez	UC	sanchez@teisa.unican.es
Carlo Sau	UNICA	carlo.sau@diee.unica.it
Shayan Tabatabaei Nikkhah	TUE	s.tabatabaei.nikkhah@tue.nl
Giacomo Valente	UNIVAQ	giacomo.valente@univaq.it
Luis Medina Valdés	7SOLS	luis.medina@sevensols.com

Document history

Version	**Date**	**Change**
v1.0	22-05-2020	Final deliverable for EU review.

Document data

Keywords	runtime platforms, runtime reconfiguration, runtime monitoring
Editor Address data	Name: Giacomo Valente Partner: UNIVAQ Email **giacomo.valente@univaq.it** Phone: **+393490685728**

Table of Contents

Table of Acronyms

Acronym	Meaning
V-PCC	video-based point cloud compression
TSN	Time Sensitive Networking
AR	Augmented reality
VR	Virtual Reality
MPSoC	Multi-Processor System on Chip
SEM IP	Soft Error Mitigation IP
VI/Os	Virtual Input / Output (s)
QRM	Quality and Resource Management
TSN	Time Sensitive Network
DSL	Domain Specific Language
RIE	Runtime reconfiguration Implementation of Embedded systems
gPTP	generalized Precision Time Protocol
BMCA	Best Master Clock Algorithm

1. **Executive Summary**

This report represents deliverable D4.3, one of the outcomes of Task 4.2 and Task 4.3 in the WP4 of the FitOptiVis project. The main objective of WP4 is to deal with the complexity of application runtime management while considering a diverse set of heterogeneous platform components and configurations.

Preliminary achievements of Tasks T4.2-T4.4 until the end of project year 2 are reported in this document. This includes developments in monitoring, profiling and measuring techniques and reconfigurability support.

This first iteration deliverable provides an overview of runtime reconfiguration and runtime monitoring mechanisms: these technologies span different levels of abstraction and serve to satisfy applications with diverse set of requirements. Specifically, the deliverable is split in two parts: in the first part, an abstract view of reconfiguration mechanisms, followed by specific instances, is reported. In the second part, an abstract view of the monitoring mechanisms, followed by specific instances, is reported. A second iteration of this deliverable will be done as a D4.4 due M30.

The content of this Deliverable contributes to MS6 (Second release of the virtual platform, components and methods. Partial demonstrators release.).

2. **Introduction**

Work package 4 addresses Objective 3 of the FitOptiVis project:

> Objective 3: Real-time multi-objective combinatorial optimisation; data and process distribution; run-time adaptation through virtualization; run-time quality and resource management; energy driven adaptations; workload (re-)distribution; support for run-time upgrades.

In WP4, the consortium develops techniques for run-time resource management within the system architecture template outlined in WP2. There are two key technology enablers for successful realization of runtime management; measurement/ monitoring, and reconfiguration mechanisms. These technologies are mainly investigated in the Tasks 4.2 and Task 4.3.

This deliverable reports on the outcomes of Task 4.2 and Task 4.3 in the first two years of the project. Task 4.2 focuses on providing mechanisms for the constant monitoring of dynamic phenomena to adapt to changing conditions at runtime. Task 4.3 focuses on developing mechanisms for the system reconfigurations at various abstraction levels, which is a key enabler in the quality and resource management framework. The deliverable is organized as follows.

In Chapter 3, we report of the reconfiguration mechanisms developed by various partners within the FitOptiVis project. Section 3.1 provides an overview of the three main categories of reconfiguration mechanisms being considered in the FitOptiVis project – adding/removing components, changing the component configuration, and changing the component compositions. It further shows an abstract view on how these mechanisms will be used the QRM framework.

Section 3.2 presents a specific instance of the reconfiguration mechanisms realized on the CompSOC platform. The reported work realized the first and the third type of reconfiguration mechanisms. Section 3.3 presents a Multi-Dataflow Composer (MDC) tool-based reconfiguration mechanism that allows for quality and budget adaptation in runtime and falls under second category of reconfiguration (i.e., changing the component configuration). Section 3.4 presents a reconfiguration mechanism for changing quality and budget (i.e., changing the component configuration) in runtime on Nvidia Jetson embedded devices. In Section 3.5, we describe a reconfiguration mechanism to adapt synchronization setting of a TSN and in essence, to reconfigure in terms of quality (i.e., quality of service). Finally, in Section 3.6, we present a RIE (of Embedded systems) based reconfiguration library which provides a general DSL framework for component implementation and reconfigurations.

In Chapter 4, the report provides the description of monitoring, profiling and measuring support developed within the FitOptiVis project, and also reports preliminary practical setups related to FitOptiVis use cases. To satisfy the diverse set of requirements found in FitOptiVis use cases, multiple concrete platforms are needed, each tailored to serve different types of requirements. In this regard, monitoring techniques can span at different levels, from cloud to edge; moreover, for each level a monitor can be software of hardware, albeit mainly requiring the synergy of both. In FitOptiVis, we aim to unify on the level of concepts, principles, and abstractions to find and extract commonalities found in different domains: for this reason, in Section 4.1, a reference platform for monitoring systems in FitOptiVis is reported; it allows to have a reference structure to describe the monitoring systems developed as part of the FitOptiVis platform.

In Section 4.2, three enabling solutions to implement monitoring actions are reported: FIVIS (Section 4.2.1), that is a common data storage, visualization, and analysis platform, an extension to DSL (Section 4.2.2) developed in WP2 to express monitoring requirements, and AIPHS (Section 4.2.3), a framework to implement custom monitoring systems for reconfigurable platforms.

In Section 4.3, instances of monitoring systems constituting the FitOptiVis platform are described, following the proposed reference platform. These instances make preliminary usage of enabling solutions, as highlighted in the report. In particular, some monitoring system instances target cloud-edge scenarios, such as 3D industrial inspection system (Section 4.3.1), Habit Tracking for elderly people (Section 4.3.3 and Section 4.3.10), Smart Grid (Section 4.3.3), Virtual and Augmented Reality (Section 4.3.6 and Section 4.3.7). Other monitoring system instances target edge scenarios, allowing also the communication of data with the cloud, such as monitor for object recognition and localization in space (Section 4.3.4 and Section 4.3.9) and monitor for FPGA-based coprocessors (Section 4.3.8 and Section 4.3.11). Finally, a monitor for the cloud-edge interconnection is proposed, targeting Time Sensitive Networks (Section 4.3.8). As integration, an Appendix reporting some example codes related to monitor instances is provided.

3. Runtime reconfiguration

3.1 Overview

We define *system configuration* as the set of components a system is composed of, their configurations (specified by their parameter set-points), and their compositions. In fact, such a composition is another component under the FitOptiVis component framework. Subsequently, we define *reconfiguration* as an action or a set of actions leading to a change(s) in system configuration. Therefore, based on the impact of actions, we categorize them into three classes as follows (see Figure 1):

- **Actions adding/removing components:** These actions add/removecomponents to/from the system. A component can be one of the following entities:
  - Application
  - Virtual Resource (VR)/Virtual Execution Platform (VEP)
  - Resource/Execution Platform (EP)
  - Deployed Application (application + VEP)
  - Hosted VEP (VEP + EP)

  These actions are triggered by users manually, automatically by Quality and Resource Management (QRM) components, or via custom mechanisms from the applications themselves.

  Some examples are:

  - Adding a stream in the Multi-Source Streaming use case which is done by a surgeon. This adds either an application or a deployed application to the system.
  - Hot plugging a hardware component adds a resource to the system.
  - Creating a VEP by QRM components (e.g., hypervisor) to deploy an application. Based on budget requirements of an application, hypervisor creates a VEP to deploy the application.
  - Spawning an OpenCL kernel by an application. When an OpenCL application calls a kernel, it adds another task to the system which requires certain budget (e.g., an Nvidia GPU with at least a certain compute power) to execute. Following the kernel call, QRM components create a VEP to deploy and execute the kernel.
  - Modify the component implementation in order to use efficiently a particular execution resource. A component could be implemented in a GPU with a particular algorithm but an FPGA implementation could require a different approach.

- **Actions changing component configurations:** Configurations of components are defined by set-points their parameters are set at. In general, a change in parameter set-points results in a change(s) to the following component properties:
  - Inputs/outputs
  - Required/provided budget
  - Qualities

These actions are triggered either by users, QRM components or applications.

Some examples are:

- Changing output resolution of a video stream on surgeon's demand. This affects required budget of the stream (e.g., a higher resolution requires more processing power, network bandwidth, and screen pixels to process, transmit, and display a stream).

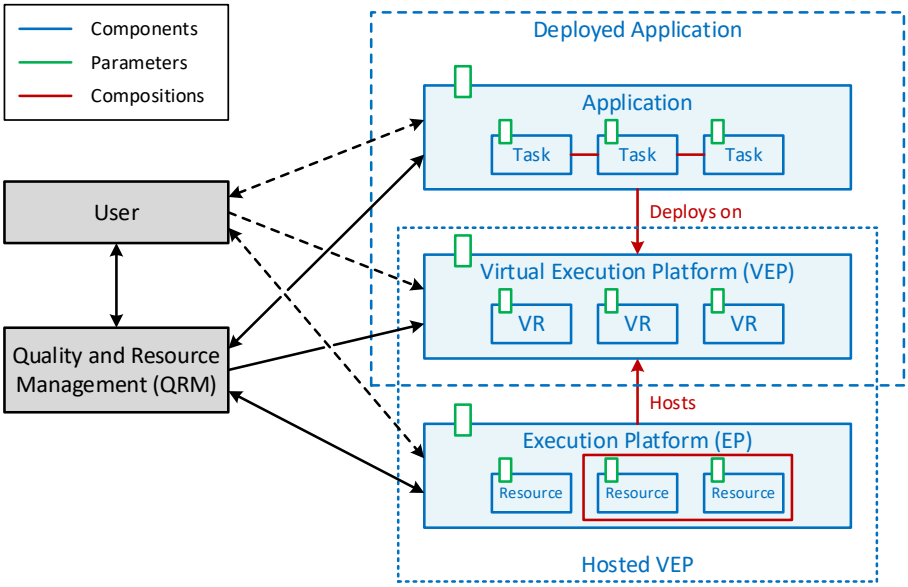

Figure 1: Overview of reconfiguration categories considered under FitOptiVis

Detecting workload transitions and asking for more/less budget. Based on input characteristics (e.g., framerate, resolution, number of streams, number of objects in a video), applications change their resource requirements, which is followed by reconfiguration of the VEP on which the application is deployed (done by QRM components).

- Reducing voltage/frequency of an overheated processor by QRM components.
- Changing topology of a DNN when a different recognition accuracy is needed. This needs to reconfigure the recognition task as well as the VEP on which it is deployed, since the new topology may need more or different resources (e.g., GPU instead of CPU) to execute within the same time.
  Switching profiles on Jetson TX2 platform. QRM components can switch performance modes of Jetson TX2 to optimize system power consumption.

- **Actions changing component compositions:** Compositions are vertical, horizontal, or free. Vertical compositions have to do with budget connections and are either deployments (application-to-VEP connections) or hosting (EP-to-VEP connections):
  - Deployments are established by i) finding application configurations whose required budgets are matched with a VEP's provided budget (i.e., budget matching), ii) selecting one of the matched configurations, and iii) installing the chosen configuration.
  - A hosting is binding (i.e., mapping) a VEP to physical resources in EP. This is also done by i) finding EP resources whose provided budgets are matched

with the VEP's required budget, ii) selecting the best mapping, and iii) binding the VEP on the chosen resources.

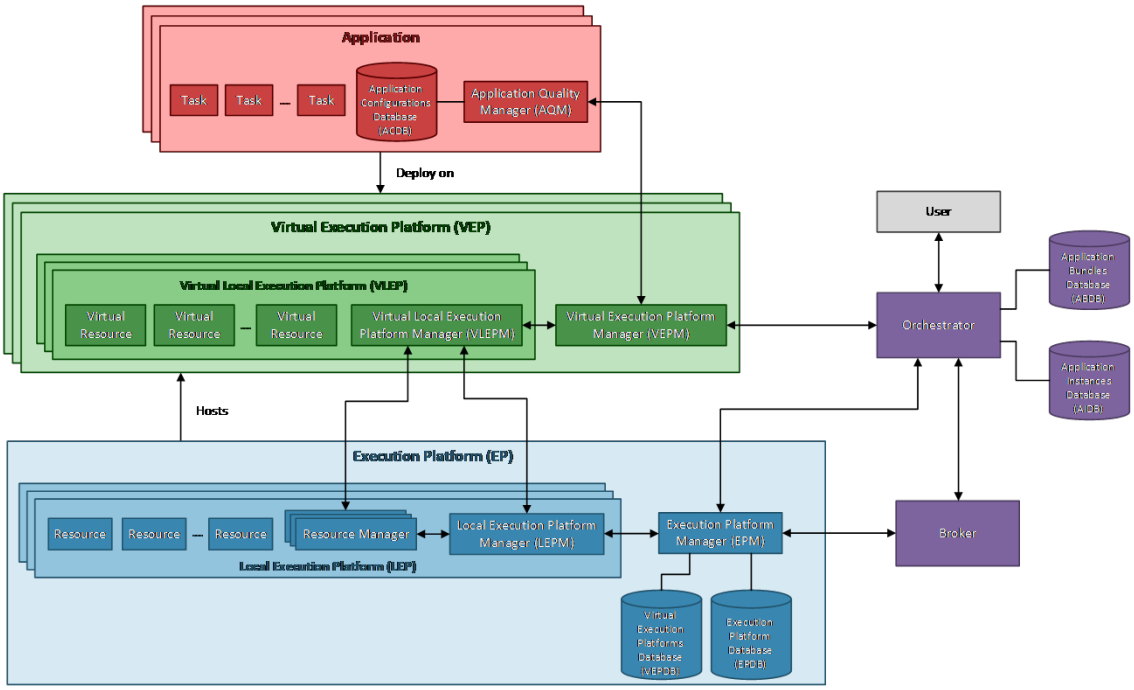

Figure 2: Block diagram of the proposed QRM architecture in CompSOC

Horizontal compositions connect inputs and outputs (e.g., connecting outputs of a task to inputs of another one). Free compositions connect neither input/outputs nor provided/required budgets. Rather, they are done to constrain the way a set of components can be composed to other components (e.g., a processor and a memory which are coupled together by an interconnect can be only used together).

The actions can establish, modify, or stop connections and are triggered by users QRM components, or applications.

Examples are:

- Manual mapping/unmapping VEPs done by users.
- Mappings/unmappings done by QRM components to optimize costs, resource utilization (e.g., load balancing), reliability, etc.
- Removing a displayed video stream from the screen.
- Reconfiguration of a crossbar changing the resources that are connected to it, which changes their free composition.
- Reconfiguration topology of a task graph, which changes horizontal compositions (e.g., changing the order of filters in an image processing application).

In the following, we describe a number of reconfiguration mechanisms developed using the above three categories of actions

3.2 Dynamic Reconfiguration in CompSOC

We employ the QRM architecture depicted in Figure 2 to perform dynamic reconfiguration in CompSOC platform. The architecture is designed in such a way that performing any type of reconfiguration action is possible. Details of this architecture are reported in D4.2. Initially, we have implemented an application deployment mechanism where the first and third category of actions are being used, which are adding/removing components and changing component compositions.

We assume the deployment command is issued by the end user. In other words, the user decides to execute/stop an application. The following steps are performed to deploy (i.e., execute) an application:

- User sends a command to the Orchestrator which contains the identifier of an application followed by its quality constraints:

  run_app <app_id> <quality_id> <minimum_quality> <quality_id> <minimum_quality> ...

- The Orchestrator queries the Application Bundles Database to obtain the application's bundle. If the bundle is not stored in the database, the deployment process stops. Bundles are stored in JSON files and the following pattern is used to make bundles:

```
{
    "id": "<component id>",
    "configurations":
    [
        {
            "id": "<configuration id>",
            "parameters":
            [
                {"<parameter_id>": <parameter_value>},
                …
            ],
            "qualities":
            [
                {"<quality_id>": <quality_value>},
                …
            ],
            "required_budget":
            [
                {
                    "cpu": {"cycles": <#cycles>, "period":<period in seconds>},
                    "local_memory": <size in bytes>,
                    "shared_memory_1": <size in bytes>,
                    "shared_memory_2": <size in bytes>
                },
                …
            ],
            "provided_budget":
            [
                {
                    "cpu": {"cycles": <#cycles>, "period":<period in seconds>,
"clock":<frequency in Hertz>},
                    "local_memory": <size in bytes>,
                    "shared_memory_1": <size in bytes>,
                    "shared_memory_2": <size in bytes>
```

```
                    },
                    ...
                ],
                "initial_state":
                [
                    {"src": "<hex file address>"},
                    ...
                ]
            },
            ...
        ]
    }
```

The pattern is consistent with FitOptiVis component model proposed in WP2. For our initial implementation, we have used a simple budget model to describe budgets required by applications and provided by the Execution Platform. The budgets are specialized for a CompSoC instance that is composed of multiple tiles (3 in this case) connected to each other using shared memory instances. Each tile has its own local instruction/data memory. The user can add a bundle to the database using the following command:

add_bundle <bundle_json_file>

If the bundle is found, the Orchestrator asks the Broker to find the best application configuration and its mapping by sending a command containing the application bundle and quality constraints. The best configuration is the one that meets the quality constraints and has the minimum deployment cost. Here we use resource usage as the deployment cost.

- To find the best application configuration and its mapping, the Broker needs to know the platform state. Accordingly, the Broker asks the Execution Platform Manager (EPM) to report its current state.
- Upon the Broker's command, the EPM queries the Execution Platform Database and sends Local Execution Platform bundles to the Broker. These bundles follow the same pattern explained before.
- After obtaining the application and LEP bundles, the Broker performs Pareto optimization to find the best application configuration and mapping. To do so, we employ the Pareto Calculator tool (http://www.es.ele.tue.nl/pareto/calc/). Note that during the optimization, the Broker must check if the budget required by the application matches the budget provided by the EP. The budget matching framework discussed in D4.2 is used for this purpose. Refer to D4.2 for the details.
- Once the optimization is done, the Broker sends the identifier of the best application configuration, the identifier of the LEP to deploy the application on, and the LEP's configuration identifier to the Orchestrator. Note that the platform may also have multiple configurations (e.g., low power mode, normal mode, high-performance mode) and the Broker selects the best one as well.
- Upon receiving the optimization results, the Orchestrator asks the EPM to create a Virtual Execution Platform (VEP) and load the application. Accordingly, it sends the bundle of the selected application configuration, the selected LEP id, and its configuration id to the EPM. Since each VEP is managed by a Virtual Execution Platform Manager (VEPM), the Orchestrator sends the VEPM's bundle alongside the aforementioned bundles and identifiers.

- The EPM first creates a VEP to load a VEPM. The creation of a VEP starts with reserving its required budget. This is done by updating the EPDB and VEPDB, so that the reserved budget is subtracted from the provided budget. After the reservation, the Orchestrator asks the LEPMs to allocate budgets and create a VEP. Each resource has a Resource Manager, which is part of the microkernel, that does budget allocation and virtual resource creation. LEPMs employ the API provided by the microkernel to create VEPs. Once a VEP for the VEPM is created, the EPM asks the LEPMs to load and run the VEPM. This is done by sending a hex file to the LEPMs.
- Once the VEPM is deployed, the EPM first reserves the budget required by the application, and then it sends the application configuration bundle to the VEPM.
- The VEPM follows the steps similar to what the EPM takes (to create a VEP and load an application) to create a VEP for the application. Once the VEP is created, the VEPM loads and starts the application.

The sequence diagram of application deployment is shown in Figure 3. For stopping and removing a running application, the “brokering” step is bypassed, and the Orchestrator directly asks the EPM to stop an application. The EPM first makes sure that the budgets are deallocated and VEPs are destroyed. Then, it releases the reserved budgets by updating the databases.

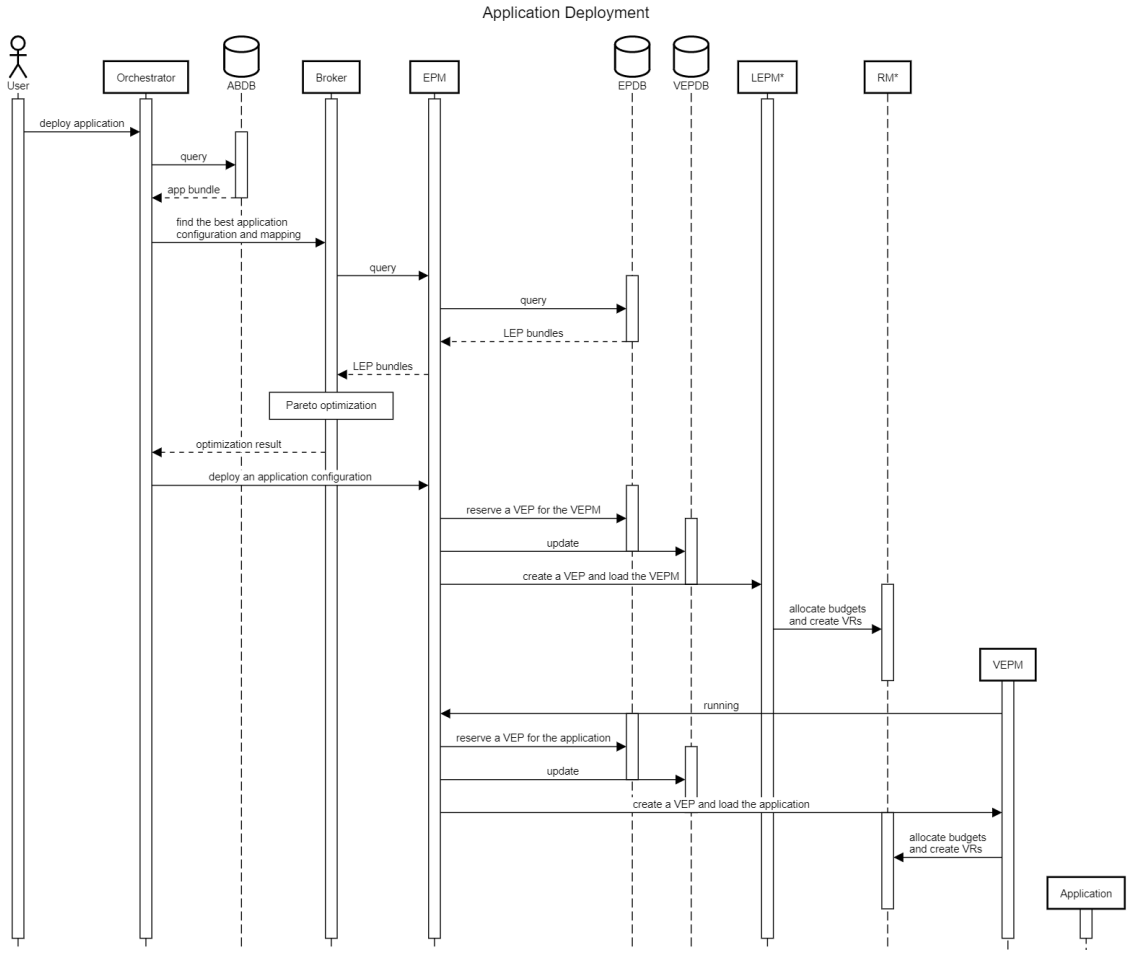

Figure 3 Sequence diagram of application deployment

3.3 Dynamic Reconfiguration using Multi-Dataflow Composer

The Multi-Dataflow Composer (MDC) tool, from UNISS and UNICA, starting from an input set of dataflow specifications, is able to generate Coarse-Grain Virtual Reconfigurable accelerators, able to execute the different functionalities specified with the dataflows. This belongs to the second type of action to change the configuration by modifying budget and quality of a component in the runtime. It does not only offer the support for the deployment of Xilinx compliant IPs, ready to be used in a processor-coprocessor system, but also the support for their management at run-time.

The Coarse-Grain Reconfiguration offered by MDC is virtual in the sense that resources are always available in the accelerator, and they are multiplexed in time according to the identifier (ID) of the selected operation, to be properly driven by the user. So that, resource occupancy efficiency will be not very high in the resulting reconfigurable accelerator (resources and connections are not all reused and reconfigured among different configuration), but reconfiguration can be achieved very quickly, ideally in a single clock cycle, due to the limited set of configuration points.

Reconfiguration, and in turn possible supported operations, are of two main types:

- Functional-oriented (see Figure 4) – the accelerator offers different functionalities (e.g. different image processing algorithms).

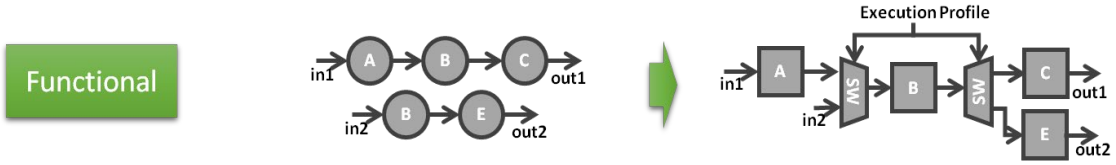

Figure 4: Functional-oriented reconfiguration

An example of functional-oriented reconfiguration where MDC has been successfully adopted is for neural signal processing [CAR13]. In this case, an algorithm for the denoising of a neural signal coming from the peripheral nervous system has been rolled and split into different sub-operations, these latter modeled as dataflow graphs. Then, a dynamic reconfigurable accelerator for such sub-operations has been assembled by MDC, resulting in substantial benefits in terms of resources and power consumption. As depicted in Figure 5, MDC dynamic reconfiguration (blue bar) allows saving about 40% area and power with respect to the corresponding non reconfigurable system where all the sub-operation graphs are instantiated in parallel (red bar). Moreover, it saves more than 86% of the same metrics if an unrolled implementation, where the denoising step

is not split into sub-operations (green bar).

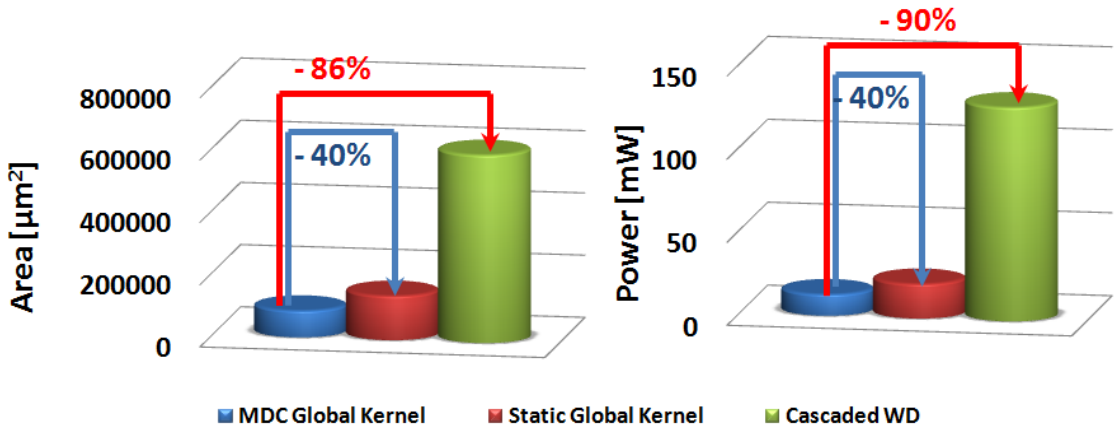

Figure 5: Area and power consumption histograms of the MDC generated denoiser (MDC Global Kernel) with respect to the corresponding non-reconfigurable denoiser (Static Global Kernel) and with respect to a unrolled atomic denoiser implementation (Cascaded WD).

- Working point-oriented (see Figure 6) – the accelerator is able to execute the same functionality but with different trade-offs in terms of non-functional metrics (e.g. different image quality vs. power consumption profiles in encoding/decoding algorithms).

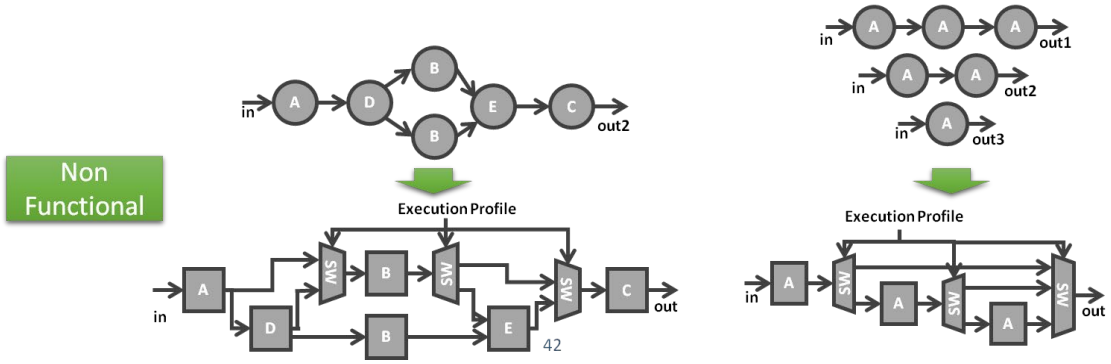

Figure 6: Working point-oriented reconfiguration

An example of working point-oriented reconfiguration achieved through MDC is in the field of video coding [SAU17]. In this case, the fractional pixel interpolation filters adopted for motion estimation/compensation in the HEVC codec have been considered. In particular, approximate computing has been applied at the algorithm level to derive approximate filters by using a reduced number of taps, with respect to legacy values. For instance, considering the luma color component, two approximate filters have been derived by adopting 5 and 3 taps with respect to the legacy 8/7 ones. These filters have been modelled as dataflow graphs and processed by MDC in order to provide a reconfigurable filter able to switch among the different versions, from legacy to approximated. As depicted in Figure 7, the obtained reconfigurable filter has different working points offering a different trade-off between quality and energy consumption. In particular, in terms of energy consumption, it is possible to have up to 27% savings with

respect to the standalone legacy implementation, by employing only 3 taps instead of 8.

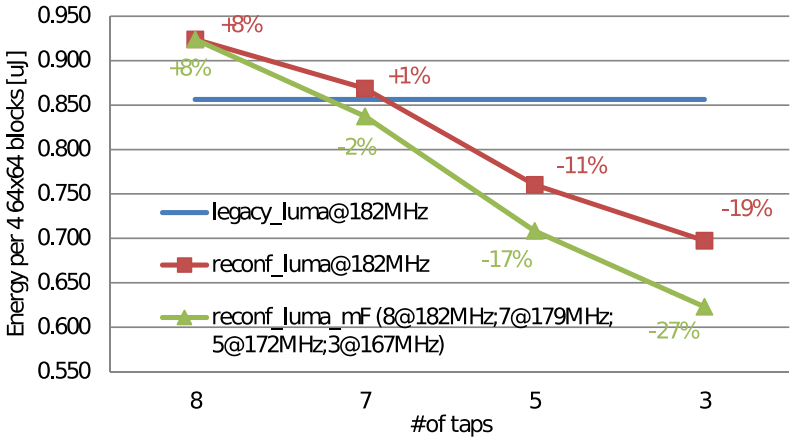

Figure 7 Reconfigurable HEVC interpolation filter: the supported working points provide different energy versus quality (Inv. proportional to # of taps) trade-off.

3.4 Reconfiguration in Nvidia Jetson embedded devices

UC3 (Habit tracking) and UC9 (Smart grid) use the NVidia Jetson embedded devices Jetson Xavier and Jetson TX2. These Nvidia® devices support runtime adaptation for example, varying the operating frequency of the GPU and ARM-CPUs, or the number of active GPU cores. This adaptation enables energy consumption vs time performance trade-offs, also in terms of hardware requirements.

In this case, the reconfiguration takes place by selecting different alternatives pre-defined for some components or modifying provided budgets. For example, the reduction or increase in the budget provided by Jetson platforms to the application components that require them. This reconfiguration is activated by the system after monitoring power consumption over a period of time and when some components require a higher or lower quantity of resources provided by the platform in terms of compute capability.

Habit tracking (UC3)

Regarding UC3, we do reconfiguration to robustify the confidence for an inferred critical action. In order to improve the confidence, apart from using the RGB-input video stream, an Optical Flow stream is also analysed. Optical Flow comprises information about speed and angle of the movement of pixels between frames. This mid-level cue is fed to a neural network that is capable to recognize actions from this spatio-temporal flow. Thus, the reconfiguration follows the dataflow represented in Figure 8.

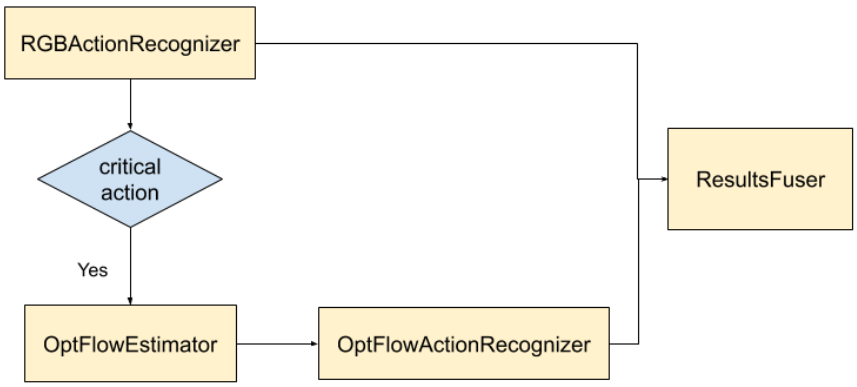

Figure 8: Diagram of tasks involved in action recognition

For instance, if we detect that the active model cannot distinguish between two actions for a period of time or we want to make sure that a person has fallen down and now is lying on the floor, we use an alternative with a more complex model that takes the RGB-stream and the OpticalFlow stream and infers a new label based on the results of a neural model that takes both.

Estimating optical flow from a video stream is a resource-intensive task even when computed by the GPU. For this reason, we have planned to compute this using **pocl-remote** from TUT, offloading the optical flow processing to the cloud server. Thanks to pocl-remote, the resources of the cloud are seen as a local resource for the software application. In other words, this framework allows us to do calculations on an external GPU over the network in a transparent way using OpenCL. Considering that the Jetson devices have a limited amount of hardware resources (Jetson Xavier allocates 512 CUDA cores) and the GPU in the Jetson devices will be used to do the inference of the neural network models. The pocl-remote framework offers us a way to accelerate computation and reduce power consumption on the embedded devices. Our external GPU in the cloud is an RTX2080Ti with 4096 CUDA cores.

After some initial tests of using pocl-remote between the Jetson devices and the PC with the powerful GPU, we have observed an improvement in the performance of **50%** in comparison of using the Jetson GPU when it is free and we are not doing inferences. In addition, we can see a significant improvement if we do the estimation through pocl-remote compared to using the ARM-CPUs. This is shown in Figure 9. Processing time improvement is about 12x estimating Optical Flow through pocl-remote compared to using the Jetson TX2 edge platform.

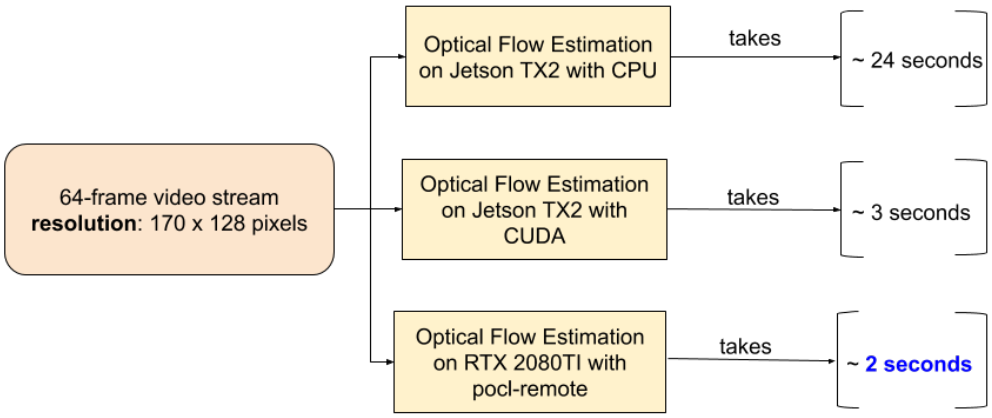

Figure 9: Compare time required to estimate Optical Flow TV-L1

In summary, pocl-remote speeds up computation of the Optical Flow, decreasing the amount of resources used on the Jetson edge device and consequently the energy consumption at the edge.

Specific alternatives for the components and other reconfiguration actions are also summarized in Deliverable 5.2.

Smart Grid

Regarding UC9 (Smart-Grid), the video surveillance system resulting from our collaboration is dynamically adaptable. The workflow of the system changes taking into account the events that occur in the monitored facility and the logic of the program itself. So, for example, when the HumanDetector sub-component does not detect any target in the scene, the rest of the tasks included in the other sub-components of the system are not executed (green area in Figure 10). However, if it detects one or more targets, the tracking of these targets is carried out by the Tracker sub-component (blue area in Figure 10). Also, the execution of this other sub-component can trigger the generation of an alarm through the AlarmGenerator sub-component (red area in Figure 10). More details are given in Deliverable 5.2.

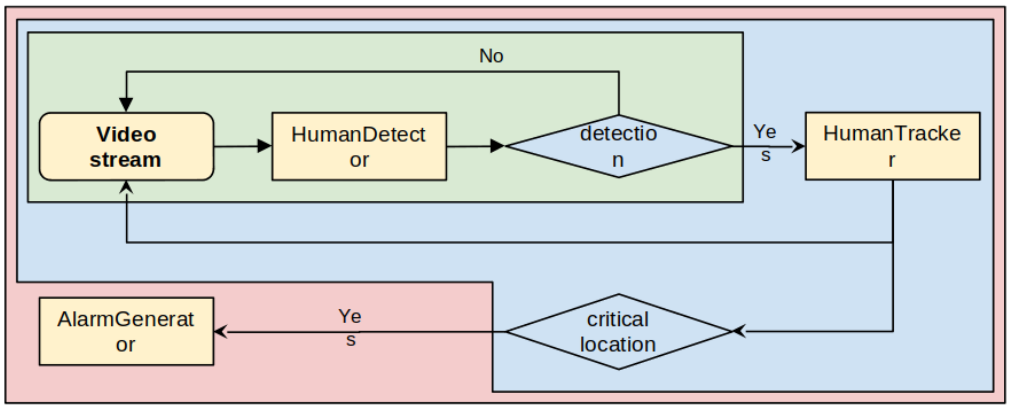

Figure 10: Smart-Grid surveillance system component composition

3.5 Reconfiguration of Time Sensitive Network (TSN)

Accurate and reliable time synchronization is key for guaranteeing deterministic Quality of Service in the presence of mixed critical traffics. Time synchronization is required not only in the end processing nodes, but also on the time-aware traffic shapers present on the forwarding nodes, to provide deterministic delivery. Early fault detection and fast switchover is required to minimize determinism violations.

The generalized Precision Time Protocol (gPTP) defined on the IEEE 802.1AS defines protocol mechanisms to provide continuous monitoring of the synchronization status and overcome network eventualities, such link or node failures, including the grandmaster or network time reference.

The monitoring mechanisms are described on Section 4.3.8. This section will discuss how these monitors are applied to adapt the behaviour of time-aware stations, starting from each individual active interface (port role). These mechanisms conform with the so-called Best Master Clock Algorithm (BMCA).

The Best Master Clock Algorithm

The BMCA determines the grandmaster (network time reference), as well as the behaviour of each time-aware station to spread the synchronization information along the network. As the breakup of this chain may imply determinism violations, the IEEE 802.1AS states that every TSN station must execute the BMCA periodically and be ready to replace the current grandmaster and provide synchronization to their peers in case of failure.

To this end, each time-aware station periodically compares itself with the grandmasters elected by their peers. The grandmaster eligibility is evaluated according to six attributes, namely the best master selection information, in the sequence listed on the Table 1.

Table 1: Best master selection information.

Attribute name	Short description
priority1	Most-significant priority declaration in the execution of the best master clock algorithm. Lowest values take precedence. Although all values are allowed, 0 and 255 are forbidden under normal operation
ClockClass	Traceability of the synchronized time (timing from GPS, Atomic clock, internal oscillator).

ClockAccuracy	Expected time accuracy
offsetScaledLogVariance	Representation of an estimate of the PTP variance
priority2	Least-significant priority declaration in the execution of the best master clock algorithm
Clock Identity	The clock Identity is an 8-octet stream providing unique identification of the current node.

These attributes can be classified as administrative or descriptive. Whereas descriptive parameters provide information regarding the precision capability, the administrative ones (priority1 and priority2) can be arbitrarily set and allow the control of BMCA for a given network

The attributes of the elected grandmaster are propagated to the remote peers by means of Announce messages. Besides, each node participating on the election registers itself on the pathTrace field, conforming the time-synchronization spanning tree, which is the route followed by the synchronization information (see Figure 11).

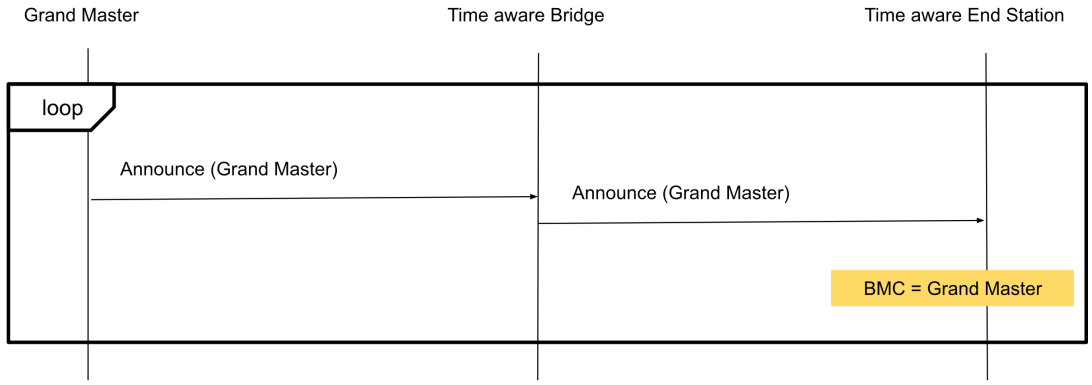

Figure 11: Announce message exchange and Best Master Clock election

Besides, the Announce message propagation indicates the availability of the time-aware nodes present on the time-synchronization spanning tree. The Announce message is discarded after a given timeout (typically three times the configured announce message periodicity). The announce messages is not sent and not considered on reception if the propagation delay measurement is not completed successfully (i.e. asCapable flag is not true). Consequently, a link or node failure along the time-synchronization spanning tree results on its reconfiguration and eventually, on the election of a new grandmaster.

The BMCA also should configure the port role on each active interface according to the resulted time-spanning tree. IEEE 802.1AS defines the roles reported in Table 2.

Table 2: Roles defined by IEEE 802.1AS

Role	Explanation
Master	Active interface sourcing synchronization information.
Slave	Active interface receiving and processing synchronization information
Passive	Active interface receiving synchronization information and backing the slave interface
Disabled	Non active interface or not available (PHY layer reporting disconnected status)

Note that there is only one Slave Port on each interface, which corresponds to the interface with the shortest time-synchronization spanning tree. The Passive ports have longer spanning trees and back the Slave Port in case of network failure. This way, gPTP takes advantage of redundant network topologies. Master ports are present in the grandmaster and intermediate bridges and are responsible for spreading the synchronization information to attached peers.

3.6 RIE-based reconfiguration method

RIE (Runtime reconfiguration Implementation of Embedded systems) is a component-based implementation methodology. It allows creating C++ components from an extension of the DSL language (SDSL, Service-oriented DSL) that was proposed on WP2. A generator creates a C++ implementation template in which components are implemented as classes that make use of the RIE library. RIE provides run-time reconfiguration capabilities that allow managing component implementations and configurations at runtime. Reconfiguration decisions are taken depending on some qualities that are traced at runtime.

In the RIE-based methodology, a component may have several set points that define different implementations and configuration parameters. Every component is modelled with a C++ base class that define the component interfaces. All component implementations derive from this base class and share the same interfaces (provided and required services) and configuration parameters. The RIE library includes several methods that allows access at runtime to the components and modify their set points.

Each component may have different alternatives or implementations that can be exchanged at runtime. Each of these implementations represent a different component mapping of the application into a physical platform, this vertical composition may be changed dynamically in response to a monitoring result.

Figure 12. RIE Methodology

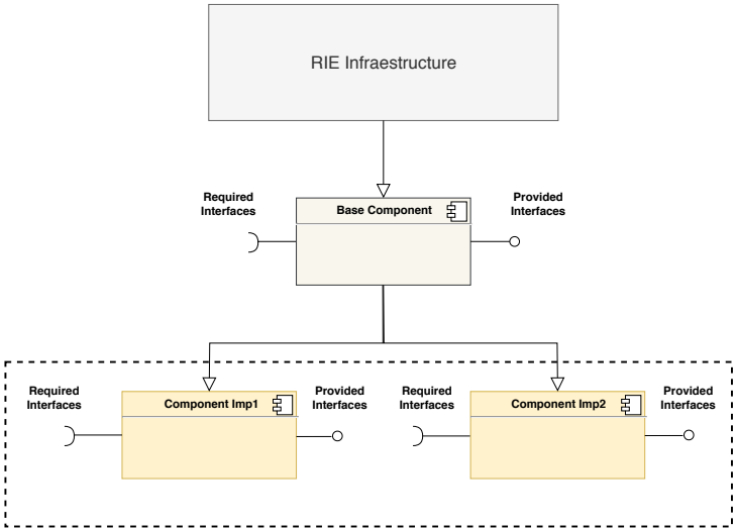

Figure 13: RIE implementation strategy

In Figure 13, the RIE implementation strategy is shown. The RIE library provides the main infrastructure. For example, the methods that allow modifying the component allocation or accessing the component parameters are defined in this library. The basic components are derived from the RIE library classes. A basic component defines the interfaces and common parameters and qualities of a component. All the implementations of the basic component will share the same interfaces and common parameters/qualities. All the implementation are derived from this basic component. These implementations could provide different algorithms or specific implementations for a particular hardware resource. They could also have specific parameters or qualities. For example, a Camera component could be a basic component. This component could have several implementations (USB camera, Hardware camera, etc) that are derived from the base components and share the same interfaces. The base component and its implementations share a configuration list, a JSON string with all recognized component configuration. This list can be modified at runtime and new component implementations could be included. An example of camera configuration is:

```
Camera_configuration = [
       { "s0", {"RIE_Impl":"Camera","fps":30," RGB_W ":640, …
       { "s1", {"RIE_Impl":"CameraUSB","fps":30," RGB_W ":640, …
       { "s2" ,{"RIE_Impl":"CameraHW",…
```

The list defines three set points (s0, s1 and s2). At runtime, the RIE library provides a method (reconfigure) that allows modifying the component set point and parameters. Some component parameters are defined in the RIE methodology. For example, the “RIE_Impl” parameter defines the derived class that will implement the basic component in a particular set point. The RIE reconfiguration process includes four stages:

1. Initialisation. The RIE library provides a method (reconfiguration) that allows initializing the reconfiguration process. This method is executed by the runtime

manager that could be a system component (self-reconfiguration). The runtime manager selects the new system configuration.
2. Place the system in a safe state. The RIE infrastructure accesses all the system components and executes a component specific stop instruction. This instruction forces the component to change its state. This process could require some time because some components could need time to finish the current task. For this reason, the RIE library checks the component state until all the components are in a safe state.
3. Component reconfiguration. Taking into account the system hierarchy, the RIE library modifies the parameters and implementations of the system components.
4. System resume. After system reconfiguration, the RIE library sends a component specific command (resume) in order to re-initialize the component operation.

RIE also supports edge component implementations. In this case, the component configuration has to include three parameters:

- RIE_Impl: This parameter defines the local implementation of the remote component. This implementation is a wrapper that includes the local infrastructure that is required to execute remote procedures. The current RIE version supports a socket-based local infrastructure although the new version will provide a grpc (https://grpc.io/) services.
- RIE_RemoteSetPoint: Set point of the remote component.
- RIE_URL: The component configuration does not define the remote server that will provide the component services. The configuration only includes a label (RIE_URL) that has to be defined at runtime. During reconfiguration, the RIE library uses this label to find the remote server that will provide the service.

In the case of remote/edge components, the component reconfiguration process requires seven additional steps:

1. When the local component is in a safe state, the local implementation is modified. The new local component implementation is an instance of a wrapper component that is used by all possible remote/edge implementations.
2. The RIE library read the RIE_URL parameter and requires from the Component Implementation Server (CIS) all the information related with the RIE_URL parameter. In the RIE methodology, the CIS server has a similar role to a DNS server. The CIS server defines the URL of the server that will provide the component services as well as the component name in the remote server.
3. The local RIE infrastructure requires to the remote RIE infrastructure information about the remote component.
4. The local provided services are connected to the remote component services.
5. The remote required services are connected to the local component required services.
6. The remote component is reconfigured to the set point that is defined in the RIE_RemoteSetPoint parameter.
7. The remote component is included in the component list of the local system.

An SDSL example is shown below (see Figure 14 and Figure 15), where a Camera component has 2 different set-points with different implementations and different configuration parameters (fps, latency, …) which will be monitored to analyse if reconfiguration must be taken.

```
component Camera{
        alternatives{
                component cameraHW camhw;
                 component cameraUSB camusb;

                quality EstimatedFps:integer;
                quality Latency:integer;

                //Association between qualities and monitor parameters
                EstimatedFps==EstimatedFps.frameRate.mon1;
                Latency==Latency.maxLatency.mon1;
        }
}
```

Figure 14: Camera component

```
component cameraUSB{
        implementation of Camera;

        Fps==30;

        requires cores reqcores{
        //Constraints
        };
        requires memorycon reqmem{
        //Constraints
        };
```

```
component cameraHW{
        implementation of Camera;

        Fps==45;

        requires cores reqcores{
        //Constraints
        };
        requires memorycon reqmem{
        //Constraints
        };
```

Figure 15: Different set-points for the Camera component

In this example they can be observed component parameters as well as qualities (associated to monitor parameters).

A component can also have different functional implementations, so a component can be replaced by another one sharing common interfaces. An SDSL example description is shown below, where a component ImgProc has 2 different functional implementation Rgb2gray and Sobeledge, they have the same interfaces but different functionality, so one can be replaced by the other one.

```
component ImgProc {

        alternatives {
                component Rgb2gray rgb2gray;
                component Sobeledge sobel;

                providesinterface VideoInterface ip1;
                requiresinterface VideoInterface ir1;
        }

}

component Rgb2gray {
        providesinterface VideoInterface ip1;
        requiresinterface VideoInterface ir1;
}

component Sobeledge {
        providesinterface VideoInterface ip1;
        requiresinterface VideoInterface ir1;
}
```

4. Runtime Monitoring, Profiling and Measuring

This chapter describes the monitoring, profiling and measuring support developed within the FitOptiVis project, and also reports preliminary practical setups related to FitOptiVis use cases. First, a reference platform for monitoring systems in FitOptiVis is proposed: this platform allows to highlight how specific monitoring requirements are going to be addressed from partners. Then, some enabling solutions to perform monitoring in FitOptiVis are described: they came out after the analysis of requirements coming from use-case providers, WP3 (monitor to refine design-time models) and WP4 (monitor for runtime management), and their goal is to support on the development of monitoring systems. Finally, instances of monitoring systems constituting the FitOptiVis platform are described, following the proposed reference platform.

4.1 Reference Platform for monitoring

With the goal to properly identify the monitoring techniques developed within the FitOptiVis project, a reference architecture of a cloud-edge computing system has been considered, shown in Figure 16. In this type of architectures, monitoring techniques can span at different levels, from cloud to edge; moreover, for each level a monitor can be software of hardware, albeit mainly requiring the synergy of both. This means that the development of monitoring systems in FitOptiVis scenarios, and corresponding system-level services, involves several trade-offs from architectural point of view.

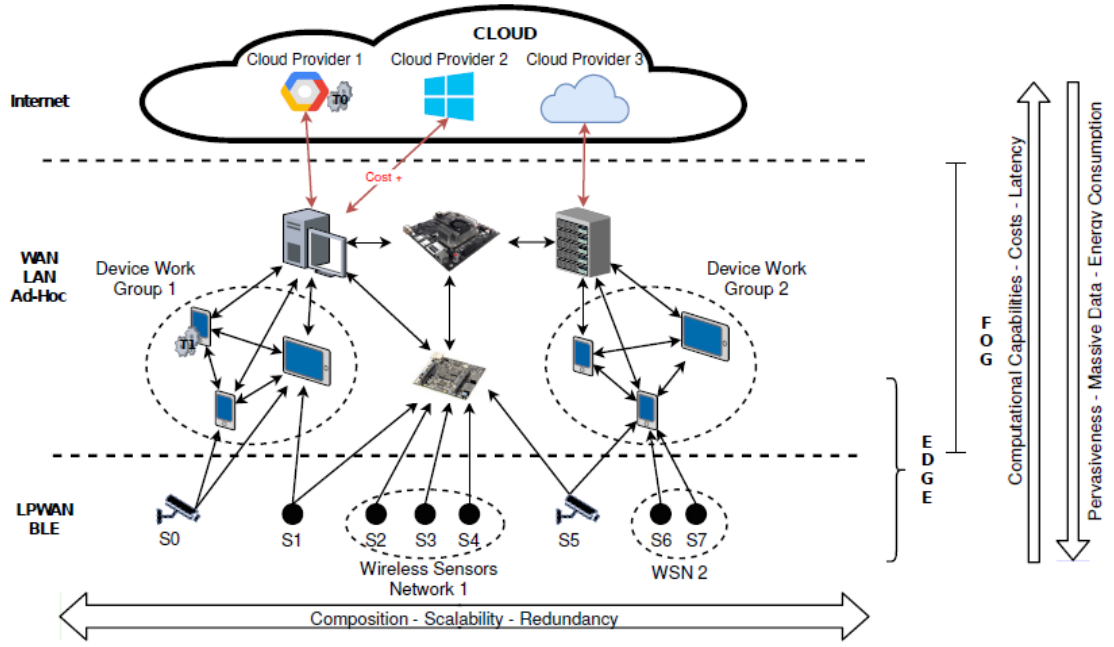

Figure 16: A reference architecture for Cloud-Edge computing systems [ZAN18]

For these reasons, a reference model of a monitoring action has been developed, with the sake of clarity. It is reported in Figure 17 and shows the actors involved in a monitoring action. Independently on how many layers are involved, some components can be always identified: a *Unit Under Monitoring* (UUM), a *monitoring infrastructure* that extracts raw information from the UUM by means of hardware/software mechanisms,

and a *Data Storage, Analytics and Visualization* part that organizes, filters and parses the raw information to obtain the monitoring information.

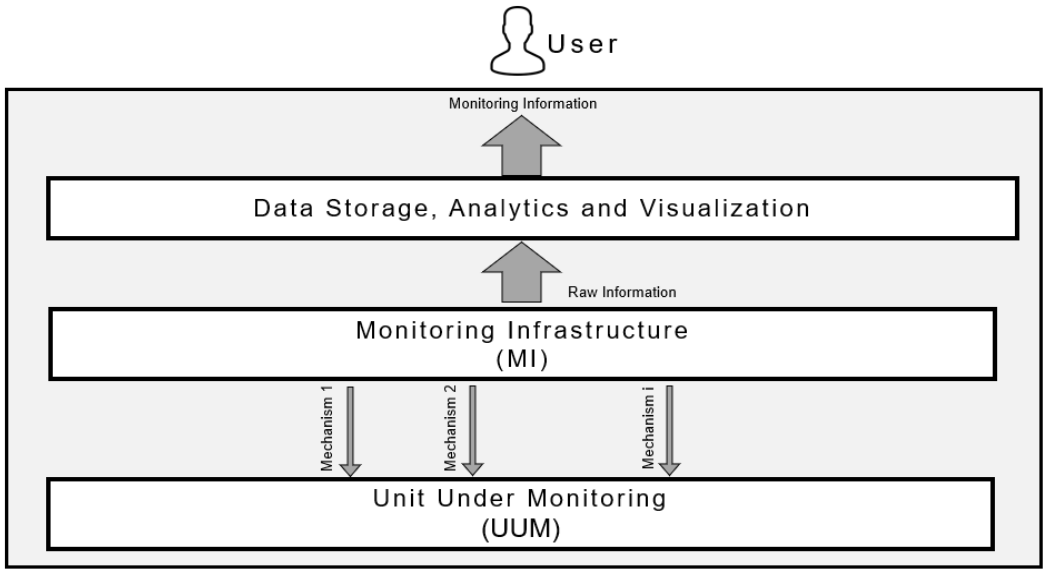

Figure 17: Reference model of a monitoring action.

4.2 Enabling Solutions to perform monitoring in FitOptiVis

In the context of Task 4.2, basing on the requirements provided by (i) Use case providers, (ii) WP3 tasks (related to design methods refinement using runtime information) and (iii) WP4 tasks (related to runtime actions starting from runtime information), some solutions have been identified and developed in order to support on the construction of monitoring systems.

In particular, CUNI developed FIVIS, a common data storage, visualization, and analysis platform. UC developed an extension to DSL that allows the expression of monitoring requirements during the model creation using the DSL [D2.1]. UNIVAQ, UNISS and UNICA developed AIPHS, a framework to build custom monitoring systems at the edge. This section reports details about these enabling solutions.

4.2.1 FIVIS data storage, visualization and analytics platform

Monitoring is one of the key components of adaptive systems based on the MAPE-k loop paradigm, because it provides basis for adaptation decisions. In general, monitoring requires the ability to periodically store a system-specific set of metrics associated with a point in time or with an observable state of a system, and to present the collected data to consumers.

In the simplest form, the data can be consumed in visual form through plots and domain-specific dashboards and adaptation can be driven by human decisions. Alternatively,

and more in line with MAPE-k paradigm, the data can be processed and analysed by a machine and reacted to in an autonomous fashion, which requires the monitoring system to also support external access to the stored data and support time-based queries to allow the data to be analysed and acted upon.

Depending on the frequency and the amount of data stored for each observation, the amount of data matching a particular query may become too large to send (potentially repeatedly) to clients across network for analysis. A monitoring system should therefore support some form of scalable data analytics to avoid transporting huge amounts of data to clients and instead transport only aggregate analysis results.

To this end, CUNI is building a common data storage, visualization, and analysis platform, FIVIS, which will provide partners with the ability to store data in a central location, build custom dashboards, execute analytic tasks, and query both data and analysis results.

4.2.1.1 Overview

The FIVIS system provides support for aggregating data from multiple sources and enables executing customized analyses on the data to provide content for customized dashboards and reports consumed by humans, as well as transformed data streams suitable for consumption by machine, e.g., components responsible for adaptation of the monitored system. To aid with creating custom visualizations, the system provides predefined widgets for displaying data using different types of charts (line chart, bar chart, pie chart, legend) and other elements (time range selector, data access). Additional types of charts for statistical visualizations are envisioned (XY chart, violin plot, box plot). Specific visualizations are created through a web-based interface provided by the system.

The system is intended to interact with different kinds of users. An *end user* is a consumer of custom visualizations and reports embedded in a use-case specific user interface panel. An end user is expected to select or change domain-specific parameters of the visual outputs, but not to define new visualizations. These are defined by an *administrator* (use-case or partner specific) by configuring and deploying panels into the use-case specific user interface, choosing which data sets to display in which panel. No programming skills are necessary. The final type of user is a *contractor*, who is responsible for creating visualization templates. These templates instantiate widgets and other elements that make up a particular panel. Each template is a snippet of JavaScript code which binds all the elements of a panel together. A contractor is expected to possess a basic knowledge of web development technologies, such as JavaScript, HTML, and CSS.

4.2.1.2 Architecture

The architecture of the system is shown in Figure 18. The system consists of a server part hosted on the system provider’s infrastructure and a client part, which executes in a web browser.

The server part is responsible for managing the data and for providing API endpoints for different tasks and users. A data entry API endpoint (Data Sink) allows external systems

to store (push) signal data into the system. Alternatively, the system can use an application-specific data pump to pull data from an external system.

Signal data represents master data obtained through observation. The system keeps the master data in a MySQL database, but generally works only with data in a temporary storage provided by the ElasticSearch framework. All master data are initially indexed by ElasticSearch, but all data derived from the master data (filtered or smoothed data, trends, etc.) are only kept in the temporary storage.

To ensure that the visualization widgets in the client remain responsive when dealing with large data sets, the system needs to avoid sending all the data matching a query to the client for rendering. Instead, the system computes all aggregates on the server side and only sends to the client data points that will be actually visible. This requires computing a significant number of aggregates in a short time (in response to information about the user's viewport and selected data).

To this end, the system uses a combination of the ElasticSearch framework, which serves as a distributed noSQL database providing near real-time searches and aggregates, and the Spark framework, which provides scalable computational platform for data analytics. Analytic data can be included in visualizations and even though they can be always recomputed from master sensor data, they are kept in ElasticSearch to improve performance. Using ElasticSearch and Spark allows scaling the computational resources as necessary to provide smooth user experience.

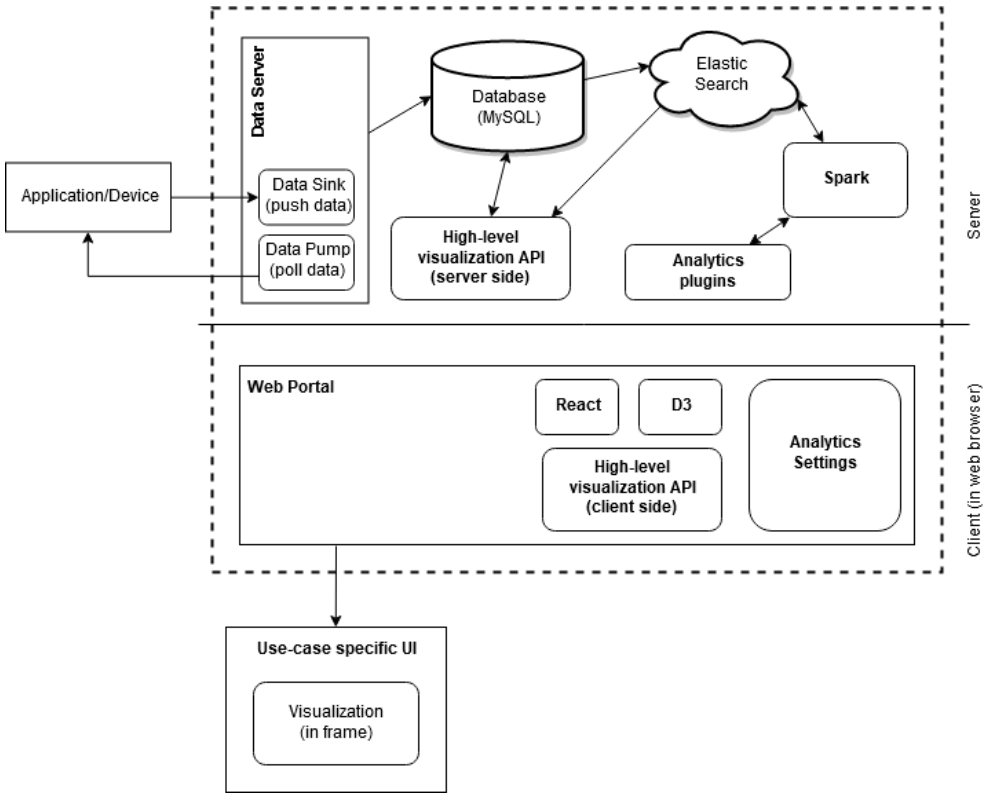

Figure 18: Architecture of the FIVIS system

4.2.1.3 Data Model

The system defines a simple meta-model for sensor data, i.e., the data stored persistently in a database. The meta-model is shown in Figure 19 below. The data are conceptually organized into uniquely identified signal sets. Each signal set groups one or more signals, where each signal represents a stream of typed values from a sensor.

Each signal set is described by a schema, which contains an ordered set of signal descriptors, one for each signal. A signal descriptor captures signal name and the type of values produced. The system currently supports logical (Boolean), numeric (Integer, Double), textual (String) and temporal (DateTime) values.

Actual data are stored in records, each of which contains an ordered set of typed values. The ordering of values corresponds to the ordering of signals in the schema. The system does not interpret the data in any way; the only requirement is that each entry has a unique identifier with a defined (ASCII) ordering. This allows the system to keep track of sensor data to determine if and where new records were inserted. This is necessary for proper scheduling of data analysis tasks – if new sensor data are appended to the existing data, the system may only need to schedule incremental analysis of the new data. If (for some reason) data are inserted in the middle of existing data, the system may need to recompute the analyses for all data.

Any other interpretation of the data (including whether the data represent a time-series or just a sequence of values without any notion of time) is left to the analysis tasks and the visualizations.

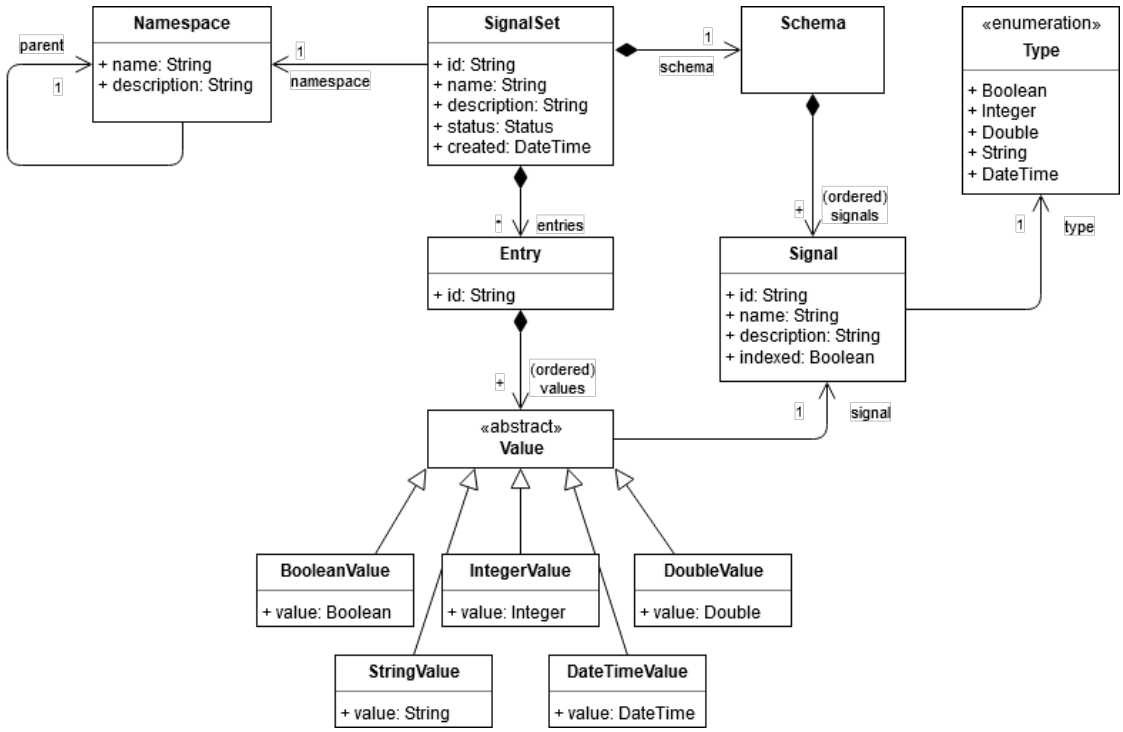

Figure 19: Meta-model of master (sensor) data in persistent storage

4.2.1.4 Data Server Interface

The Data Server part of the system provides an interface for data entry. Two modes of operation are envisioned (but only one is implemented so far):

- **Push mode**. This mode allows an application to push sensor data to the system in form of JSON payload transmitted through a REST endpoint. The frequency of updates and the amount of data transmitted is determined by the application (device) and typically depends on the capacity of internal buffers, connectivity, and available processing capacity.
- **Pull mode**. This mode is intended for devices that cannot push data to a REST endpoint, either because they completely lack a network stack, or because they do not have sufficient resources to issue an HTTP request with JSON payload. In this case, we envision the system to poll the device through a remote agent which would be responsible for obtaining the data from the device locally, in a device-specific fashion, and converting it to the expected JSON payload. No device-specific data formats or agents have been defined so far. Partners interested in using the system in this mode should contact the system provider (CUNI) for assistance.

Signal Data Payload

The JSON document with sensor data is represented either by a single payload object, or an array of such objects. A payload object is a dictionary with four keys, some of which can be optional under some circumstances:

- `partnerId: string`, identifies a particular partner. This field is required.
- `signalSetId: string`, identifies the signal set to which to store the data. The value of `signalSetId` together with `partnerId` make up a unique identifier (in the form `"partnerId:signalSetId"`) of a signal set in the system. This field is required.
- `schema: object`, defines the name and type of data for each signal in a signal set. Schema is an object with named slots, where the name of a slot corresponds to signal name and the string value of a slot denotes the signal type (currently one of `boolean`, `integer`, `double`, `string`, and `datetime`). This field can be omitted (in some cases).
    - Including the schema with each payload object allows adding new signals to the signal set on demand. However, leaving out an existing signal will cause the server to reject the payload to avoid deleting signal data by accident.
    - The schema object can be omitted if there is no need to add new signals to the signal set. However, in most cases, including the schema object in the payload should not cause noticeable overhead and allows to capture the state of migration when extending the signal set.
- `data: object[]`, which contains records with signal values. Data is a collection of objects, each representing a single record with signal values. Each record object has named slots, with slot names corresponding to signal names, and slot values holding signal values. Each record must have a slot named `id`, which represents a record's unique identifier. The identifier is free-form and providing the identifiers is the responsibility of the application. The only requirement is that ASCII ordering is well defined on the identifier, because the system uses it to establish record ordering.

The following listing shows an example of sensor data payload in the JSON format. The schema defines four signals with different types, and provides two data records which provide a record identifier in addition to signal values:

```
{
    "partnerId": "XXXX",
    "signalSetId": "YYYY",
    "schema": {
        "ts": "datetime",
        "sig1": "integer",
        "sig2": "double",
        "sig3": "boolean"
},
    "data": [
        {
            "id": "0001", "ts" : "2019-02-20T18:25:43.511Z",
            "sig1": 12, "sig2": 34.2, "sig3": true
        },
            "id": "0002", "ts" : "2019-02-20T18:25:44.000Z",
            "sig1": 12, "sig2": 34.2, "sig3": true
        }
    ]
}
```

Posting Signal Data

Signal data has to be posted to a REST endpoint. The URL of the endpoint is determined by the system operator. The system is currently operated by CUNI, but by the end of the project, we aim to provide a virtualized appliance that can be run by any partner privately

The `Content-Type` header of the POST request should be set to `application/json` and the request should contain an `access-token` header with a token that can be generated/reset in the client user interface of the FIVIS system. The following listing shows the payload from the example above being posted to the system using the `curl` utility:

```
curl https://api.FIVIS.smartarch.cz/api/signals
  --request POST \
  --header "Content-Type: application/json" \
  --header 'access-token: 8e3f4bf6bec954b40a0ec08ab0dc0c11d0d18fed'
  --data ‘{
  "partnerId": "cuni",
  "signalSetId": "test",
  "schema": {
    "ts": "datetime",
    "sig1": "integer",
    "sig2": "double",
    "sig3": "boolean"
  },
  "data": [
    {
      "id": "0001", "ts" : "2019-02-20T18:25:43.511Z",
      "sig1": 12, "sig2": 34.2, "sig3": true
    },
    {
      "id": "0002", "ts" : "2019-02-20T18:25:44.000Z",
      "sig1": 12, "sig2": 34.2, "sig3": true
    }
  ]
}’
```

For embedded devices with limited support for shell or Python, a simple FIVIS client library utilizing `libcurl` and an example CPU monitoring application supporting batched/delayed data transmission has been made available at GitHub: https://github.com/d-iii-s/FIVIS-client

Alternatively, a plugin to the Telegraf system agent is under development, which allows monitoring data to be sent to FIVIS in addition to other data storage, analysis, and visualization systems.

Computed Signals

A signal set can contain additional signals that are computed from other signals in the same record. This is useful for rudimentary filtering, e.g., for clamping or filtering values that exceed reasonable range, or for fixing data that are known to be broken in a certain time period. This approach allows leaving the master data intact, yet visualizes correct data.

For example, lateral and longitudinal acceleration calculated from GPS data can produce signals with high variability. If this is a problem for subsequent analysis or visualization, one solution would be to average the calculated acceleration over a longer time period, or clamp/filter out values that don't make sense. If we are dealing with GPS data from a car, we can reasonably assume that any kind of acceleration outside the range of [-2.0, 2.0] G is extremely unlikely for a normal car with normal tires, and we can therefore clamp or filter out such values.

To do that (while keeping the master data intact), we can create an additional signal of type Painless Script, and define an expression which produces the value of the signal based on the other signal values in the same record. In Painless Script, the document values (in our case the record containing signal values) can be accessed from a dictionary object named `doc`.

To clamp values to a specific range, we could use the following script:

```
return Math.min(Math.max(doc.{{lat_acc_g}}.value, -2.0), 2.0)
```

Alternatively, we could filter out values outside a given range using the following script:

```
def value = doc.{{lat_acc_g}}.value;
return (-2.0 < value && value < 2.0) ? value : null;
```

4.2.1.5 Data Processing

The IVIS framework underneath FIVIS provides the concepts of tasks and jobs to enable additional data processing. These make it possible to write custom programs which process existing data, or gather additional data from other resources. The framework provides a basic UI for coding, and a mechanism for job activation.

Tasks

A *task* is an element containing code, files and the definition of parameters. Each task has a type, and is handled according to that type. Two tasks differing in type may use different libraries, or completely different programming languages. A task is not directly executable – it represents a template computation on a certain type of data, but does not define where the data comes from. Instead, it defines parameters which allow passing this information into a task, and these are configured in the context of a job.

Jobs

A job holds the configuration parameters for a task, i.e., it instantiates the computational template defined by a task. Multiple jobs can utilize the same task with different parameters. A job can be activated either manually, or triggered automatically.

The framework provides the following triggers for job activation:

- **Periodic trigger**, which allows running jobs repeatedly with a set period, and
- **Signal set trigger**, which allows running jobs whenever new data is added to a given signal set.

The execution of jobs can be moderated by specifying additional conditions:

- **Minimal interval**, which ensures that a job only runs when a set interval since last run has elapsed, and
- **Delay**, which delays the execution of a job for a set interval after it was triggered.

4.2.1.6 Client Interface

The FIVIS system provides means for creating custom data visualizations built using web technologies. These visualizations can be embedded in any web application, and they can be also displayed in directly in the client user interface, organized into dashboards. To enable parametrization and reuse, the visualizations are built using the following concepts.

Workspace

A *workspace* is a top-level concept which groups related *panels* and their configuration. The framework provides UI elements to navigate to the workspace and to the panels it defines. The framework cannot display any data without a workspace with panels.

Panel

A *panel* is an element that provides a particular view of data in a particular workspace. Technically, a panel holds configuration parameters (if any) for an instance of a visualization *template*, which does the actual rendering. All panel parameters (including its name and description) are specific to a particular workspace. A workspace without panels does not display anything.

Template

A template is the most important element of the visualization framework because it does the actual rendering. In contrast, workspaces and panels are just containers. A template defines how to display data with a particular structure. Technically, a template is a React.Component which can receive parameters and defines how to visualize the data. React component can also keep state information and modify the visualization in response to state changes.

The framework provides a number of predefined components to enable rapid development of simple dashboards. Some components provide support for plotting of data from the ElasticSearch backend, while other components provide UI elements that can be used for selecting signals or time ranges to be displayed. When using the predefined components, most of the template code usually deals with constructing configuration objects which tell the components which data to display and/or where to store their state (in case of stateful components).

External Parameters

Visualization templates are necessarily going to be tailored to specific use cases. It often requires making assumptions about the kind of signals found in a signal set. These assumptions will be usually encoded in the template code, but in general developers should strive to make templates as flexible as possible.

To enable such flexibility, templates can accept external configuration parameters which are associated with an instance of a template in a particular panel of a particular workspace. This allows using a single template to display signal data from different signal sets, as long as the signal data can be interpreted in the same way. The names of the signal sets and the signal names can be provided from outside to make templates independent of data storage and management concerns. Similarly, external parameters can be used to control certain aspects of a component's output, e.g., a message to display, or a color to use.

Template parameters are described by a JSON snippet which contains an array of parameter specifier objects, one per template parameter. When instantiating a template in a particular workspace panel, the framework provides a simple editor for each template parameter so that the template can be provided with parameter values specific to that particular template instance.

Further details related to definition of template parameters and accessing them from template code will be available in technical documentation which is under development.

Plotting Components

The client part of the system provides a number of predefined plotting components. These are intended to be used in templates to visualize different kinds of data, while the role of the templates is to handle the configuration of and interaction with the plotting components.

The system currently provides plotting components to support the following charts:

- Line charts
- Area charts
- Pie charts
- Histograms

Creating new plotting components requires extending the underlying framework. Support for additional chart types, such as violin plots, are under development.
The system also provides a set of auxiliary user interface components which allow the user to select a time range, define chart legends, and pick signal sets, signals, and colors.

4.2.1.7 System Status

The system is currently in a prototype stage, with both human-machine and machine-machine interfaces still under development. An instance hosted by CUNI has been set up and allows interested partners to store sensor data (push mode) in the system through a REST endpoint. Application-specific data pumps (pull mode) are not yet supported. CUNI works with interested partners on development of initial visualization templates and panels suitable for particular use cases.

4.2.2 DSL extension to express monitoring requirements

UC has extended the QRML DSL developed in FitOptiVis (reported in D2.1) in order to support monitoring requirements. The main objective is to provide automatic monitor code generation from DSL extension SDSL.

The DSL extension includes a new language feature (monitor) that allows defining monitors. The monitor definition is independent from a particular component; the same monitor type can be used in several components.

The monitor type declaration identifies the monitor and defines two fields:

- Provider: This field allows identifying the agent that provides the traces and it depends on the tracing implementation. For example, in FIVIS the provider is the FitOptiVis partner.
- Event: list of signals traced by the monitor.

Figure 20 provides an example of a monitor declaration (VideoTrace) using SDSL.This declaration is oriented to a FIVIS implementation.

```
monitor VideoTrace{
    provider unican;
    event Performances{
        Comp:undefined;//type String
        timeStamp:real;
        EstimatedFps:real;
        Latency:real;
    }
}
```

Figure 20: Example of SDSL monitor description

In this case, the monitor (VideoTrace) includes a trace provider (unican) that is a FitOptiVis partner name. This is a FIVIS constraint but it is not required in other implementations such as lttng. The example also includes an event (Performances), although the language supports an arbitrary number of them. The event declaration defines the type of every signal that will be monitored. It is interesting to highlight that the component name (“Comp” signal of “undef” type because string is not a DSL supported type) and the time in which the event is captured (timeStamp) are included in the event signal list for FIVIS-oriented monitor generation.

The monitor types are instantiated in the components. For example, Figure 21 includes a monitor (monitor1) in the “Display” component. The “usesmonitor” reserved word is used to declare the instance. The monitor signals (EstimatedFps. performances. monitor1 and Latency. perfomances. monitor1) are associated to the monitored component qualities (EstimatedFps and Latency). The FIVIS oriented signals (Comp and timestamp) are not associated to component qualities because the generator produces specific code for them. Additionally, the generator provides a method (trace_Performances) that allows reporting the event signals from the user code to the tracer. This methodology allows tracing user-defined parameters. In order to trace

platform parameters (e.g. percentage of used CPU), a specific platform monitoring component is normally required.

```
component Display{

    requiresinterface VideoInterface ir1;
    usesmonitor VideoTrace monitor1;


    quality EstimatedFps:real;
    quality Latency:real;

    //Association between qualities and monitor parameters
    EstimatedFps.Performances.mon1==EstimatedFps;
    Latency.Performances.mon1==Latency;

}
```

Figure 21: Example of monitor instance in a component

4.2.3 AIPHS framework to build custom edge monitoring systems

4.2.3.1 Overview

In this section, an enabling solution targeting edge-computing devices is presented. Specifically, a framework to support in the runtime monitoring part of the MAPE-K loop is proposed: AIPHS (acronym of *AdaptIve Potential Hardware Profiling System*) allows the generation of monitoring systems targeting architectures implemented on FPGAs. AIPHS starts from a basic library of elements [D5.1], takes as input the monitoring requirements and the hardware architecture, providing as output a monitored system.

4.2.3.2 Monitoring system composition

AIPHS is based on a library of hardware elements written in VHDL. Basing on monitoring requirements, the framework generates a number of hardware monitors, distributed within the hardware architecture. The framework allows to generate monitoring systems that satisfy different requirements, expressed by means of metrics. The list of currently supported metrics is reported in D5.1.

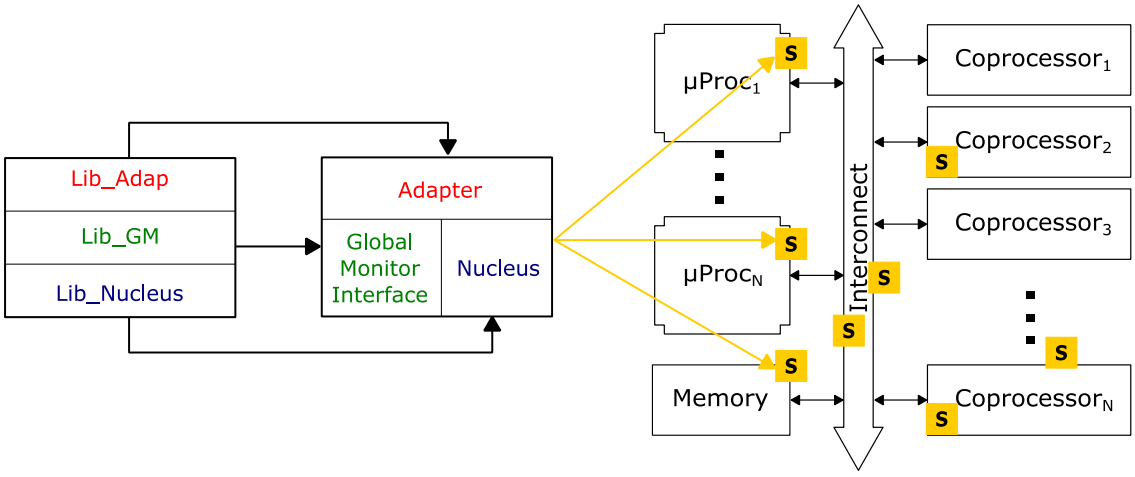

Figure 22: Monitor generation in AIPHS

The monitoring systems automatically generated by AIPHS are based on a number of *sniffers* (S) distributed within the hardware architecture of the system under monitoring. The generation of monitoring systems is reported in Figure 22: each sniffer is built by using IP-cores part of three different libraries. In particular, each sniffer has a *nucleus*, an *adapter*, and a *global monitor interface* (GMI). The adapter takes input from monitored interconnections (interconnection dependent signals) and feeds the nucleus with interconnection independent signals. In turn, the nucleus processes its inputs through two elements, namely *event monitor* and *time monitor*, that perform the monitoring action. They are both highly configurable and the nucleus results are written on a set of registers. Finally, the GMI is able to communicate with the information collector.

By taking as input the metrics to be monitored and the target hardware of the unit under monitoring, the framework automatically:

1) builds the first part of the sniffers by using the content of the *LIB_NUCLEUS* library;
2) builds the adapter and the GMI by using, respectively, the content of *LIB_ADAP* and *LIB_GM* libraries.

The current composition of the three libraries is reported in Table 3.

Table 3: Current composition of AIPHS libraries.

Library	Elements
LIB_NUCLEUS	Event Monitor Unit, Time Monitor Unit
LIB_GM (i.e., how to send monitor results)	Xilinx FSL [XIL12], AMBA APB [ARM10], AMBA AXI [ARM19]
LIB_ADAPT (i.e., what can be monitored)	Xilinx LMB [XIL16], Xilinx FSL [XIL12], AMBA APB [ARM10], AMBA AHB [ARM12], AMBA AXI [ARM19]

4.2.3.3 Interface

AIPHS is provided with bare-metal APIs to interact with the generated monitors. In the next future, Linux based APIs will be provided. Automatic parsing of the logs produced by the monitors built using AIPHS by using the *Common Trace Format* [CTF20] will be also provided.

The log file structure of the raw information exhibits a similar structure to the one reported in the work in [KOR13]:

Table 4: Structure of event instances of the monitoring systems generated by AIPHS

EVENT_ATTRIBUTE	EVENT_ID	EVENT_INFO

Each field is further customizable, depending on the application.

Example of application of AIPHS generated monitoring systems to an indoor localization algorithm, executing on Leon3 processors, is reported in Figure 23. Two sniffers are generated: s1 monitors the runtime execution of the application running on top of Linux user space (s1 is generated with NUCLEUS=Event Monitor Unit, GMI=AMBA APB, ADAPTER=AMBA AHB), while s2 monitors the correct behaviour of AMBA AHB system bus, triggering issues toward the external (s2 is generated with NUCLEUS=Event Monitor Unit and Time Monitor Unit, GMI=AMBA APB, ADAPTER=AMBA AHB). The monitoring information are sent to Leon3 processors.

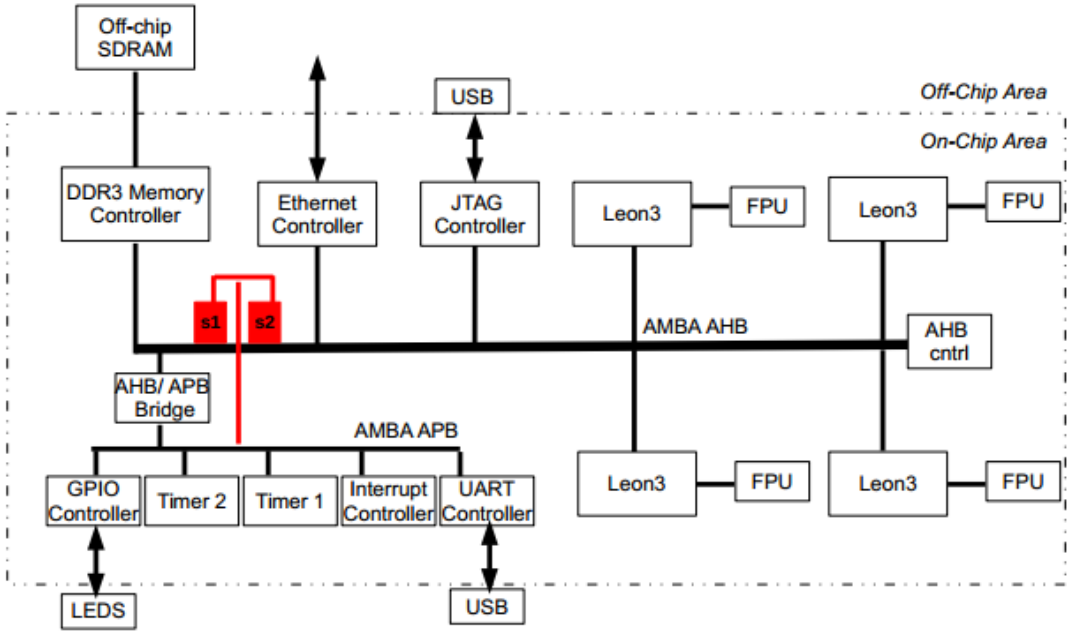

Figure 23: AIPHS generated monitors on Leon3 based platform.

4.3 Instances

The development of monitoring systems in FitOptiVis scenarios, and corresponding system-level services, involves several trade-offs from architectural point of view. In this section, the monitoring solutions developed in Task 4.2 to satisfy the different requirements are reported.

4.3.1 Monitoring in 3D industrial inspection system

ITI is developing a 3D Industrial Inspection system which is designed to use sixteen edge computer boards connected to the same number of cameras. In order to manage the state of this hardware, a plugin for Telegraf, an agent that collects time series data, is being created. Each required device or application will push information to the monitoring software hosted in a central server (FIVIS). This server receives the events, stores them, and shows the data through a graphical environment.

Monitoring Requirements

The 3D Industrial Inspection use case developed by ITI (Zero Gravity 3D) requires a monitoring system at the edge, featuring a minimal intrusiveness and very small bandwidth consumption. The level of intrusiveness depends on the time interval of monitoring events, however, even with a small interval, the intrusiveness should be minimal. As a general requirement, monitoring must not affect memory and timing performance at the edge. In other words, this process must not delay in any way the tasks performed on the edge capturer. This restriction can be partially avoided dividing the monitoring application into two parts. First, the client-side agent which is responsible of pushing events. This program accomplishes the minimal and non-intrusive requirements. Secondly, the server-side, which can be installed on a different dedicated computer, is in charge of storing the received data and providing the graphical user interface.

Regarding the information to be gathered, Zero Gravity 3D collects the following data for monitoring its state:

- Network bandwidth.
- Throughput, measured as parts per minute.
- CPU load.
- Memory usage.

Unit Under Monitoring

ITI's Zero Gravity 3D is built including sixteen edge computing boards. These devices must be monitored in a non-intrusive way and all the data produced must be sent to a central storage to be interpreted.

Monitoring Infrastructure

Telegraf is a monitoring agent that collects data from different sources such as services like databases or web servers and computer built-in sensors. Using Telegraf plugins, data can be filtered, transformed, decorated and finally stored on a file or sent to a database or any other software. Ideally, the data can be transmitted to a server providing storage and computational resources to host an application offering a graphical user interface. This agent software has been installed on all the sixteen edge boards of the 3D Industrial Inspection Case and ITI has created a plugin for sending data to FIVIS, the software that plays the role of the aforementioned storage server (see Figure 24). The plugin is based on a Telegraf Json serializer so, it will be consistent with Telegraf architecture and provides the serialization of data into a JSON document which is compatible with FIVIS.

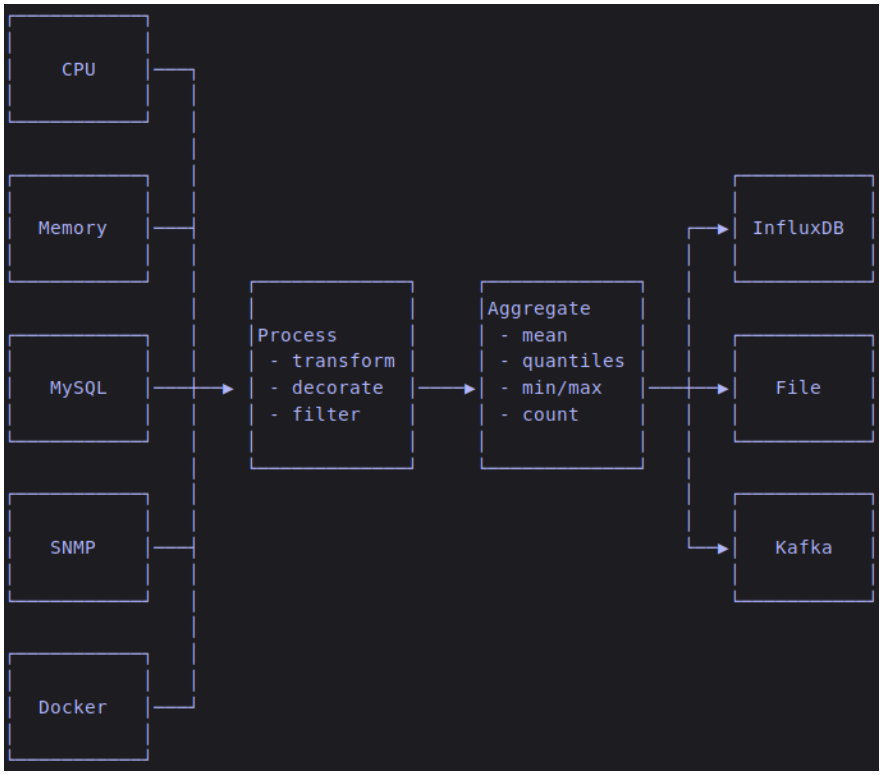

Figure 24 Telegraf data flow

This monitoring scenario can be extended to other use cases of FitOptiVis. The agent and its plugin can be installed on almost any device capable of running Linux and, in this way, the device will send the information collected by Telegraf to FIVIS.

Data Storage, Analytics and Visualization

FIVIS is the monitoring tool featuring the storage capability indicated above and the creation of dashboards from stored data. As said before, ITI has developed a standard Telegraf plugin which can be installed easily on any device capable of running Linux.

This solution provides a way to send data to FIVIS monitoring tool (see Figure 25) reducing the development effort and giving access to many data sources such as databases, network, CPU, memory usage and board sensors.

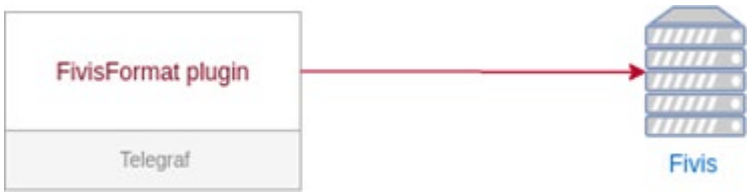

Figure 25 Telegraf plugin block diagram

4.3.2 Heterogeneous Distributed Computing Adaptation Monitoring

In the OpenCL-based heterogeneous distributed software stack, runtime monitoring data will be exposed to the application developer or the adaptation layer via an OpenCL device layer extension under development. Its completion targets the Deliverable D4.4 due in M30. This data will be used by the adaptation layer to choose different target devices (local or remote) and kernel variations (use a simpler algorithm with worse quality results or a more complex one with better results). In case of changes in condition.

Monitoring Requirements

Since the FitOptiVis software stack is a distributed stack which includes heterogeneous platforms with various type of devices having different characteristic, optimizing the computation globally is challenging. Therefore, it is crucial to produce accurate and interference free profiling data of the application execution *globally* within the distributed context.

So far during the design of the adaptation layer on top of the pocl-remote, we have identified the following data one needs to monitor to drive automatic adaptation and reconfiguration:

- **Compute clusters** in the close proximity: probing that happens whenever network connectivity changes in a roaming situation. This is to discover available compute resources to remotely offload computation to. After “the platform discovery”, the information of the found remote devices is given using standard device queries of the OpenCL API. This information can be used to assess if a more complex algorithm could be executed beneficially in the remote node.
- **Network condition** (latency, bandwidth) to the connected compute cluster. This information is used to drive the adaptation algorithm that figures out if offloading to the remote compute node is beneficial and can be done within the latency requirements of the application.
- **Device occupancy**. In the first version, this only gives information of availability of the device for the remote ones. No resource sharing in the remote is yet supported. However, for the local devices, monitoring of the local devices

(CPU/GPU utilization) is needed since it also drives the decision of when offloading is feasible or not.

Unit Under Monitoring

The distributed heterogeneous software runtime encompasses the whole execution environment, both locally and in the optional remote servers. The monitoring system thus executes in the “host” (the local device) fully, or partially, in case there are remote resources that are being monitored. In that case the remote driver or the pocl-daemon is responsible of collecting the data from the monitored compute devices.

Monitoring Infrastructure

As of M24, the monitoring infrastructure in pocl-remote is still under development and prototyping. A runnable demonstrator is expected to be ready by M30 with at least the primary monitored characteristics supported.

The profiling data that provides a global view to the application optimizer (to help design time optimizations of WP3) and the runtime adaptation layer of WP4 includes the start and end times of the kernels and their connections (dependencies) via events and shared input or output buffers. For providing the profiling data, a lightweight tracing capability was added to *pocl* infrastructure. It can be enabled by setting environment variable POCL_TRACING=cq. The implication of this setting is that all command queues get automatically their profiling data flag set on after which they start producing event time stamp information which can be collected **lazily** (whenever there’s a suitable spot in the application execution with minimal interference to the collected data) and pushed to the data collection server FIVIS lead developed by CUNI.

At M24, the basic non-intrusive time stamp collection infrastructure was added and published in the open source POCL branch. What is left for the next period to do is a component that feeds this data to the FIVIS data collection server for visualization purposes.

Data Storage, Analytics and Visualization

The data collection server has an UI that is responsible of visualizing the data which will be utilized in application optimization. At least a *swimlane* visualization of tasks executing on the distributed devices is planned.

Also other interfaces/analyzers for the data are developed so it can be inspected locally to produce feedback to the application design time (WP3). One of them is a Chromium based one, which utilizes the web request breakdown visualization integrated in the Chromium web browser engine for visualizing the execution in a swimlane-type manner**.**

The profile data collector is also designed in such a way that it can help runtime adaptation decisions in the developed automated adaptation loop: For example, auto-tuning of kernels (which implementation, parameters of the implementation) can be performed during the application execution when such a feedback loop has been implemented. This is accomplished by separating the collection of the data from the parts that push/dump the data for the consumers.

4.3.3 Monitoring systems for reconfiguration for Habit Tracking and Smart Grid

UGR is developing a component for monitoring the elderly at their own home for the Habit Tracking UC and a smart video-surveillance system for the Smart Grid UC. In both cases, UGR is considering run-time reconfiguration based on different metrics explained in the subsequent sections. The reconfiguration impacts both, the hardware resources and the software components that we run on the available platforms. In both our components, UGR sends monitoring data to the FIVIS platform, to visualize the metrics.

Monitoring Requirements

In the first place, as we are working with different NVidia SoCs (Jetson TX2 and Xavier), we are interested in monitoring the platform metrics shown below because it helps us to know the performance of our system. As well as the use of the different hardware components (CPU and GPU).

- **Temperature:** it is monitored in Celsius (ºC). We are able to measure once every second the temperature of the below components, independently.
  - Mainboard
  - CPU
  - GPU
- **Power consumption:** it is monitored in Watts (W). We are also able to monitor it for the next hardware components at least once per second.
  - CPU
  - GPU
  - RAM
- **Performance mode:** This platform is capable of changing its behaviour and its available resources at runtime. So, changing the operating frequency of the CPU and the GPU will provide different performance of the system, and also different values of power consumption and temperature. For this reason, we want to know the current operating mode active at any given time in order to find out what resources are being used. Each performance mode has an unique *id* and we measure it every time it is changed.

Habit Tracking use case

In the Habit tracking use case, we are measuring some qualities that will help us make decisions to do some reconfigurations of the system.

- **Neural Network Performance:** It is measured in frames per seconds (FPS). Inside the system we have Deep Neural Networks that analyse a video stream and outputs the confidence of which indoor action has been performed. It is measured every time a Deep Neural Network does an inference over a batch of frames. This helps us monitor if the system is working in real-time.
- **Deep Neural Network evaluation metrics:** We have trained several neural networks that offer a different ratio of their power consumption and provided accuracy. Thus, we have measured the quality of each model in terms of accuracy, F1-Score, Precision and Recall over a common test set with action videos. This is computed only once when the neural model is created. In this way, we are able to compare the models that we have, and we will adapt the

target model according to the system requirements. This metric is measured every time the Deep Neural Network is changed due to a reconfiguration.

- **Confidence of recognized actions:** When a video stream is fed into the system, we get the probabilities of which action has been performed in a sequence of frames. Recording this information is useful because it can give us an idea of the system accuracy and requirements. For example, if for a period of time the current active neural model is not capable of distinguishing between two actions, we can reconfigure the system and use a better but more complex model, that will also use more resources but that achieves better accuracy according to the tests done previously on the available benchmarks. It is measured every time an inference over the video is performed.

Smart-Grid Surveillance system use case

In order to carry out re-configurations in our system, we monitor certain metrics specific to the video-surveillance task and the quality of the classification of the machine learning models involved:

- **Edge performance**: Measured in frames per second (FPS). Since it is a distributed system with a server-node structure, the performance of the software running on each of the components must be evaluated to determine whether the requirement for real-time operation is met.
- **Cloud software performance**: Measured in frames per second (FPS). Similarly to the above, it has to be determined whether the software on the server side is running to meet our real-time requirements.
- **Joint system performance**. Measured in frames per second (FPS). As it is a distributed system, the aggregated performance of the system has to be determined jointly, considering both the part that is executed at edges and the part of the system is executed at the cloud/server.
- **Confidence in people detection**: Given anomalous situations in the scene, in which certain regions of interest are analyzed, this metric provides a measure of confidence in which it is reported whether each of these anomalous situations is given by the presence or absence of a human subject.
- **Similarity in people re-identification**: When one or more people are detected, the extraction of characteristics for re-identification is carried out using the history of monitoring carried out to date. By comparing these characteristics of the identified subjects with those of the previously identified subjects, a similarity of re-identification is generated. This similarity metric serves to alert us if a new human subject appears on the scene, not yet considered, and which could give cause to reconfigure the system in some way in order to clarify a possible intrusion.

Units Under Monitoring

Smart-Grid and Habit Tracking Use Cases both make use of two main types of platforms to operate. Firstly, the work of both systems at node level, is carried out in one or more System on Chip (SoC) NVidia embedded platforms.

For the Habit Tracking UC we will only use the Jetson Xavier edge node, while the Smart Grid UC will make use of a Jetson Xavier edge node and a Jetson TX2 edge node. On the other hand, work at the cloud level, where some of the most demanding computing

is done, runs on a high-performance PC that acts as a server. The main characteristics of the SoCs used as nodes in both use cases are described below:

- **NVidia Jetson TX2**: The Jetson TX2 module corresponds to a System on a Chip platform with a six-core CPU (2 Denver 64-bit CPUs + Quad-Core A57 Complex), with 8 GB L128 bit DDR4 memory and a GPU with NVIDIA Pascal™ architecture with 256 CUDA cores. This fully-configurable device supports different working modes.
- **NVidia Jetson Xavier**: The Jetson Xavier module corresponds to a System on a Chip platform with a octa-core CPU (8-core Carmel ARM v8.2 64-bit CPU), with 16GB 256-Bit LPDDR4x memory and a GPU with 512-core Volta with 64 Tensor Cores. Again, this device is fully configurable adapting performance, energy consumption, or working frequency.
- **High-performance PC**: The PC that will act as a server has a 6-core CPU (intel i5-8400), 32GB-RAM memory, as well as a RTX 2080Ti GPU with 4352 GPU-cores

For the Smart-Grid use case, the different SoC platforms presented are used for distributed processing at edge level. These platforms are in charge of carrying out video local surveillance tasks on the video stream coming from the camera connected to each of the edge nodes. To bring together the information from the different nodes to carry out more complex tasks such as tracking people in a multi-camera environment, part of the processing is done at the cloud level. Both the Jetson TX2 and the Jetson Xavier models are used to demonstrate the heterogeneity, adaptability and scalability of the video surveillance system.

As for the Habit Tracking use case, the NVidia Jetson Xavier is the device used to run the system. It is connected to a camera that provides a video stream. This video is processed and analyzed with a Deep Neural Network inside this SoC platform. Finally, it outputs the confidence of which actions have been recognized in the video feed.

Monitoring tasks in both cases are carried out in the edge nodes of the system, which correspond to the SoCs presented above. Qualities such as the temperature of the platforms, their energy consumption, or the performance mode in which they operate are constantly monitored. Additionally, other domain-specific metrics are also taken into account (discussed in the text below in more detail) for example: 1) for the Habit Tracking use case, performance or confidence metrics from the Deep Learning models used are shown; 2) for the Smart-Grid use case, the system performance at each of the edge nodes level, confidence of the algorithms when carrying out detection and/or re-identification of people.

Monitoring Infrastructure

To monitor the metrics and qualities of our systems we are using several tools:

- **Python software**: We have developed a Python library that is capable of gathering information about temperature, power consumption of the different hardware components, as well as the active performance mode. This library collects data from checking some *sysfs nodes* from the device.
    - **Temperature:** This data is obtained in *milliCelsius* and then it is converted to *Celsius*.

- **Power consumption:** This data is originally in *milliWatts* and then it is converted to *Watts*.
- **Performance mode:** It is a unique value that identifies the active performance mode. It is obtained through the command *nvpmodel*.

Habit Tracking use case

- **Python main system software**: The habit tracking main system is developed in Python, and within this system, two qualities are measured:
  - **Neural Network Performance:** This is measured by dividing the number of frames being analyzed in the neural network model inference by the time the inference took.
  - **Confidence of recognized actions:** This is obtained as the output of the neural network model, because it assigns to each action a probability that it has occurred during the analyzed video between 0 and 1.
- **Python script**: This Python script uses *Keras* and *scikit learn* to measure the evaluation metrics of a Deep Neural model over a test set with videos.
  - **Deep Neural Network evaluation metrics:** These metrics are computed doing an inference over all the videos of the test set, which are videos that the neural network has not seen before during training, and then compare if the output of the model matches the real action performed in each video. The test set is composed of videos from different datasets:
    - **Online action recognition datasets:** We have compiled 2295 videos from a variety of heterogeneous datasets (see Table 5).

Table 5: Heterogeneous datasets

Dataset	Year	Actions	Clips
HMDB51 [KUE11]	2011	51	6766
UCF-101 [SOO12]	2012	101	13320
Fall Detection Dataset [CHAR13]	2013	2	222
Charades [SIG16]	2016	157	66500
STAIR Dataset [YOS18]	2018	100	102462

Kinetics [CAR18]	2018	600	495547

- **Own recorded videos:** We have also recorded more than 200 videos at home to test the neural network model with videos similar to those that will be analyzed when making a real use of the system (see Table 6).

Table 6: Recorded Video

Dataset	Year	Actions	Clips
Our own (TBD [UGR20])	2020	16	233

Smart Surveillance use case

- **Python edge software:** The part of the Smart-Grid Intelligent Video Surveillance System that runs in a distributed way in the different nodes has been developed in the Python programming language. From this software, the following qualities are extracted:
  - **Edge software performance**: Measured in frames per second (FPS). It is calculated by dividing the number of frames of the video stream analyzed in each of the nodes, by the time employed in that task. For this timing, we make use of the *time* function of the native Python *time* library, which returns the number of seconds passed since epoch with millisecond precision.
  - **Confidence in people detection**: Confidence (%) in the classification of each of the regions of interest considered is obtained by inferring these regions through our machine learning model implemented with TensorFlow. With this, we obtain the confidence with which one of our regions of interest includes or not a human subject.
- **Python cloud software:** The part of the system that ultimately runs in the cloud has also been implemented with the Python programming language. These are the qualities that are calculated in it and how they are obtained:
  - **Cloud software performance**: Measured in frames per second (FPS). It is calculated by dividing the number of frames corresponding to the same moment of time and coming from each of the nodes, by the time needed to process them and perform the tracking task on them. For this timing, we make use of the *time* function of the native Python *time* library, which returns the number of seconds passed since epoch with millisecond precision.
  - **Joint system performance**. Measured in frames per second (FPS). It is the sum of the cloud and edge software performance metrics.
  - **Similarity in people re-identification**: A characteristic vector of a human subject is extracted with a deep learning self-encoder model developed with TensorFlow. Subsequently, an assignment problem is solved between this feature vector and those of the human subjects already detected previously by our system. The inverse of the Euclidean distances between these characteristic vectors is the measure of similarity of a human subject in re-identification.

- **System evaluation metrics**: In order to have a picture of the system's performance in each of its operating modes or configurations, evaluation measurements of the system as a whole are also obtained, running on different test datasets. Examples of these metrics are: multiple object tracking accuracy and precision, human subjects mostly followed and lost, identity switches in re-identification, etc. These are the different datasets used for the calculation of these metrics:
  - **Third-party datasets**: In order to test the performance of the system in the detection, tracking and re-identification of human subjects (see Table 7).

Table 7: Third-party datasets.

Dataset	Reference	Description
INRIA Person Dataset	[DAL05]	This dataset contains 1805 images and X people normalized to 64x128 pixels. The people are usually standing, but appear in any orientation and against a wide variety of background image including crowds
VIRAT Video	[OH11]	This surveillance video dataset is characterized by collecting data from natural scenes that showed people

Dataset		performing normal actions in standard contexts, with uncontrolled and disordered backgrounds. This dataset includes the recording of different types of human actions, recorded in multiple locations, in more than 29 hours of video feed.
Oxford Town Centre Dataset	[HAR19]	The Oxford Town Centre dataset is a CCTV video of pedestrians in a busy downtown area in Oxford and includes approximately 2,200 people. The Oxford Town Centre dataset is unique in that it uses footage from a public surveillance camera that would otherwise be designated for public safety.
Duke MTMC	[RIS16]	Duke MTMC (Multi-Target, Multi-Camera) is a dataset of surveillance video footage taken on Duke University's campus. The dataset contains over 14 hours of synchronized surveillance video from 8 cameras at 1080p and 60 FPS, with over 2 million frames of 2,000 students walking to and from classes.

- **Own recorded dataset**: Inside the facilities of our university, some shots have been recorded simulating the setup and the real situations for which the system is designed. These videos are composed of **4 shots** from **two cameras** that record the same infrastructure from different perspectives, with overlapping and individual recording between cameras. The simulated actions in this dataset are listed below:
  - Normal behaviour of operators in an electrical substation: Walk through the facilities, fixing components, etc.
  - Interaction between operators without occlusions: Two or more people walking through the facilities together, conversation between operators, etc.
  - Interaction between operators with occlusions: Salutation with contact, occlusions between operators when walking, etc.
  - Interaction of the operators with the perimeters of the installation: Walk around the safe perimeters without entering them (lurking), intruding into these perimeters, leaving them, etc.

Suspicious behaviour score: Using space-time information from the tracking of human subjects within the scene, the degree to which the subject's behaviour is suspicious is determined. This measure is interesting to reconfigure the system to pay

more attention to a subject or scene shown by one or several cameras when, for example, there is an intrusion in the installation or in a secure perimeter.

Data Storage, Analytics and Visualization

Currently, we are using the FIVIS platform to store and visualize our monitored data. The monitored metrics are sent to FIVIS once they are recorded. Next, we can see some illustrative examples for some of the monitored data.

- **Temperature:** Figure 26 shows the temperature of the mainboard, the CPU and the GPU in Celsius during a concrete period of time.

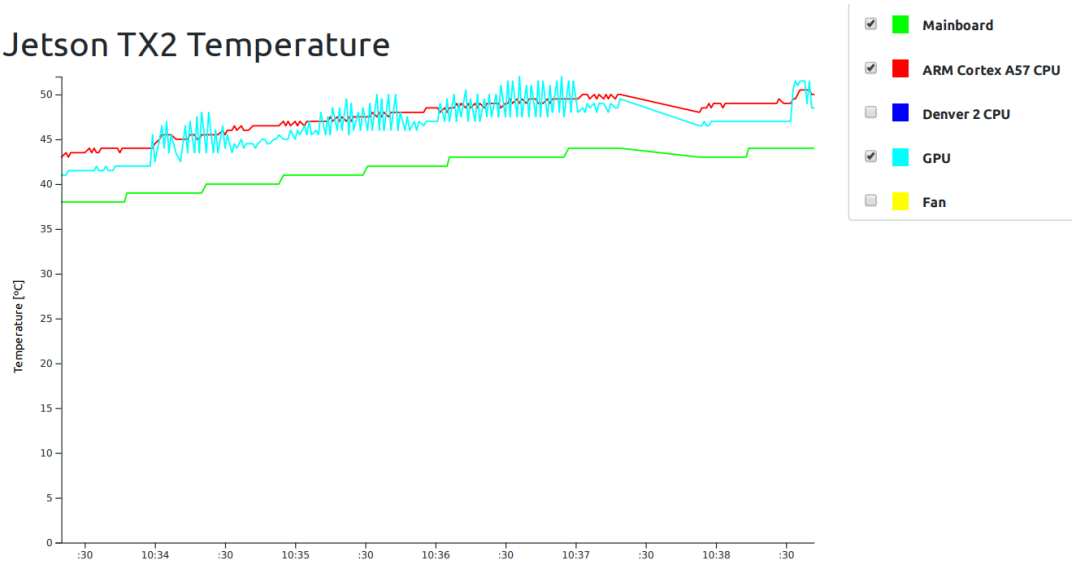

Figure 26: Temperature plot.

- **Power Consumption:** Figure 27 shows the power consumption at each second while our system is running.

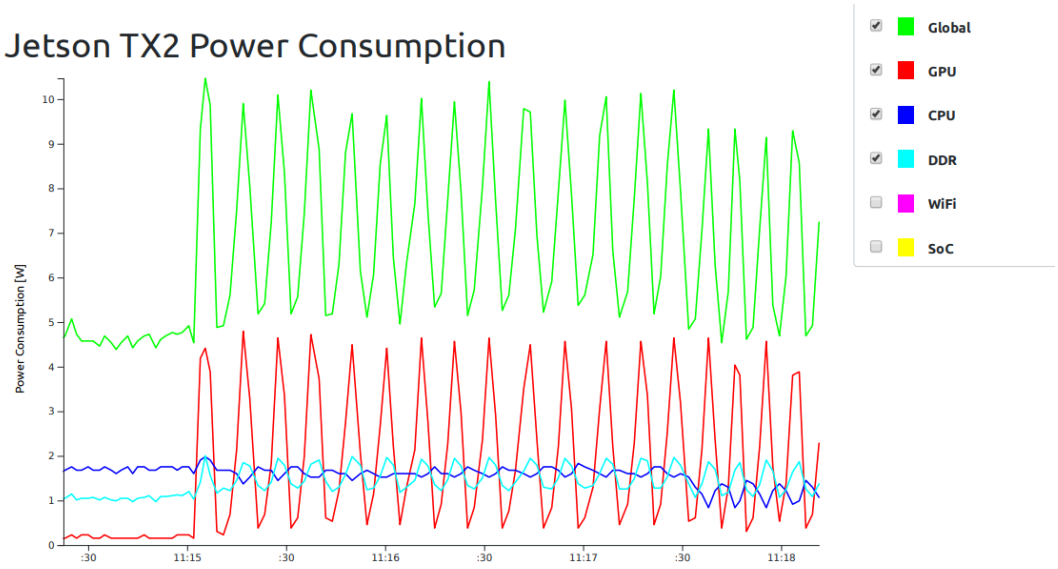

Figure 27: Power consumption plot.

Habit Tracking

In the Habit Tracking use case, we have two concrete qualities, which are shown in the visualizations below.

- **Neural Network Performance:** We are able to check here that the system is running between 46 and 51 frames per second, achieving real time performance (see Figure 28).

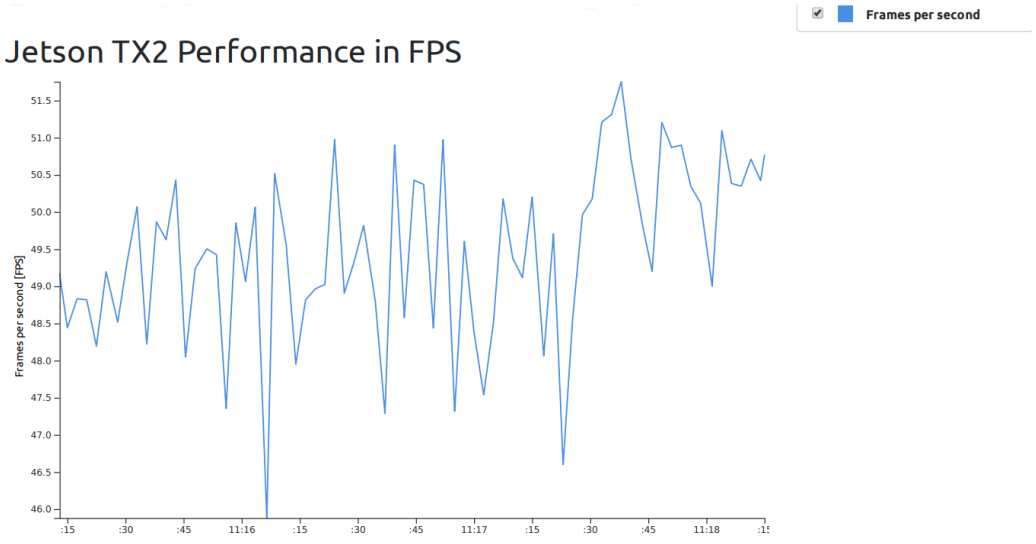

Figure 28: Performance in frame per seconds.

- **Confidence of recognized actions:** In Figure 29, we can appreciate how the actions detected vary along time. It mainly detects that someone is eating with a

high confidence over the 80%, and then other actions are recognized with a low and similar percentage.

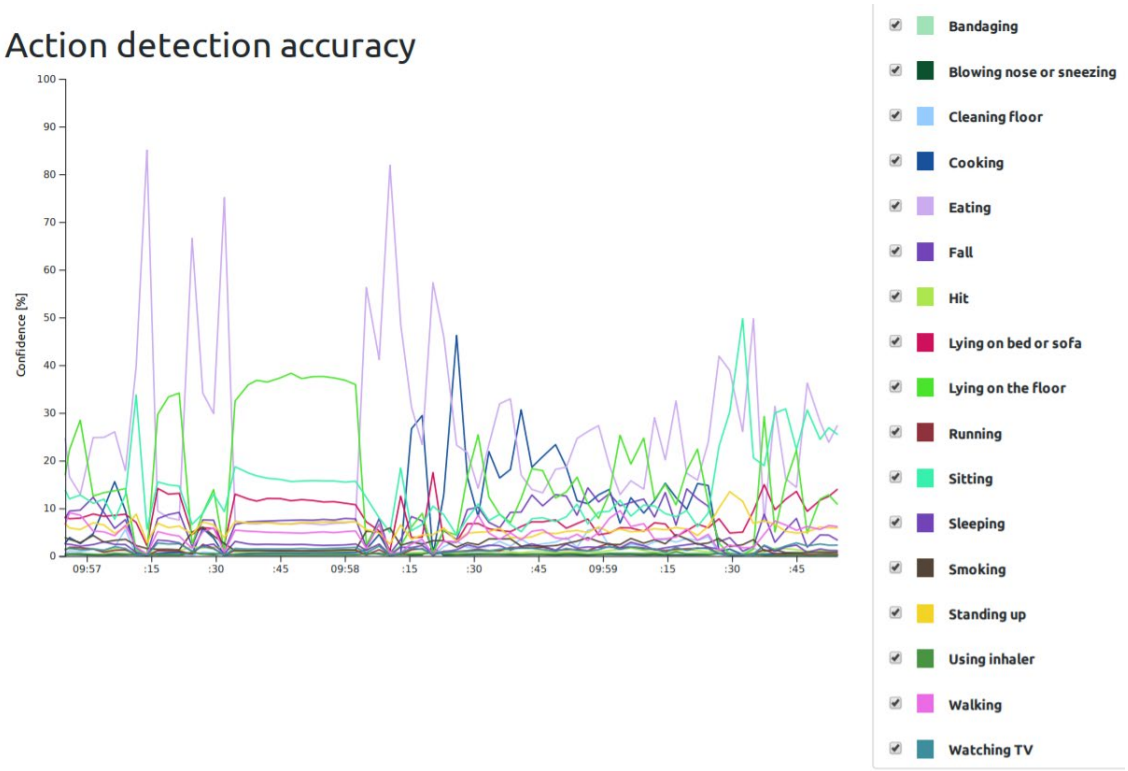

Figure 29: Action detection accuracy.

4.3.4 Monitoring capabilities for object recognition in space applications

In this section, we describe some monitoring capabilities that will be used in the autonomous space exploration use case for object recognition. The methodology provides two types of monitoring mechanisms:

1. Internal quality parameter monitoring. This mechanism was described on section 4.2.2. The DSL component description could define component qualities. These qualities could be modified at run time by the component code in order to guide the runtime reconfiguration process. In this case, a system component (runtime reconfiguration manager) could autonomously decide to modify the system configuration (resilience systems).

2. Outward trace monitoring. In this case the system reports runtime traces to an external trace storing and visualization framework. These traces could be used to modify the system state. For example, the system uses the Linux lttng package in order to collect and manage the runtime traces. These monitors are not included in the SDL description.

This section is focused on monitor requirements for the autonomous space exploration use case that use both mechanisms.

Monitoring Requirements

Hardware platform parameters are monitored using the lttng tracing framework for the Linux kernel (first mechanism). Specifically, the following parameters must be monitored.

- Memory occupancy: Platform monitorization reports memory occupancy along time to be aware of memory can be provided to application components.

- Available cores: We monitor along time the number of cores which are available to be used by the different application tasks.

Application components have also different monitoring requirements. These monitors use the first approach (Internal quality parameter monitoring) and they are defined in the system SDL description:

- Frame rate: Video processing applications must consider the frame rate to fit provided fps with required fps among components. This event is used to evaluate the system performance and the component behaviour.
- Latency: This quality evaluates the time that a particular service has spent to execute its task. The latency of individual components or services is used to detect possible bottlenecks and to take reconfiguration decisions if neccesary.

- Compression rate: Video compressor component gives a measure of the compression rate that it applies to the input video. Component configuration parameters (e.g. compression quality) allows modifying the compression rate and improve the system performances.

- Object recognition percentage: Recognizer component provides a quality that indicates the probability of detection of an object. If the probability is too low, the runtime manager could modify the convolutional neuronal network or the trained weight is order to improve results.

- Radiolink rate: The space satellite has a radio link with the ground station. The performances of this link can change during the time. This quality evaluates the current performances of this radio link.

Unit Under Monitoring

The space application will be executed in several physical platforms, each of them with different features and resources:

- Nvidia Jetson TX2: It is a power-efficient embedded computing device. It's built around an NVIDIA Pascal™-family GPU and loaded with 8GB of memory and 59.7GB/s of memory bandwidth. It contains different kind of hardware interfaces that make it easy to integrate it into a wide range of products and applications.

- Nvidia Jetson Nano: It´s a small powerful embedded system used on application which requires low power consumption. It includes and NVIDIA Maxwell family GPU, an ARM Cortex-A57 processor and 4 GB of memory.

- Zynq Ultrascale + ZCU106: It Combines four Arm Cortex-A53 high-performance energy-efficient 64-bit application processors with two Arm Cortex-R5F real-time processors and a programmable logic array (FPGA). This platform provides power savings, heterogeneous processing, and programmable acceleration.

Monitoring Infrastructure

From SDSL monitor description, a generator creates a C++ monitor implementation. This implementation could use the LINUX lttng library for trace management or other infrastructures (e.g. FIVIS).

Figure 30 shows a simple video trace of the frame rate parameter for the component “Display” using lttng:

```
[12:31:06.964179559] (+20.003809525) fernando-X556UJ VideoTrace:frameRate: { cpu_id = 0 }, { field_component = "Display", field_fps = 3 }
[12:31:26.967666958] (+20.003487399) fernando-X556UJ VideoTrace:frameRate: { cpu_id = 1 }, { field_component = "Display", field_fps = 3 }
[12:31:35.793486576] (+8.825819618) fernando-X556UJ VideoTrace:frameRate: { cpu_id = 3 }, { field_component = "Display", field_fps = 6 }
[12:31:42.930633185] (+7.137146609) fernando-X556UJ VideoTrace:frameRate: { cpu_id = 1 }, { field_component = "Display", field_fps = 5 }
[12:31:50.571483931] (+7.640850746) fernando-X556UJ VideoTrace:frameRate: { cpu_id = 1 }, { field_component = "Display", field_fps = 7 }
[12:31:58.388824704] (+7.817340773) fernando-X556UJ VideoTrace:frameRate: { cpu_id = 0 }, { field_component = "Display", field_fps = 6 }
[12:32:06.957885750] (+8.569061046) fernando-X556UJ VideoTrace:frameRate: { cpu_id = 0 }, { field_component = "Display", field_fps = 4 }
[12:32:15.291193776] (+8.333308026) fernando-X556UJ VideoTrace:frameRate: { cpu_id = 1 }, { field_component = "Display", field_fps = 8 }
```

Figure 30: Example of fps monitorization trace in a component

These results are used to dynamically reconfigure the systems, using the RIE methodology. For example, if the system needs to produce more frame per second than provided, the runtime manager selects a set point in which a component is implemented into a FPGA, in order to increase the frame rate.

4.3.5 Monitoring of 8xSIMD Floating point Accelerators

In Y2, UTIA developed run-time reconfigurable 8xSIMD Floating point Accelerators for Zynq 7000 and Zynq Ultrascale family of devices. See Figure 31. Detailed description is reported in D5.2. Integration into complete Linux system is described in D3.2. See the released application notes and evaluation packages [KAD19-1], [KAD19-2] for details. The runtime reconfiguration is described in D4.2. The runtime monitoring is described in this section.

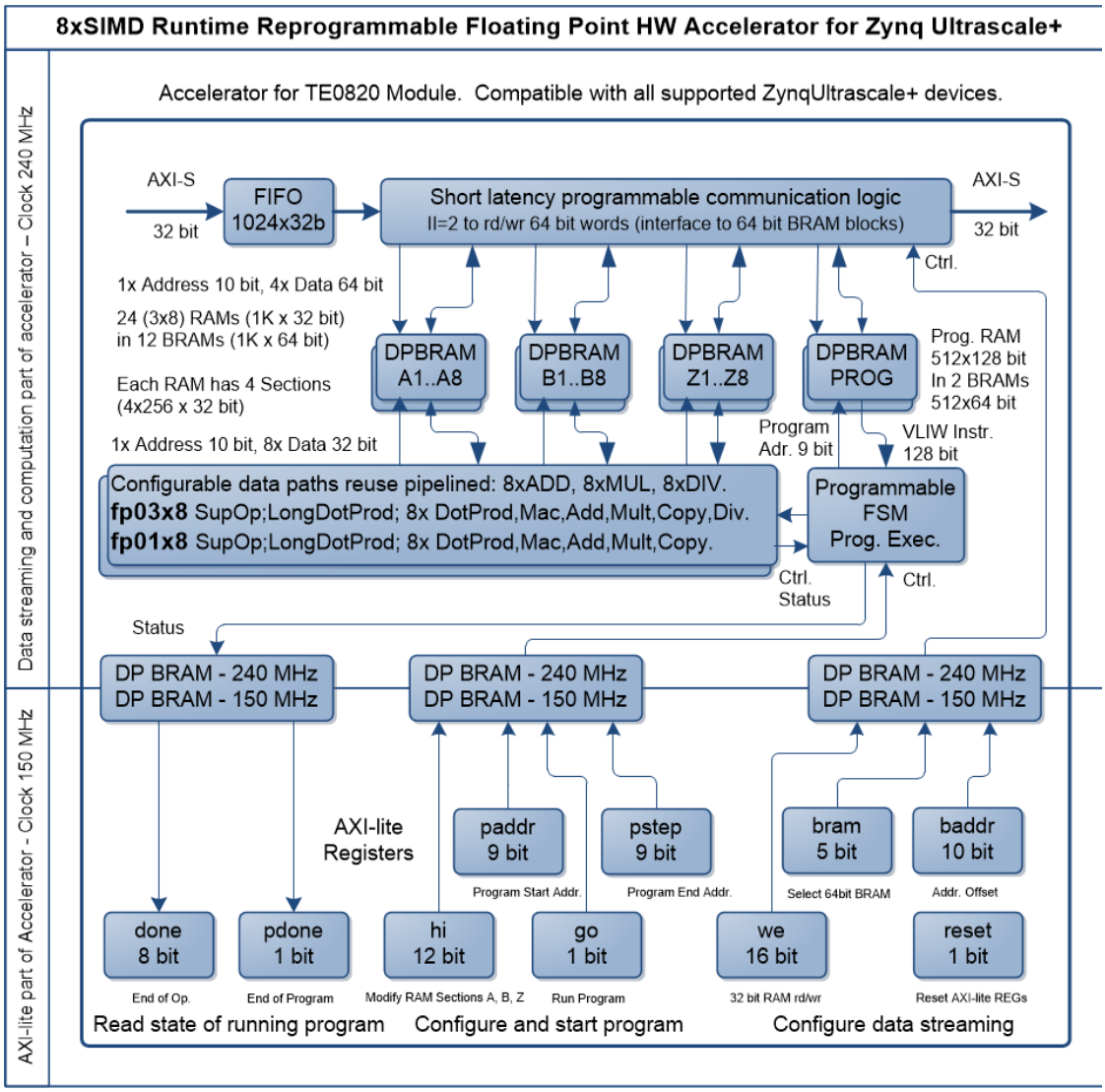

Figure 31 8xSIMD Floating point run-time reconfigurable accelerator

Monitoring Requirements

SW application can run on embedded HW with one or several accelerators with possibly different capabilities. It is therefore required to provide mechanisms for the SW to identify in the run-time what are the capabilities of the HW accelerator currently present in the programmable logic (PL) of the Zynq or zynq Ultrascale+ device. Base on this information, the processor can decide how to program the HW accelerator and what operations to accelerate.

Unit Under Monitoring

The unit under monitoring is one 8xSIMD run-time reconfigurable floating point HW accelerator with internal structure described in Figure 31.

Table 8: Floating point functions present in all accelerators {10 or 20 or 30 or 40}.

SIMD OP	code (dec)	8xSIMD Floating Point Operation Description
VVER	0	Return capabilities of the accelerator and status of license
VZ2A	1	8xSIMD vector copy $a_m[i] <= z_m[j]$; m=1..8
VB2A	2	8xSIMD vector copy $a_m[i] <= b_m[j]$; m=1..8
VZ2B	3	8xSIMD vector copy $b_m[i] <= z_m[j]$; m=1..8
VA2B	4	8xSIMD vector copy $b_m[i] <= a_m[j]$; m=1..8
Auto-increments:		*Example: for (n=0;n<=CNT;n++){i=i+B_INC; j=j+A_INC;}*
VADD	5	8xSIMD vector add $z_m[i] <= a_m[j] + b_m[k]$; m=1..8
VADD_BZ2A	6	8xSIMD vector add $a_m[i] <= b_m[j] + z_m[k]$; m=1..8
VADD_AZ2B	7	8xSIMD vector add $b_m[i] <= a_m[j] + z_m[k]$; m=1..8
Auto-increments:		*Example: for (n=0;n<=CNT;n++){i=i+B_INC; j=j+A_INC; k=k+Z_INC;}*
VSUB	8	8xSIMD vector sub $z_m[i] <= a_m[j] - b_m[k]$; m=1..8
VSUB_BZ2A	9	8xSIMD vector sub $a_m[i] <= b_m[j] - z_m[k]$; m=1..8
VSUB_AZ2B	10	8xSIMD vector sub $b_m[i] <= a_m[j] - z_m[k]$; m=1..8
Auto-increments:		*Example: for (n=0;n<=CNT;n++){i=i+B_INC; j=j+A_INC; k=k+Z_INC;}*
VMULT	11	8xSIMD vector mult $z_m[i] <= a_m[j] * b_m[k]$; m=1..8
VMULT_BZ2A	12	8xSIMD vector mult $a_m[i] <= b_m[j] * z_m[k]$; m=1..8
VMULT_AZ2B	13	8xSIMD vector mult $b_m[i] <= a_m[j] * z_m[k]$; m=1..8
Auto-increments:		*Example: for (n=0;n<=CNT;n++){i=i+B_INC; j=j+A_INC; k=k+Z_INC;}*

Accelerators fp01x8 with all capabilities do not support 8xSIMD floating point division.
Accelerators fp03x8 with all capabilities support 8xSIMD floating point division.

Table 8 indicates supported vector floating point operations identical for all accelerators. Table 9 provides list of specific vector functions which might be also implemented in the accelerator. Each of these additional functions (see Table 9) requires extra HW resources related to wider data multiplexers and finite state machines (FSMs). These resources also require extra static and dynamic power. That is why the capabilities of accelerators can be different.

Table 9: Floating point functions in accelerators with the capabilities {10, 20, 30, 40}.

SIMD OP code (dec)	8xSIMD Floating Point Operation Description
VPROD 14 **FP01, FP03: 30,40**	8xSIMD vector products. $z_m[i] <= a_m'[j..j+nn]*b_m[k..k+nn];$ m=1..8; nn range 0..255
VMAC 15 **FP01, FP03: 20,30,40**	8xSIMD vector MACs. $z_m[i..i+nn] <= z_m[i..i+nn] + a_m[j..j+nn] * b_m[k..k+nn];$ m=1..8; nn range 0..10
VMSUBAC 16 **FP01, FP03: 20,30,40**	8xSIMD vector MSUBACs. $z_m[i..i+nn] <= z_m[i..i+nn] - a_m[j..j+nn] * b_m[k..k+nn];$ m=1..8; nn range 0..10
LONG_VPROD 17 **FP01, FP03: 40**	Single long vector product . $z_m[i] <= ((a_1'[j..j+nn]*b_1[k..k+nn]+a_2'[j..j+nn]*b_2[k..k+nn]) + (a_3'[j..j+nn]*b_3[k..k+nn]+a_4'[j..j+nn]*b_4[k..k+nn])) + ((a_5'[j..j+nn]*b_5[k..k+nn]+a_6'[j..j+nn]*b_6[k..k+nn]) + (a_7'[j..j+nn]*b_7[k..k+nn]+a_8'[j..j+nn]*b_8[k..k+nn]));$ m=1..8; nn range 0..255
VDIV 20 **FP03: 10,20,30,40** **FP01: not supported**	8xSIMD vector Division. $z_m[i] <= a_m[j] / b_m[k];$ m=1..8
Auto-increments:	*Example: for(n=0;n<=CNT;n++){i=i+Z_INC; j=j+A_INC; k=k+B_INC;}*

If the accelerator executes instruction VVER, it returns to the dedicated place in its internal memory an unsigned 32 bit value with information about the capabilities of the unit under monitoring.

FP01x8 Accelerator

- 16383 = 0000 3FFF capability {10}
- 32767 = 0000 7FFF capability [10, 30}
- 114687 = 0001 BFFF capability {10, 20}
- 131071 = 0001 FFFF capability {10, 20, 30}
- 262143 = 0003 FFFF capability {10, 20, 30, 40 }

FP03x8 Accelerator

- 1064959 = 0010 3FFF capability {10}
- 1081343 = 0010 7FFF capability [10, 30}
- 1163263 = 0011 BFFF capability {10, 20}
- 1179647 = 0011 FFFF capability {10, 20, 30}
- 1310719 = 0013 FFFF capability {10, 20, 30, 40}

FP01x8 and FP03x8 Accelerator – Status of the evaluation license

- If the two MSB bits of the unsigned 32 bit value are equal to zero, this indicates that the evaluation license will expire soon.

- In case of the commercially available release version of the accelerator, the two MSB bits of the unsigned 32 bit value are always equal to one and the license will never expire.

Monitoring Infrastructure

Monitoring infrastructure is using the AXI-streaming data communication interface:

Monitoring infrastructure - standard data interfaces

Type of interface	Device	Clock	Device	Clock
• Data streaming I/O: AXI-S 32 bit	ZynqUltrascale+:	240 MHz;	Zynq	115 MHz
• Computation: 8xSIMD FP32	ZynqUltrascale+:	240 MHz;	Zynq	115 MHz
• Firmware program VLIW 128 bit	ZynqUltrascale+:	240 MHz;	Zynq	115 MHz
• Configuration I/O: AXI-lite 32 bit	ZynqUltrascale+:	150 MHz;	Zynq	100 MHz

Monitoring infrastructure - standard data communication infrastructure

The design time support (WP3) have provided in Y2 the design flows for automated generation of these data streaming HW (data movers):

- Zero Copy HW data mover without DMA unit
- DMA HW data mover with DMA unit
- SG DMA HW data mover with interrupts

Monitoring processor is ARM A9 or A53 running user application under Debian OS.

- Firmware is re-programmable in run-time by data streaming.
- Computation & data streaming can be performed in parallel.

Monitoring infrastructure - standard AXI-lite Registers

Each accelerator is controlled from ARM by set of AXI-lite registers. See Figure 31:

Monitoring processor reconfigures accelerator by change of firmware

The 8xSIMD Accelerator executes sequences of VLIW vector 8xSIMD instructions defined in a program sequence (firmware) of accelerator program memory. This firmware can be first defined in user SW program and then downloaded via the streaming interface to the accelerator. The content of the program memory will usually contain multiple different sequences of VLIW instructions. Computation performed in the accelerator can overlap with stream-based data communication. This is controlled by SW user-defined code from the Arm user-space app level and it can be used for the run-time reconfiguration by loading of new program sequence to program memory of the accelerator in parallel with performed computation.

Example: Matrix multiplication

User needs to perform HW accelerated matrix multiplication Z[64,64] = A[64,64] * B[64,64]. The complete matrix operation is split into shorter sequences of VLIW instructions. The complete matrix product is scheduled by SW user code in Arm by run-time reconfiguration of pointers to pre-loaded programming sequences.

This run-time reconfiguration process is performed in parallel to streaming part of data of matrix B[64,64] from DDR to the accelerator. Rows of this matrix are propagated as identical to all 8xSIMD memories in 8 subsequent stages.

Example: reconfigure accelerator by temporary change of firmware

User can reconfigure accelerator by

1. temporary saving of some data and firmware from accelerator to DDR
2. change of firmware,
3. execution of this firmware (for example the SupOp instruction)
4. reading of results from data memory of the accelerator to arm SW
5. returning back the original firmware from data stored in DDR
6. returning back the original data from data stored in DDR

After performing these 6 steps, the accelerator data and firmware is restored to its original state and the SW user/developer have information from the accelerator about supported 8xSIMD operations and also about the status of the HW license.

Example: read accelerator capabilities

User needs to find what vector 8xSIMD operations are actually supported by the accelerator. This information is needed for SW decision, which firmware version can be used. If 8xSIMD DotProd instruction is supported by the accelerator, the accelerated matrix multiplication Z[64,64] = A[64,64] * B[64,64] will use them for efficiency reason. If DotProd instruction is not supported, the matrix multiplication can be “assembled” by the user Arm SW by sequencing of 8xSIMD vector Mac (multiply and accumulate) instructions. If the Mac instruction is not supported, the matrix multiplication can be “assembled” by the user Arm SW by sequencing of 8xSIMD vector Add and Mult instructions. In this case, the MFLOP/s performance will be reduced by cca. 50% (Add and Mult operators are not chained), but the accelerator takes less HW resources and this might be critical in some platform configuration as the PL area is used by pre-defined HW accelerated video processing. The example code is reported in Appendix A.

The function also demonstrates how to allocate temporary data vectors in the physical continuous memory section of the DDR and how to free them before the exit from the function.

Reconfigure streaming data paths of serial connected accelerators

The accelerators can be connected in serial chains. This results in saved resources for HW data movers and enables also direct communication from an accelerator to a next accelerator in the chain. But such connection creates dependency of accelerators and run-time reconfiguration of the data path is needed to full-fill the needed tasks.

User can reconfigure in the run-time the accelerator streaming data path to reach these different functionalities:

1. Set all accelerators in the chain as “pass-through” with exception of one of accelerator, where the streaming data are used for
    a. Rd (read data from the selected accelerator to Arm DDR)
    b. Wr (write data to the selected accelerator from Arm DDR)
    c. Rd and Wr (perform a. and in the same time b. from the same 64 bit BRAM block or from a different 64 bit BRAM block)

2. Wr identical data to 2, 3 or 4 selected 64 bit BRAM blocks to all accelerators in the serial chain of accelerators.
3. Rd from one selected accelerator and write data to another selected accelerator located as one of next accelerators in the chain.

4.3.6 Monitoring of V-PCC in Virtual Reality

Point clouds for immersive media technology have received substantial interest in recent years. Such representation of 3D scenery provides freedom of movement for the viewer. However, transmitting and/or storing such content requires large amount of data and it is not feasible on today's network technology. Thus, there is a necessity for having efficient compression algorithms in order to facilitate proper transmission and storage of such content.

Recently, projection-based methods have been considered for compressing point cloud data. In these methods, the point cloud data are projected onto a 2D image plane in order to utilize the current 2D video coding standards for compressing such content. These video-based point cloud compression (V-PCC) schemes can provide significant improvement over state-of-the-art methods in terms of compression efficiency.

Monitoring Requirements

The main advantage of the selected V-PCC approach is its compatibility with current 2D video coding standards. As 2D video coding standards, such as High Efficiency Video Coding (H.265 / HEVC), are already supported by billions of devices and distribution solutions, thus it is possible to integrate this type of solution in the current products and services.

In addition, the V-PCC system provides remarkable bitrate reduction compared to reference technology in the terms of both objective and subjective quality. Bitrate requirements are reduced by around 75% for geometry and approximately 50% for colour attribute over the state-of-the-art compression technology. The important bitrate reductions can be achieved by new type of algorithmic solutions.

Figure 32 illustrates block diagram of overall process of the V-PCC system. A brief description of the block diagrams that are used in the V-PCC technique is provided as follows:

- 3D to 2D projection: Projecting each individual point cloud of a sequence onto the 2D geometry. One 2D plane is allocated for texture projections and one for geometry.
- 2D encoding: The 2D planes (geometry and texture) are encoded with the current standard 2D video codecs (e.g. HEVC).
- 2D decoding: The encoded 2D planes are decoded with current standard 2D video codecs.
- 3D to 2D projection: The reconstructed 3D scene can be created by using decoded planes. In back-projection process, texture plane is used for colour

intensity value of point and the position of point is determined by corresponding geometry value.

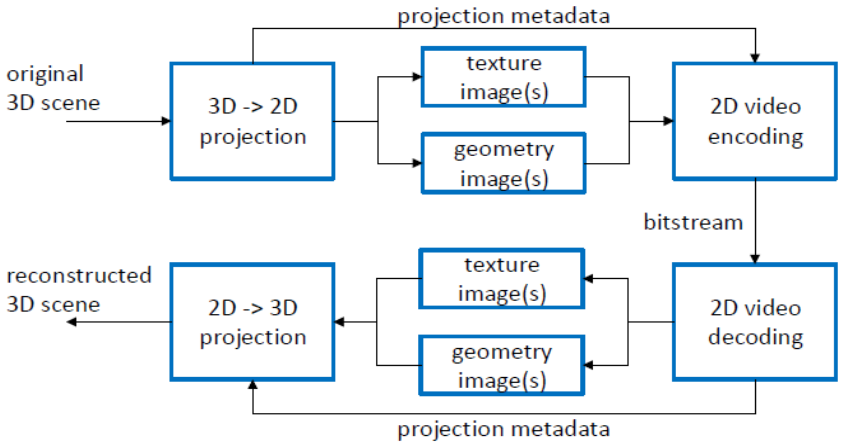

Figure 32: Projection-based volumetric video coding workflow in V-PPC system.

Figure 33 illustrates subjective performance of the V-PCC method compared to the reference technology. As can be seen, the V-PCC system outperforms the reference technology for compressing the point cloud data significantly.

Figure 33: (left) Original point cloud (middle) decoded point cloud at 13 Mbit/s for V-PCC system, and (right) reference technology.

The necessity to monitor the runtime state of the system provides the monitoring requirements. In particular, it is required to monitor the system performance with the following metrics:

- Near real-time (soft real-time) performance in terms of frames-per-second and Megabits-per-second.
- The energy usage for the whole V-PCC system.

Units Under Monitoring

V-PCC Decoding Performance

In typical 2D video player applications, only one video stream is decoded and displayed. Some stereo and VR stereo 360 video players handle and synchronise two video tracks, one track for each eye. However, V-PCC decoding requires the synchronisation of three video decoder instances. Unfortunately, current Android and iOS video decoders do not support adequate synchronisation of video streams. Video bitstreams are decoded on a “best effort" basis, depending on the available processing resources.

The possible frame-skipping is a serious problem for V-PCC, as all three video frames are needed for 3D reconstruction. In typical 2D video display there is not much need for the application to know if a frame was skipped and displayed twice in a row. Especially, as the human eye adapts to such an event as it happens only rarely. For V-PCC decoding this is however a serious error, as it will destroy the complete 3D reconstruction. To avoid this issue and guarantee a high-quality V-PCC playback, sophisticated frame buffering is essential. Table 10 provides an overview of V-PCC decoding performance on current mobile hardware, e.g. decoding and synchronising three individual video decoder instances.

Table 10: V-PCC decoding performance overview

DEVICE	GPU / CHIPSET	FPS
iPhone 6S	Apple A9 GPU	15.4
iPhone 8	Apple A11	30.3
iPhone X	Apple A11	30.3
iPhone XS	Apple A12 GPU	30.3
Samsung Galaxy S10	Mali-G76 / Exynos 9820	25.0
Huawei Mate Pro 20	Mali-G76 / Kirin 980	23.3
Google Pixel	Adreno 530 / Snapdragon 821	29.4
Google Pixel 2 XL	Adreno 540 / Snapdragon 835	30.3
Nokia 8 Sirocco	Adreno 540 / Snapdragon 835	30.3
Google Pixel 3	Adreno 630 / Snapdragon 845	25.6
OnePlus (A6013)	Adreno 630 / Snapdragon 845	25.6

V-PCC AR Rendering Performance

Decoding the separate video streams is just one part of the V-PCC decoding pipeline. In order to fully evaluate how well a V-PCC standard can be implemented on current generation mobile devices, the V-PCC test model was modified and ported to Android and iOS platforms. A very minimalistic point cloud reconstruction process was selected, without any further postprocessing such as 3D smoothing. A GPU accelerated version of the 3D point reconstruction was implemented based on OpenGL ES 3.0. Table 11 summarises the achieved rendering capabilities on various mobile phones. The rendering performance is consistent between devices with the same GPU (see Table 10).

Table 11: V-PCC AR point rendering capabilities.

Release year	Device	GPU	Mpoints/frame @ 60 fps
2014	Samsung Galaxy Note 4	Adreno 420	1.04
2016	Samsung Galaxy S7 Edge	Mali-T880	0.43
2016	Google Pixel	Adreno 530	1.60
2017	Google Pixel 2	Adreno 540	1.81
2018/Q4	OnePlus 6T	Adreno 630	1.90
2018/Q4	Huawei Mate Pro 20	Mali-G76	1.32

The 3D point reconstruction is calculated inside a vertex shader to save vertex memory bandwidth. Output of the shader is a uniformly distributed 1-pixel sized points, filling the display of the mobile device. In order to measure peak performance, the video dimensions (width, height) were increased to the point until the decoding performance drops below 57 fps. Typically, mobile devices can only sustain maximum performance for a short period of time. In order to measure more reliable benchmark results, the rendering benchmark is kept running for ten minutes and the overall average is reported as how many million points can be rendered in real-time (60 fps).

Typical V-PCC content currently consists of around 1 Million points per frame. Looking at the results from Table 10 and Table 11, it can be seen that even 3-year-old hardware is already capable of decoding and rendering V-PCC bitstreams.

Current Nokia's V-PCC player can decode and render MPEG V-PCC coded bit streams in real-time. An early version of Nokia's V-PCC application was presented as VIP internal Nokia demo at the Mobile World Congress 2019 in Barcelona. The current version has been showed at the International Broadcasting Conference (IBC2019, September 13-17, 2019) in Amsterdam. The V-PCC playback application source code has been made available to the public as reference: Nokia Technologies, "Video Point Cloud Coding (V-PCC) AR Demo," https://github.com/nokiatech/vpcc

Monitoring infrastructure

The runtime state of the system includes measured performance, which can be handled by a generic data model. Relevant metrics to be monitored/evaluated are the following:

- Near real-time (soft real-time) performance: System performance can be evaluated in terms of frames-per-second and Megabits-per-second. It is worth noting that system robustness/performance/quality depends highly on the selected computational algorithms. We have made our development work using state-of-the-art methods to ensure that possible use cases can be implemented easily in normal phone platforms. Later we plan to investigate more advanced computational models that may require more optimization of the system code to achieve more performance in the system level.

It is not an easy task to calculate the energy usage for the whole V-PCC system, since continuous computational load and required advanced algorithms will present a challenge in terms of optimizing the energy usage of the system as a whole. Thus, we have so far planned only initial measurements on power usage and based on the achieved initial results. We will try to adjust the implemented algorithms to enable optimal energy usage of V-PCC system in the last year of the FitOptiVis project.

4.3.7 Monitoring in Salmi-Care System

HURJA's AR-based (Augmented Reality) Salmi Care Platform is capable of motivating rehabilitation patients to make daily exercises by utilizing AR-based gamification techniques, assisting rehabilitation patients & brain damage patients & elderly people in their daily tasks, and monitoring daily activities & vital signs of patients as well as automatically alerting nurses/relatives in case of emergency. Rehabilitation patients, brain damage patients, and elderly people are wearing AR-glasses (HoloLens II) as User Interface for Salmi Care service. Service can be used via voice commands and/or gestures. Service also enables patients to communicate with nurses/doctors/relatives/peers via video calls that are directly shown on AR-glasses.

Monitoring Requirements

The runtime state of the system includes measured performance and energy usage, which can be handled by a generic data model. Relevant metrics to be monitored/evaluated are the following:

- Near real-time (soft real-time) performance: System performance was monitored/evaluated in terms of frames-per-second and kilobits-per-second. It is worth noting that AR-feature robustness/performance depends highly on the selected AR-glass model. We have made our development work using state-of-the-art HoloLens 2 AR-glasses to ensure that all possible use cases can be implemented easily. Later on we plan to investigate the use of other (cheaper and less powerful) AR-glass options that may require more optimization of the system code to achieve the level of performance comparable with the high-end, state-of-the-art AR-glasses.

- Optimal energy usage: It is not an easy task to calculate the energy usage for the whole Salmi Care system, since continuous camera feed and required advanced algorithms will present a challenge in terms of optimizing the energy usage of the system as a whole. Thus, we have so far performed only initial measurements on power usage and based on the achieved initial results, we will make adjustments to the implemented algorithms to enable optimal energy usage of Salmi Care system.

Furthermore, the system monitors the achieved level of satisfaction of all end-user groups that can be handled by a generic data model:

- The intended users of the Salmi Care system will be rehabilitation patients (assisted living), brain damage patients (assisted living), elderly people (assisted living), relatives (monitoring and situational awareness), nurses (home visits), and doctors (emergency cases). We have made careful plans to achieve the required level of satisfaction for all of these end-users of our Salmi Care system. However, we cannot yet completely fulfill all of the below-mentioned end-users requirements or all the needed features, but by the end of the project, we will have fully functional version of Salmi Care system that fulfils the level of satisfaction for all of these end-user groups.

Unit Under Monitoring

For achieving near real-time (soft real-time) performance on our low-power mobile AR-based Salmi Care Platform we have utilized smart feature extraction, segmentation, and classification algorithms to reduce bandwidth usage by only sending the necessary parts of images/videos. A mobile application called Extent can upon request download a JSON packet which consists of a list (descriptions) of wakeup images, objects, entities, and actions. Either the request can come from the Salmi MAPS website, from the Salmi Care mobile application, or directly from the Extent mobile application if the “free roam” state has been switched on (requires GPS). End-users have the option to switch the “free roam” state off at any time and when this happens, the Extent mobile application downloads new content only upon request from an external source (currently only the Salmi Care Platform related sources are available). The Extent mobile application downloads all required wakeup images, 3D-models, textures, audio files, videos, etc. based on the instructions received via JSON packet.

To optimize the run-time performance of the Salmi Care Platform all of these packets can be downloaded in advance. All files will be saved locally into end-users’ mobile device (smart phone or tablet) and those will be shown to end-users based on instructions received via JSON packets as soon as matching wakeup image, object, entity, or action has been found, or when an end-user is within a certain pre-defined distance from the target. Free roam data will be removed on-the-fly from end-users’ devices when each session ends. The Extent mobile application is currently being developed using C# programming language on top of the Unity 3D engine and the server back-end side is currently being developed using PHP. During our early testing phase, all description packets are in JSON format.

Monitoring Infrastructure

As of M24, the status of implementation of these monitoring data features is as follows:

- *Monitoring of performance of rehabilitation patient's daily exercises:* Rehabilitation patient's daily exercises are monitored and related data is collected for further analysis in order to determine how effective training is for each patient. The variables under monitoring are total duration of the exercise, the duration of each individual sub-session inside the exercise session, and amount of correct/incorrect actions made during the exercise session. Data will be sent to the cloud server for further analysis in real-time during the exercise session.
- *Monitoring of application performance:* Application performance will be monitored actively and most important target will be refresh frequency of the application that represents in the high level how well application works. Refresh frequency will be measured as Frames-Per-Second (FPS) and in case of HoloLens II AR-glasses it is 60 FPS. Especially for AR-based applications it is very important that FPS will be at least 60 all the time so that the user experience of AR-world is as fluent and as convenient as possible. Another variable used for application performance monitoring is the usage of RAM (Random Access Memory), but it is not as important as FPS-monitoring since in rehabilitation application there are only few really heavy operations in terms of RAM usage. However, different operating systems will react differently to the situation when RAM runs out and thus in case of HoloLens II we have to make sure that the application never uses all the RAM in any circumstances to make sure the application remains stable and usable all the time. The application performance is monitored by utilizing the real-time development platform Unity's own tools (see Figure 34). We are able to monitor, by using Unity's own performance monitoring tool, the following variables during each frame:
  - CPU: Calls, Garbage Collection Allocations, Time ms, and Self ms.
  - Rendering: SetPass Calls, Draw Calls, Total Batches, Triangles, Vertices, Used Textures (amt + memory usage), VRAM Usage, and Shadow Casters.
  - Audio: Total Audio Sources, Playing Audio Sources, Paused Audio Sources, Audio Clip Count, Audio Voices, Total Audio CPU usage (%), DSP CPU usage (%), Streaming CPU usage (%), Other CPU usage usage (%), Total Audio Memory (MB), Streaming File Memory usage (MB), Streaming Decode Memory usage (MB), Sample Sound Memory usage (MB), and Other Memory usage (MB).
  - Memory: Texture/Mesh/Material/Animation/Audio/GC Memory usage and In-Depth Analysis can be seen with Unity's memory snapshot tool:

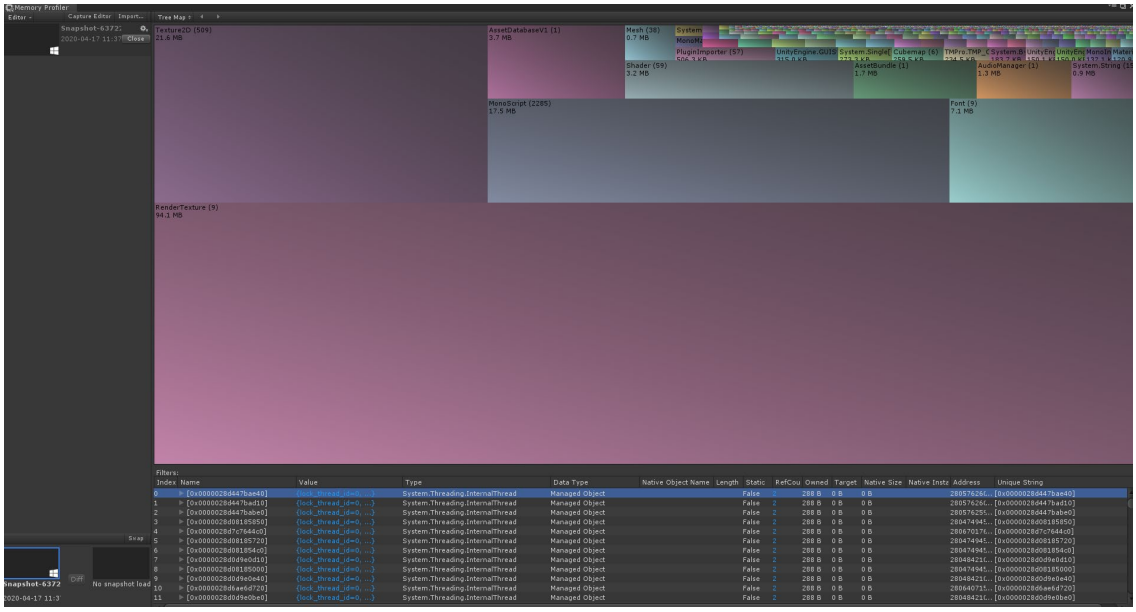

Figure 34: Unity based tool for application performance monitor.

Data Storage, Analytics and Visualization

Collected data will be stored in secure servers. Analytics and visualization will be done utilizing appropriate analytics/visualization tools, such as Microsoft Power BI and AWS Analytics/Visualization services. Statistics of the users include the amount of correct and incorrect actions, duration of each action, and total duration of the exercise session. Data from previous exercise sessions can be used to keep track and compare how the user has progressed in the rehabilitation.

4.3.8 TSN support for concurrent monitoring of multiple heterogenous systems

Monitoring infrastructures provided by TSN

Best effort, lowest priority TSN streams will be provided to collect monitoring information for both Habit Tracking and Surveillance for Smart Grid critical infrastructure. These traffics will be isolated from payload traffic, such control communication between distributed processing nodes or from time critical messages.

Moreover, timestamping support is provided to distributed nodes under monitoring to facilitate coherent processing and the understanding of collected data. Timestamping is provided by means of the generalized Precision Time Protocol (gPTP).

TSN internal monitoring

The Time Sensitive Networking bridge for FitOptiVis is a Xilinx Zynq-7000 based platform, composed by FPGA logic and software. TSN provides convergence of mixed critical traffics relying on stringent time synchronization. For this reason, runtime

monitoring of gPTP provides information about self-capability and network-wide capability of delivering RT-QoS.
As well as other protocol aspects, the different metrics to be delivered on runtime monitoring are defined on IEEE 802.1AS:

- Current time deviation. The current synchronization deviation is computed at every arrival of Sync messages generated by the elected grandMaster. This information is used by time-critical applications to verify the enabling conditions of deterministic communication. Unusual time deviations can be used to detect abnormal functionality of the network components.
- Link delay. The link delay is used by the synchronization protocol to recover the remote network time reference accurately. The link propagation delay is computed periodically to maintain the synchronization accuracy isolated from propagation delay variations. The link delay is also useful to estimate E2E latency for time-critical traffics.
- RateRatio. Frequency relationship between the network time reference and the local clock stored on the PTP Hardware Clock.
- AsCapable Interface. The AsCapable flag is associated to each time-sensitive interface and reports the synchronization capability of the remote peer. A remote node not supporting gPTP cannot be considered for grandMaster election and cannot support deterministic forwarding.
- Current grandMaster and synchronization path. The result of the BMCA is returned to the end user to check the synchronization network status. It is useful to indirect see the status of remote elements
- Port role. The BMCA also determines the functionality of each active interface in the time-aware system under monitoring. The slave interface is the one closest to the grandMaster and provides synchronization to the system. Passive ports also receive synchronization information and back the passive port in case of failure. Master ports are present on bridges and retransmit the synchronization information received from the GrandMaster. Finally, ports maybe also disabled by the user or due to network failures.
- Network status. This information is related to local PHY layer and gives information about inner hardware status.

Besides, other monitors have being considered to track the runtime of the TSN bridge (i.e. network status).

Unit Under Monitoring

The primary scope of gPTP is to obtain time offset and frequency deviations between the local PTP Hardware Clock (PHC) and the remote time reference (grandMaster). However, link delays and gPTP residence times should also be tracked. The current network time reference or grandMaster and the synchronization path linking this node to the TSN bridge under monitoring is also available to check the network status. This runtime information the basis to detect abnormalities and provide fast failover.

Monitoring Infrastructure: The Timestamping Unit (TSU)

All the quantitative metrics are based on deterministic time references taken at the egress and ingress of gPTP event messages. Such determinism is key for synchronization accuracy and is enabled by hardware timestamping located closed to

the physical medium. In this implementation, the hardware timestamping is located at the Medium Independent Interface, isolating time synchronization from the variability introduced by MAC, Bridge and higher Ethernet layers.

The hardware timestamping unit (TSU) is continuously tracking the MII interface and fetches the local clock time from PHC whenever a start of frame (SoF) delimiter is transferred. The TSU delivers the software processor ingress (Rx timestamp) and egress (Tx timestamp) times for gPTP messages along with their FCS to allow matching between messages and timestamps on gPTP protocol state machines implemented on software (see Figure 35).

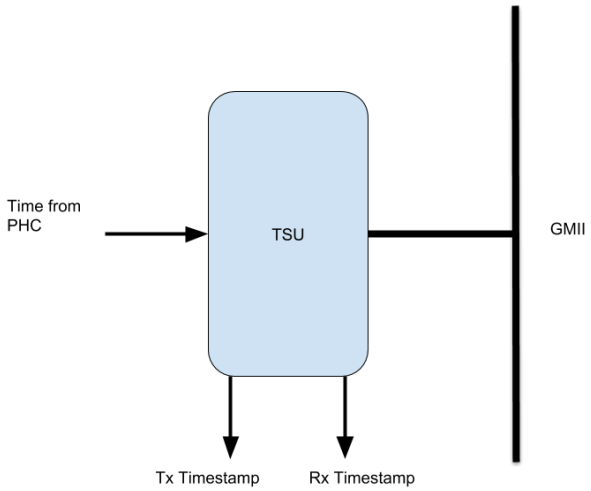

Figure 35: Timestamping Unit

Furthermore, gPTP defines the protocol mechanisms enabling the computation of the current link delay and deviation between local clock and grandMaster clock.

Propagation delay measurement

The propagation link delay for full-duplex, point-to-point links is computed following the Peer delay mechanism. This is based on a protocol handshake performed periodically between every two adjacent time-aware stations and is present on every active interface. Peer delay mechanism is shown in Figure 36.

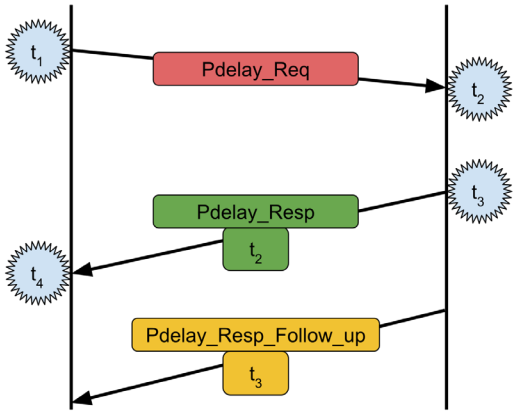

Figure 36: Regular Handshake on the Peer Delay Mechanism.

The left side of the link acts as peer delay initiator and the right as peer delay responder. From the message interchange, four timestamps are captured (t1, t2, t3, t4) and delivered to the gPTP executable at the initiator side, which computes the link delay following the equation:

$$D = \frac{(t_4 - t_1) + (t_3 - t_2)}{2}$$

Periodical computation of the link delay allows not only detect propagation delay changes, but also estimate the relationship between local clock frequencies of two adjacent time-aware systems (neighborRateRatio), by considering successive Pdelay_Resp and Pdelay_Resp_Follow_Up messages. The relation between local clock and grandMaster clock frequencies (RateRatio) can be derived from successive neighborRateRatio computations along the synchronization path. The RateRatio is used to reference remote timestamps to the local clock and obtain coherent time estimations.

Two-step PTP mechanism

IEEE 802.1AS implements a two-step PTP to recover the current remote time reference (see Figure 37). A Sync message is generated by the grandMaster and retransmitted by every time-aware bridge along the synchronization path. The follow-up message carries the originTimestamp (i.e. the Sync egress timestamp on the grandMaster) and the correction Field or propagation time until the Sync message is timestamped at the ingress of the time-aware system of interest. This propagation time is the sum of every link propagation delay and residence times on the synchronization path.

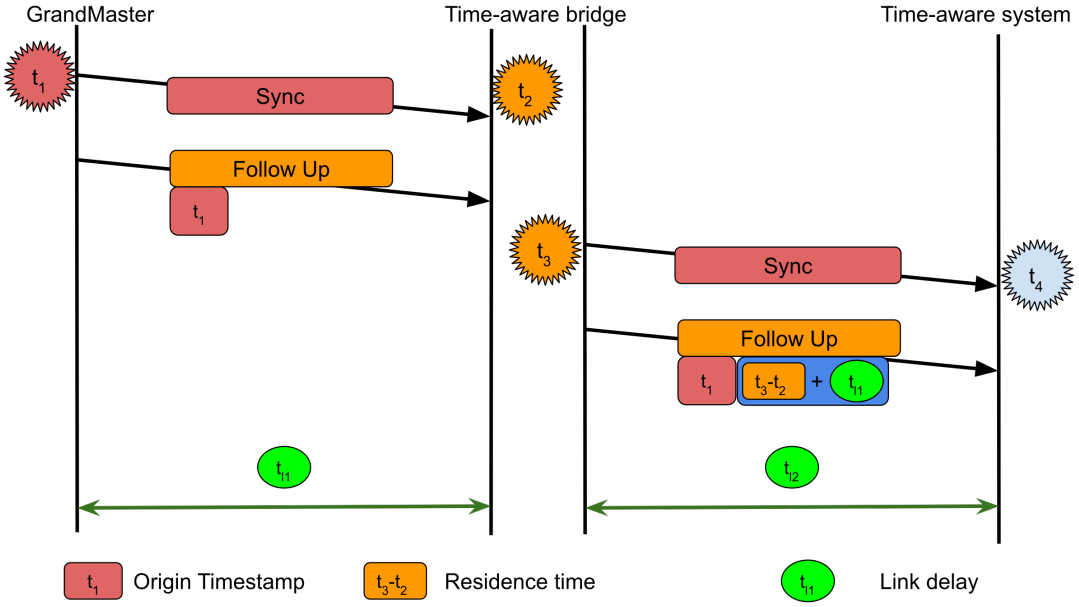

Figure 37: Two-step PTP mechanism, as defined on IEEE 802.1AS

Best Master Clock Algorithm (BMCA) monitoring

The elected grandMaster and the synchronization path give qualitative information about the inner quality of the remote time reference and the nodes participating on the propagation of the Sync message. This information is maintained by the Best Master Clock Algorithm executed on every time-aware system in the network. Finally, the BMCA also determines the port role of the time-aware system.

Data Storage, Analytics and Visualization

The TSN User API delivers these monitors to the end user. A periodic task is executed on the ARMv9 present on the Xilinx Zynq-7000 MPSoC to retrieve monitoring periodically. Runtime monitoring is delivered to a central Set-Top-Box by a Best-effort TSN stream. Monitors from all TSN stations are stored and available for presentation in the FIVIS platform.

4.3.9 Monitoring systems for localization in space applications

The Autonomous Exploration use-case is focused on the reconfiguration of a video processing chain on board of a spacecraft designated for locating different kinds of satellites. Space missions have several stages that are very different in terms of performance and environmental conditions. The target is to include into the UC a set of monitors that ease the differentiation of the stages of the mission through power

consumption and radiation monitoring. In the use-case, besides the monitors developed by TASE, the ones developed by University of Cantabria will also be used.

Monitoring Requirements

The monitors developed by TASE will focus mainly on the hardware side of the complex video-processing chain. The two main metrics to measure at runtime in order to control the reconfiguration mechanisms will be:

- Radiation dose: components off the shelve are currently being used in a lot of space missions. These kinds of components are not radiation hardened by design. It is really important to monitor the radiation induced failures on them in order to keep functionality of the designs. When a high dose of radiation is received by the components (in this case an FPGA) a full reconfiguration of the system shall be done.
- Power consumption and temperature: another important driver for reconfiguration during a mission is the power consumed by the platform. There are several power constraints in space due to the lack of refuelling and the limited amount of power delivered by the solar panels on a spacecraft. For example, it is very important to minimize power consumption during shade-phases of a mission. Power consumption will be monitored in order to verify that reconfiguration has been performed successfully and that changes in the configuration drive the consumed power to the desired values constrained by the phases of the mission.

Additional metrics and monitors will be potentially taken into consideration in the future regarding video-processing performance complementing those developed by the University of Cantabria.

Unit Under Monitoring

The unit under monitoring will be the FPGA Logic of the Zynq UltraScale+ MPSoC.

Monitoring Infrastructure and Monitoring Processor

The monitoring infrastructure consists of three basic components that will act as building blocks.

- SEM IP: Soft Error Mitigation IP. This IP is provided by Xilinx and has already been integrated on the platform. It allows to simulate the failures induced by radiation thanks to an *ad hoc* interface developed by TASE. This interface allows to reproduce different radiation doses that are related with several orbits.
- System Monitor: The System monitor is an interface present on the silicon of the FPGA that allows to keep track of the power and temperature of the system under test. It is implemented by default by Xilinx
- Virtual Input/Outputs.I/Os: This component consists on inputs and outputs that can be controlled from a GUI and allow to have information from the processing logic of the MPSoC. Thes I/Os have to be implemented on each specific components containing the desired information to monitor.

4.3.10 Pose and facial recognition in Habit Tracking with edge-cloud adaptivity

As part of the effort in Habit Tracking use case, HIB is developing a cloud-edge AI solution for detecting activities of persons in their homes as well as matches of their facial features according to a database of potential users. The main goal is to track persons within their homes to detect if they present signs of mild cognitive impairment. Technically one of the key features is to mix in the home edge processing with cloud processing, where 'edge' corresponds roughly to smart cameras with low computational capabilities (typically ARM processors running on batteries) and the 'cloud' corresponds to x86 architectures in the home with no energy constraints.

In the following subsections we present the general outline of the adaptations to be applied to the demonstrator with the help of WP4 components.

Monitoring Requirements

The main overarching requirement is to maintain a specific overall set of recognition features while maximizing the usage of energy as some of the processing elements could be running on batteries.

The **functional** 'recognition features' are currently being specified but as of the writing of this document they are in summary:

- **Pose estimation engine**: using CMU openpose[1] which yields a wire-frame model of persons in still frames.
- **Facial recognition engine**: matches faces in still frames with known profiles of persons of interest who have been trained in the system.

The main **non-functional** requirement of the system is to maintain an overall processing of frames at a sufficient rate of frames per second that enables the detection of complex behaviours. For the purposes of this in the use case, the figure is around 15 frames per second.

Unit Under Monitoring

Figure 38 represents the overall architecture of the system under analysis.

[1] https://github.com/CMU-Perceptual-Computing-Lab/openpose - openpose: multiperson human body posture detection by Carnegie Mellon University.

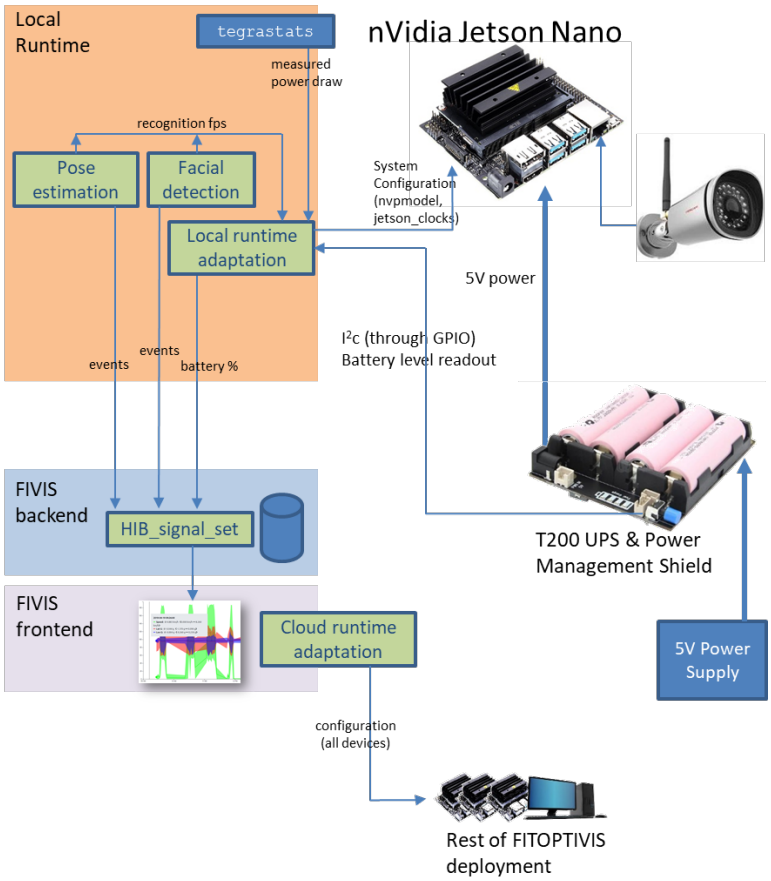

Figure 38 HIB architecture for UC3 Habit Tracking

The Figure depicts the most relevant unit under monitoring which is the edge processing device. In this case it is an Nvidia Jetson Nano single board computer (that integrates a multi core ARM CPU and a GPU focused on AI operations). For the adaptation purposes of the system the edge board(s) will be running not connecting to the mains power but using a dedicated UPS power supply using 18650 LiOn batteries. This is connected to the Jetson Nano by means of two wires: one (depicted in a thick edge) that supplies the nano of the required 5V/2A required for normal processing and another (depicted in thin edges) connecting a port in the UPS board to the GPIO pins in the Jetson Nano board. This, encoded in the industry-standard device-to-device protocol i²C[2], is used to monitor the current level of the onboard batteries.

In the living lab deployment under test by HIB this edge system is connected via a network connection to a Foscam FI9800P camera and also via network connections to the 'cloud' server and the Internet at large.

Monitoring Infrastructure

The overall infrastructure is depicted in Figure 38. In addition to the aforementioned hardware units (the Nvidia Jetson Nano and the GeekwormT200 UPS kit with i²C battery

[2] https://en.wikipedia.org/wiki/I%C2%B2C – Inter-Integrated Circuit protocol.

level monitoring), there is a ‘local’ adaptation engine running on the Nvidia board local environment as well as a ‘cloud’ adaptation engine running offline in a we server.

The local adaptation monitors the battery levels and the desired QoS parameters (chiefly the minimum fps). Combining the power draw that is required from the board’s components (collected using the `tegrastats` command line tool provided in the default OS for the board) and the available battery level in the UPS board (collected using i^2c polling on the appropriate port in the board), the system computes a battery life estimate. Whenever the battery estimate falls below the threshold set at design time, the local adaptation engine changes the execution environment for the feature recognition engines by tweaking the active cores (using the `nvpmodel` command line tool provided in the Ubuntu distribution for the Nvidia board) of the clock frequency of the cores and GPU (using the `jetson_clocks` command line tool).

Lowering the execution performance with these increases the estimated battery life. If by monitoring the performance we detect that it falls below the desired fps/QoS, then the system might transition to a different cloud/edge configuration. This is mostly done by the ‘cloud’ adaptivity system which is described in the following subsection.

Data Storage, Analytics and Visualization

The system continually collects values for the metrics of interest in the adaptation and the execution of the system (the most important of which are the fps for the recognition systems, the battery percentage from the UPS board). These are collected as a group and sent to the remote monitoring system which uses the FIVIS[ref] environment by CUNI as a unified signal set (called HIB_signal_set). In the FIVIS environment they are collected and can be analyzed later on by the system operators.

The overall performance of the system is managed by a joint ‘cloud’ adaptation system that is aware of all the computing elements in the deployment. Based on the configuration selected by the system operator, different ‘edge’ (Nvidia Jetson Nano boards) or ‘cloud’ (x86 PCs running a Linux environment) can be switched on and off on demand.

4.3.11 Monitor in Processor-Coprocessor systems

In a typical edge-computing scenario, there can be functional and strict non-functional requirements to be satisfied. This leads to heterogeneous platforms, and in Fitoptivis there are tools aimed at supporting in the development of these platforms. In this regard, MDC impacts in the development of processor to co-processors systems, offering a coarse-grained functional and non-functional reconfiguration (Section 3.3). In this section, the described runtime monitoring is part of the self-adaptive loop where the MDC reconfiguration acts: the monitoring systems are generated by using the AIPHS framework (Section 4.2.3). The framework to use AIPHS applied to MDC generated coprocessors can be accessed at the repository in [AIP20].

Monitoring Requirements

Being at the edge, it has to be considered that the impact of monitoring actions on non-functional parameters needs to be limited. On the other hand, monitoring of the current execution of the system is necessary to properly trigger the reconfiguration.

The following monitoring requirements are given when the coprocessor systems have to be monitored:

- MON1 - Limited SW overhead
- MON2 - Measure of accelerator latency
- MON3 - Measure of accelerator performance
- MON4 - Runtime verification of the accelerator

Unit Under Monitoring

The Unit Under Monitoring is given by MDC, which is able to deploy a processor-coprocessor system according to the user choice:

1. Type of processor: hard-core (ARM available on Zynq7000 FPGAs) or soft-core (Microblaze).
2. Processor-coprocessor coupling: stream based or memory mapped.
3. Use of DMA.

At the moment, the Unit Under Monitoring is given by the memory-mapped processor coprocessor system, in which the DMA is used. Both ARM and Microblaze are possible selections. Figure 17 shows the IP generated by MDC.

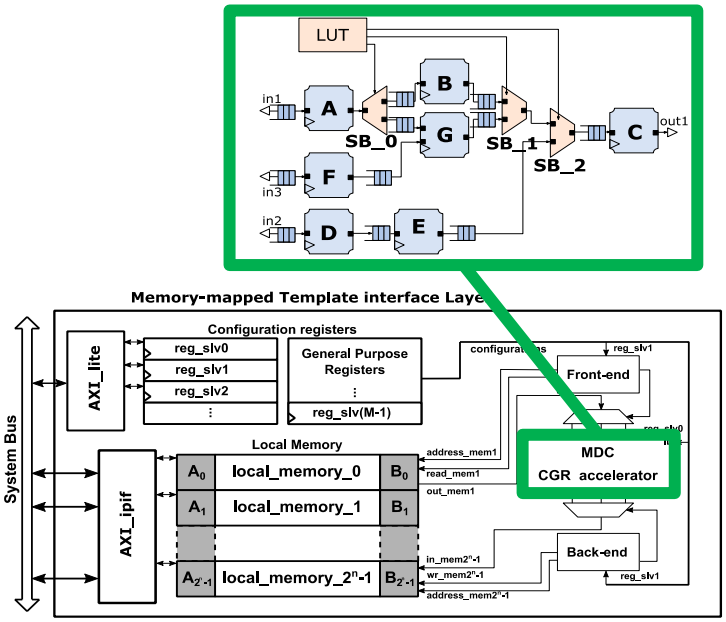

Figure 39 – The figure shows the IP generated by MDC. The MDC CGR accelerator is a coarse grain reconfigurable datapath capable of executing different functionalities, by opportunely multiplexing resources in time. This datapath is automatically encapsulated into a ready to use Xilinx IP and into a processor-coprocessor system, according to the user choices during the design time.

Monitoring Infrastructure

The monitoring infrastructure has been generated using the AIPHS framework: this allows to satisfy the MON1 requirement, since AIPHS produces hardware monitoring systems that limit the timing impact on SW execution. Three different sniffers can be selected to monitor the coprocessor. Figure 40 shows the sniffers and their placement with respect the internals of the accelerator. The sniffer at level 1 (red) monitors the processed data by the accelerator, by counting the number of writes on the AXI4-Full bus: this sniffer allows to satisfy MON3. The sniffer at level 2 (yellow) monitors the accelerator latency, allowing the satisfaction of MON2. Finally, the sniffer at level 3 (violet) allows to perform a runtime verification of internal transitions.

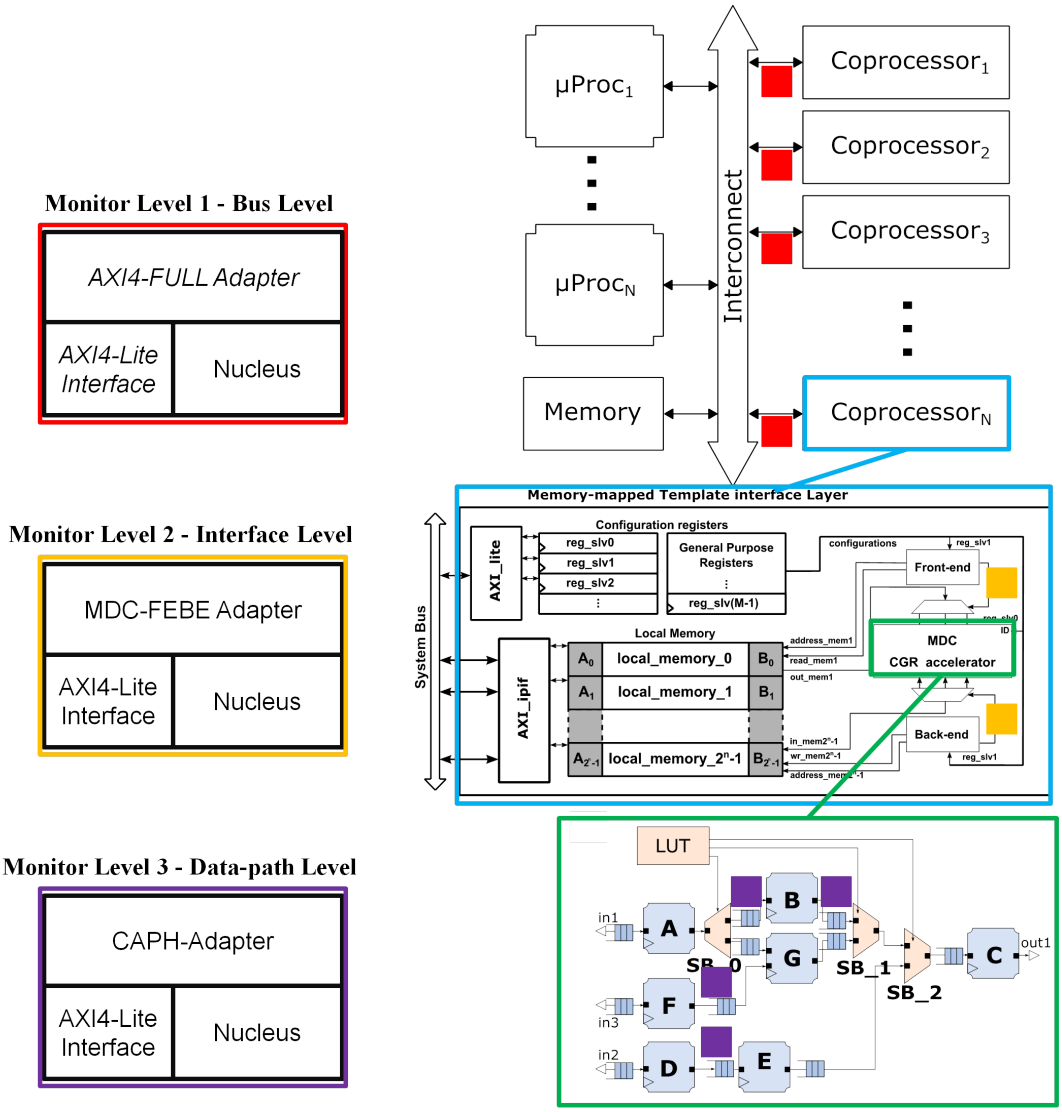

Figure 40: Sniffer generation for MDC coprocessors

Data Storage, Analytics and Visualization

Data output by the monitoring system are organized as reported in Table 12. EVENT_ATTRIBUTE contains an attribute for internal usage, ACC_ID provides a code to indicate the ID of the monitored coprocessor, LEVEL_ID indicated the monitored level, EVENT_INFORMATION contains the raw information.

Table 12: Event instances of monitors.

EVENT_ATTRIBUTE (5 BITS)	ACC_ID (4 BITS)	LEVEL_ID (2 BITS)	EVENT_INFORMATION (REMAINING BITS)

Further information, together with two working examples related to AIPHS for MDC, are reported in [AIP20]. Table 13 reports the resource utilization impact of the different monitor levels introduced in the Custom Multiplication example [AIP20].

Table 13: Resource utilization on Zynq7000 (Zedboard) for the Custom Multiplication Example [AIP20]

Configuration	Enabled Monitor Level			Purpose	LUTs	FFs
	1st	2nd	3rd			
Y0	-	-	-	-	3397	2864
Y1	X	-	.	Accelerator Performance	+8.18 %	+10.44 %
Y2	-	X	-	Accelerator latency	+2.94 %	+7.96 %
Y3	-	-	X	Runtime verification	+2.38 %	+6.25 %

5. **Conclusions**

In our summary of the outcomes of Task 4.2 and Task 4.3 from the first two years of the project, we deal primarily with two aspects of runtime support for adaptive applications developed using the FitOptiVis approach.

The first aspect concerns runtime reconfiguration, where different instances of reconfiguration mechanisms have been proposed; their combination gives rise to three main categories of reconfiguration mechanisms. In turn, this abstraction enables the development of an abstract view on how to use the different mechanisms in the quality and resource management framework.

The second aspect concerns monitoring, profiling and measuring support, where different enabling technologies and instances of monitoring mechanisms have been proposed. Monitoring techniques can span at different levels, so to unify at level of concepts, principles and abstractions to find and extract commonalities among different domains, a reference platform for monitoring in FitOptiVis has been also proposed.

In the following year, we will focus (i) on refining the proposed mechanisms, by spending effort on closing the adaptation loop that exploits the monitor and reconfiguration to adapt a system, and (ii) on developing practical setups related to FitOptiVis use cases.

These activities will result in a second iteration of this deliverable, which will incorporate outcomes from the third year of the project in deliverable D4.4 due in month 30 of the project (end of November 2020).

References

[CHAR13] I. Charfi, J. Miteran, J. Dubois, and M. Atri, “Optimized spatio-temporal descriptors for real-time fall detection: Comparison of support vector machine and Adaboost-based classification Network on Chip (NoC) View project Wireless ECG Patch View project Optimised spatio-temporal descriptors for real-time fall detection : comparison of SVM and Adaboost based classification,” Artic. J. Electron. Imaging, 2013.

[KAY17] W. Kay, J. Carreira, K. Simonyan, B. Zhang, C. Hillier, S. Vijaya-narasimhan, F. Viola, T. Green, T. Back, P. Natsev et al., “The kinetics human action video dataset,”arXiv preprint arXiv:1705.06950, 2017.

[KUE11] H. Kuehne, H. Jhuang, E. Garrote, T. Poggio, and T. Serre, “HMDB: A large video database for human motion recognition,” in 2011 International Conference on Computer Vision, 2011, pp. 2556–2563.

[SIG16] G. A. Sigurdsson, G. Varol, X. Wang, A. Farhadi, I. Laptev, and A. Gupta, “Hollywood in Homes: Crowdsourcing Data Collection for Activity Understanding,” in European Conference on Computer Vision, 2016.

[SOO12] K. Soomro, A. R. Zamir, and M. Shah, “UCF101: A Dataset of 101 Human Actions Classes From Videos in The Wild,” 2012.

[YOS18] Y. Yoshikawa, J. Lin, and A. Takeuchi, “STAIR Actions: A Video Dataset of Everyday Home Actions,” arXiv Prepr. arXiv1804.04326, 2018.

[OH11] Oh, S., Hoogs, A., Perera, A., Cuntoor, N., Chen, C.-C., Lee, J. T., ... Desai, M. (2011). A large-scale benchmark dataset for event recognition in surveillance video. In CVPR 2011 (pp. 3153–3160). IEEE. https://doi.org/10.1109/CVPR.2011.5995586

[HAR19] Harvey Adam. LaPlace, J. (2019). MegaPixels: Origins, Ethics, and Privacy Implications of Publicly Available Face Recognition Image Datasets. Retrieved from https://megapixels.cc/

[DAL05] N. Dalal (2005), “INRIA Person Dataset,” http://pascal.inrialpes.fr/data/human/

[CTF20] The Diamon Group, “The Common Trace Format”, https://diamon.org/ctf/

[KOR13] Georgios Kornaros and Dionisios Pnevmatikatos. 2013. A survey and taxonomy of on-chip monitoring of multicore systems-on-chip. ACM Trans. Des. Autom. Electron. Syst. 18, 2, Article 17 (April 2013), 38 pages.

[AIP20] AVIS – AIPHS for FitOptiVis, https://gitlab.com/alkalir/avis-aiphs-for-fitoptivis.git

[ZAN18] Michele Zanella, Giuseppe Massari, Andrea Galimberti, and William Fornaciari. 2018. Back to the future: resource management in post-cloud solutions. In Proceedings of the Workshop on INTelligent Embedded Systems Architectures and Applications (INTESA ’18). Association for Computing Machinery, New York, NY, USA, 33–38. DOI:https://doi.org/10.1145/3285017.3285028

[CAR13] N. Carta, C. Sau, F. Palumbo, D. Pani, L. Raffo, “A Coarse-Grained Reconfigurable Wavelet Denoiser Exploiting the Multi-Dataflow Composer Tool”, Conference on Design and Architectures for Signal and Image Processing, 2013.

[SAU17] C. Sau, F. Palumbo, M. Pelcat, J. Heulot, E. Nogues, D. Menard, P. Meloni, L. Raffo, “Challenging the Best HEVC Fractional Pixel FPGA Interpolators with Reconfigurable and Multi-frequency Approximate Computing”, IEEE Embedded Systems Letters, 9 (3), pp. 65-68, 2017.

[KAD19-1] J. Kadlec, Z. Pohl, L. Kohout: “Two serial connected evaluation versions of FP03x8 accelerators for TE0820-03-4EV-1E module on TE0701-06 carrier board” http://sp.utia.cz/index.php?ids=results&id=te0820_fp03x8x2s

[KAD19-2] J. Kadlec, Z. Pohl, L. Kohout: “FP01x8 Accelerator on TE0726-03M” http://sp.utia.cz/index.php?ids=results&id=te0726_fp01x8

[XIL12] Xilinx, Fast Simplex Link (2012), https://www.xilinx.com/support/documentation/ip_documentation/fsl_v20/v2_11_f/fsl_v20.pdf

[ARM10] ARM, AMBA Advanced-Peripheral Bus (2010), https://static.docs.arm.com/ihi0024/c/IHI0024C_amba_apb_protocol_spec.pdf?_ga=2.72731027.1882841013.1590093400-1278468236.1588607760

[ARM19] ARM, AMBA Advanced eXtensible Interface (2019), https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf

[XIL16] Xilinx, Local Memory Bus (2016), https://www.xilinx.com/support/documentation/ip_documentation/lmb_v10/v3_0/pg113-lmb-v10.pdf

[ARM12] ARM, AMBA Advanced High-Performance Bus (2015), https://static.docs.arm.com/ihi0033/bb/IHI0033B_B_amba_5_ahb_protocol_spec.pdf

Appendix – Example codes

The following example code is related to the Concrete Example Scenario reported in Section 4.3.5. It is reading license of the accelerator and its capabilities. Function is called by Arm in the Zynq or Zynq Ultrascale+ device. Function temporarily stores part of the current accelerator context, downloads and perform new program to get the needed information and finally restores the original state of the accelerator. See [KAD19-1], [KAD19-2] for details.

```
unsigned int test_license_0(fp03x8 fp03x8_0_inst, fp03x8 fp03x8_1_inst){
        unsigned int license_data;
        wide_dt *src_Z1_Z2_tmp = (wide_dt *) sds_alloc_non_cacheable(1*sizeof(wide_dt));
        wide_dt *src_Z3_Z4_tmp = (wide_dt *) sds_alloc_non_cacheable(1*sizeof(wide_dt));
        wide_dt *src_Z5_Z6_tmp = (wide_dt *) sds_alloc_non_cacheable(1*sizeof(wide_dt));
        wide_dt *src_Z7_Z8_tmp = (wide_dt *) sds_alloc_non_cacheable(1*sizeof(wide_dt));
        wide_dt *dest_Z1_Z2_tmp = (wide_dt *) sds_alloc_non_cacheable(1*sizeof(wide_dt));
        wide_dt *dest_Z3_Z4_tmp = (wide_dt *) sds_alloc_non_cacheable(1*sizeof(wide_dt));
        wide_dt *dest_Z5_Z6_tmp = (wide_dt *) sds_alloc_non_cacheable(1*sizeof(wide_dt));
        wide_dt *dest_Z7_Z8_tmp = (wide_dt *) sds_alloc_non_cacheable(1*sizeof(wide_dt));
        wide_dt_prog *src_P1_P2_tmp =
                        (wide_dt_prog *) sds_alloc_non_cacheable(1*sizeof(wide_dt_prog));
        wide_dt_prog *src_P3_P4_tmp =
                        (wide_dt_prog *) sds_alloc_non_cacheable(1*sizeof(wide_dt_prog));
        wide_dt_prog *dest_P1_P2_tmp =
                       (wide_dt_prog *) sds_alloc_non_cacheable(1*sizeof(wide_dt_prog));
        wide_dt_prog *dest_P3_P4_tmp =
                        (wide_dt_prog *) sds_alloc_non_cacheable(1*sizeof(wide_dt_prog));
        wide_dt_prog *src_P1_P2_lic =
                        (wide_dt_prog *) sds_alloc_non_cacheable(1*sizeof(wide_dt_prog));
        wide_dt_prog *src_P3_P4_lic =
                        (wide_dt_prog *) sds_alloc_non_cacheable(1*sizeof(wide_dt_prog));

        wide_dt_prog *dest_P1_P2_lic =
                        (wide_dt_prog *) sds_alloc_non_cacheable(1*sizeof(wide_dt_prog));
        wide_dt_prog *dest_P3_P4_lic =
                        (wide_dt_prog *) sds_alloc_non_cacheable(1*sizeof(wide_dt_prog));
        wide_dt_prog *src_P1_P2_out =
                        (wide_dt_prog *) sds_alloc_non_cacheable(1*sizeof(wide_dt_prog));
        wide_dt_prog *src_P3_P4_out =
                        (wide_dt_prog *) sds_alloc_non_cacheable(1*sizeof(wide_dt_prog));
        wide_dt_prog *dest_P1_P2_out =
                        (wide_dt_prog *) sds_alloc_non_cacheable(1*sizeof(wide_dt_prog));
        wide_dt_prog *dest_P3_P4_out =
                        (wide_dt_prog *) sds_alloc_non_cacheable(1*sizeof(wide_dt_prog));

        // Read P from accelerator 0 (accelerators 1 is set to see through)
        //=====================================================

        we = 0x0000;            fp03x8_we_write(&fp03x8_1_inst, we);
        bram = 16;              fp03x8_bram_write(&fp03x8_1_inst, bram);

        // save current program line 0
        //=====================================================
        paddr = 0;              fp03x8_paddr_write(&fp03x8_0_inst, paddr);
        we = 0x0000;            fp03x8_we_write(&fp03x8_0_inst, we);
        bram = 12;              fp03x8_bram_write(&fp03x8_0_inst, bram);

        data2hw_wrapper((unsigned*)src_P1_P2_tmp, 2*1);          //1
        capture_wrapper((unsigned*)dest_P1_P2_tmp, 2*1);         //2
        sds_wait(1);
        sds_wait(2);
```

```
paddr = 0;              fp03x8_paddr_write(&fp03x8_0_inst, paddr);
we = 0x0000;            fp03x8_we_write(&fp03x8_0_inst, we);
bram = 13;              fp03x8_bram_write(&fp03x8_0_inst, bram);

data2hw_wrapper((unsigned*)src_P3_P4_tmp, 2*1);          //1
capture_wrapper((unsigned*)dest_P3_P4_tmp, 2*1);         //2
sds_wait(1);
sds_wait(2);

// save current Z1 .. Z8 line 0
//=====================================================
paddr = 0;              fp03x8_paddr_write(&fp03x8_0_inst, paddr);
we = 0x0000;            fp03x8_we_write(&fp03x8_0_inst, we);
bram = 8;               fp03x8_bram_write(&fp03x8_0_inst, bram);

data2hw_wrapper((unsigned*)src_Z1_Z2_tmp, 2*1);          //1
capture_wrapper((unsigned*)dest_Z1_Z2_tmp, 2*1);         //2
sds_wait(1);
sds_wait(2);

paddr = 0;              fp03x8_paddr_write(&fp03x8_0_inst, paddr);
we = 0x0000;            fp03x8_we_write(&fp03x8_0_inst, we);
bram = 9;               fp03x8_bram_write(&fp03x8_0_inst, bram);

data2hw_wrapper((unsigned*)src_Z3_Z4_tmp, 2*1);          //1
capture_wrapper((unsigned*)dest_Z3_Z4_tmp, 2*1);         //2
sds_wait(1);
sds_wait(2);

paddr = 0;              fp03x8_paddr_write(&fp03x8_0_inst, paddr);
we = 0x0000;            fp03x8_we_write(&fp03x8_0_inst, we);
bram = 10;              fp03x8_bram_write(&fp03x8_0_inst, bram);

data2hw_wrapper((unsigned*)src_Z5_Z6_tmp, 2*1);          //1
capture_wrapper((unsigned*)dest_Z5_Z6_tmp, 2*1);         //2
sds_wait(1);
sds_wait(2);

paddr = 0;              fp03x8_paddr_write(&fp03x8_0_inst, paddr);
we = 0x0000;            fp03x8_we_write(&fp03x8_0_inst, we);
bram = 11;              fp03x8_bram_write(&fp03x8_0_inst, bram);

data2hw_wrapper((unsigned*)src_Z7_Z8_tmp, 2*1);          //1
capture_wrapper((unsigned*)dest_Z7_Z8_tmp, 2*1);         //2
sds_wait(1);
sds_wait(2);

//Define program to read license status program
//=====================================================
src_P1_P2_lic[0].p[0] = 0; //a_addr;
src_P1_P2_lic[0].p[1] = 0; //a_saddr;
src_P1_P2_lic[0].p[2] = 0; //a_inc;
src_P1_P2_lic[0].p[3] = 0; //b_addr;
src_P1_P2_lic[0].p[4] = 0; //b_saddr;
src_P1_P2_lic[0].p[5] = 0; //a_bank;
src_P1_P2_lic[0].p[6] = 0; //b_bank;
src_P1_P2_lic[0].p[7] = 0; //op;
src_P3_P4_lic[0].p[0] = 0; //b_inc;
src_P3_P4_lic[0].p[1] = 0; //z_addr;
src_P3_P4_lic[0].p[2] = 0; //z_saddr;
src_P3_P4_lic[0].p[3] = 0; //z_inc;
src_P3_P4_lic[0].p[4] = 0; //cnt;
src_P3_P4_lic[0].p[5] = 0; //z_bank;
src_P3_P4_lic[0].p[6] = 0; //no_op;
src_P3_P4_lic[0].p[7] = 0; //no_op;

// Write first line of program of accelerator 0 (license rd)
```

```
//=====================================================

paddr = 0;              fp03x8_paddr_write(&fp03x8_0_inst, paddr);
we = 0x1000;            fp03x8_we_write(&fp03x8_0_inst, we);
bram = 12;              fp03x8_bram_write(&fp03x8_0_inst, bram);

data2hw_wrapper((unsigned*)src_P1_P2_lic, 2*1);  //1
capture_wrapper((unsigned*)dest_P1_P2_lic, 2*1); //2
sds_wait(1);
sds_wait(2);

paddr = 0;              fp03x8_paddr_write(&fp03x8_0_inst, paddr);
we = 0x2000;            fp03x8_we_write(&fp03x8_0_inst, we);
bram = 13;              fp03x8_bram_write(&fp03x8_0_inst, bram);

data2hw_wrapper((unsigned*)src_P3_P4_lic, 2*1);          //1
capture_wrapper((unsigned*)dest_P3_P4_lic, 2*1); //2
sds_wait(1);
sds_wait(2);

// Run program of accelerator 0 (license rd)
//=====================================================
baddr = 0;              fp03x8_baddr_write(&fp03x8_0_inst, baddr);
paddr = 0;              fp03x8_paddr_write(&fp03x8_0_inst, paddr);
pstep = 0;              fp03x8_pstep_write(&fp03x8_0_inst, pstep);
go = 1;                 fp03x8_go_write(&fp03x8_0_inst, go);

go = 0;
while (fp03x8_pdone_read(&fp03x8_0_inst) == 0);
fp03x8_go_write(&fp03x8_0_inst, go);

// Read license info from Z addr 0
//=====================================================
paddr = 0;              fp03x8_paddr_write(&fp03x8_0_inst, paddr);
we = 0x0000;            fp03x8_we_write(&fp03x8_0_inst, we);
bram = 8;               fp03x8_bram_write(&fp03x8_0_inst, bram);

data2hw_wrapper((unsigned*)src_Z1_Z2_tmp, 2*1);          //1
capture_wrapper((unsigned*)dest_P1_P2_out, 2*1);         //2
sds_wait(1);
sds_wait(2);

// Write back the stored line of program of accelerator 0
//=====================================================

paddr = 0;              fp03x8_paddr_write(&fp03x8_0_inst, paddr);
we = 0x1000;            fp03x8_we_write(&fp03x8_0_inst, we);
bram = 12;              fp03x8_bram_write(&fp03x8_0_inst, bram);

data2hw_wrapper((unsigned*)dest_P1_P2_tmp, 2*1); //1
capture_wrapper((unsigned*)src_P1_P2_tmp, 2*1);          //2
sds_wait(1);
sds_wait(2);

paddr = 0;              fp03x8_paddr_write(&fp03x8_0_inst, paddr);
we = 0x2000;            fp03x8_we_write(&fp03x8_0_inst, we);
bram = 13;              fp03x8_bram_write(&fp03x8_0_inst, bram);

data2hw_wrapper((unsigned*)dest_P3_P4_tmp, 2*1); //1
capture_wrapper((unsigned*)src_P3_P4_tmp, 2*1);          //2
sds_wait(1);
sds_wait(2);

// Write back Z1 ..Z0 address 0
//=====================================================
paddr = 0;              fp03x8_paddr_write(&fp03x8_0_inst, paddr);
we = 0x0100;            fp03x8_we_write(&fp03x8_0_inst, we);
```

```
        bram = 8;               fp03x8_bram_write(&fp03x8_0_inst, bram);

        data2hw_wrapper((unsigned*)dest_Z1_Z2_tmp, 2*1); //1
        capture_wrapper((unsigned*)src_Z1_Z2_tmp, 2*1);          //2
        sds_wait(1);
        sds_wait(2);

        paddr = 0;              fp03x8_paddr_write(&fp03x8_0_inst, paddr);
        we = 0x0200;            fp03x8_we_write(&fp03x8_0_inst, we);
        bram = 9;               fp03x8_bram_write(&fp03x8_0_inst, bram);

        data2hw_wrapper((unsigned*)dest_Z3_Z4_tmp, 2*1); //1
        capture_wrapper((unsigned*)src_Z3_Z4_tmp, 2*1);          //2
        sds_wait(1);
        sds_wait(2);

        paddr = 0;              fp03x8_paddr_write(&fp03x8_0_inst, paddr);
        we = 0x0400;            fp03x8_we_write(&fp03x8_0_inst, we);
        bram = 10;              fp03x8_bram_write(&fp03x8_0_inst, bram);

        data2hw_wrapper((unsigned*)dest_Z5_Z6_tmp, 2*1); //1
        capture_wrapper((unsigned*)src_Z5_Z6_tmp, 2*1);          //2
        sds_wait(1);
        sds_wait(2);

        paddr = 0;              fp03x8_paddr_write(&fp03x8_0_inst, paddr);
        we = 0x0800;            fp03x8_we_write(&fp03x8_0_inst, we);
        bram = 11;              fp03x8_bram_write(&fp03x8_0_inst, bram);

        data2hw_wrapper((unsigned*)dest_Z7_Z8_tmp, 2*1); //1
        capture_wrapper((unsigned*)src_Z7_Z8_tmp, 2*1);          //2
        sds_wait(1);
        sds_wait(2);

        license_data =
         (unsigned int)(((unsigned int) dest_P1_P2_out[0].p[3]*256*256*256) +
                        ((unsigned int) dest_P1_P2_out[0].p[2] * 256 * 256) +
                        ((unsigned int) dest_P1_P2_out[0].p[1] * 256) +
                        ((unsigned int) dest_P1_P2_out[0].p[0]));

        sds_free(src_Z1_Z2_tmp);
        sds_free(src_Z3_Z4_tmp);
        sds_free(src_Z5_Z6_tmp);
        sds_free(src_Z7_Z8_tmp);
        sds_free(dest_Z1_Z2_tmp);
        sds_free(dest_Z3_Z4_tmp);
        sds_free(dest_Z5_Z6_tmp);
        sds_free(dest_Z7_Z8_tmp);
        sds_free(src_P1_P2_lic);
        sds_free(src_P3_P4_lic);
        sds_free(src_P1_P2_tmp);
        sds_free(src_P3_P4_tmp);
        sds_free(src_P1_P2_out);
        sds_free(src_P3_P4_out);
        sds_free(dest_P1_P2_lic);
        sds_free(dest_P3_P4_lic);
        sds_free(dest_P1_P2_tmp);
        sds_free(dest_P3_P4_tmp);
        sds_free(dest_P1_P2_out);
        sds_free(dest_P3_P4_out);
        return (license_data);
}
```