ECSEL2017-2-783162

FitOptiVis

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

Deliverable: D4.4 Monitoring, profiling, measuring and reconfiguration support for real time quality and resource management

Due date of deliverable: 31-05-2021

Actual submission date: 31-05-2021

Start date of Project: 01 June 2018 Duration: 42 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
Sajid Mohamed	TUE	s.mohamed@tue.nl
Giacomo Valente	UNIVAQ	giacomo.valente@univaq.it
Luis Medina Valdés	7SOLS	luis.medina@sevensols.com

Document history

Version	Date	Change
1.0	01.03.2021	Initial template and request for partner inputs
2.0	19.04.2021	Initial integration of partner inputs in Section 4 and 5
3.0	26.04.2021	Integration of partner inputs in Section 5
4.0	30.04.2021	Finalization for internal review
5.0	25.05.2021	Finalization taking into consideration the internal review

Document data

Keywords	runtime platforms, runtime reconfiguration, runtime monitoring
Editor Address data	Name: Dip Goswami & Giacomo Valente Partner: TUE & UNIVAQ Address: **d.goswami@tue.nl** **giacomo.valente@univaq.it** Phone: **+31 40 247 8242** & **+393490685728**

Distribution list

Date	Issue	E-mailer

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
gPTP	generalized Precision Time Protocol
BMCA	Best Master Clock Algorithm
MDC	Multi-Dataflow Composer

1. Executive Summary

This report represents deliverable D4.4 with updated results of D4.3 and final outcomes of Task 4.2 and Task 4.3 in 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. Final achievements of Tasks T4.2-T4.4 are reported in this document, including monitoring, profiling and measuring techniques, and reconfigurability support.

In this final reporting and iteration, we have updated the Iteration 2 outline reported in D4.3. This deliverable provides an overview of runtime reconfiguration and runtime monitoring mechanisms spanned over different levels of abstraction and serves to satisfy applications and use cases with diverse sets of requirements. In line with D4.3, the deliverable is split in two parts. The first part presents the overview of reconfiguration mechanisms in view of the FitOptiVis component abstraction framework and specific instances adapted by various partners. In the second part, the updated contributions of various monitoring mechanisms, developed by various partners, are reported.

The content of this deliverable contributes to MS7 (Specification update – which is made based on the performance obtained in the first iteration) and MS8 (Final demonstrators – providing technologies and methods to be integrated in the final demonstration).

2. Updates with respect to D4.3

This chapter provides a brief summary of specific content that has been updated and added to D4.4 with respect to D4.3. Naturally, the Introduction and Conclusion have been updated to reflect the new content.

2.1 Runtime reconfiguration

The following sections have been updated with respect to D4.3:

- 4.1 – Overview of runtime reconfiguration concepts in view of FitOptiVis Component framework
  - Updated few concepts with examples
- 4.2 – Dynamic reconfiguration in CompSOC
  - Updated with details of deployment framework of adding and removing applications at runtime
  - Provided the reconfiguration architecture capable of adding/removing applications in the runtime
  - Sketched the mathematical basics of the brokering and scalable Pareto-optimization
- 4.3 – Dynamic Reconfiguration using Multi-Dataflow Compositor
  - Updated with details of experimental results
  - Provided overview of static and dynamic parameter mapping from a dataflow graph to the corresponding hardware datapath
- 4.4 – Reconfiguration in Nvidia Jetson embedded devices
  - Acceleration through PoCL-remote improved and updated
  - Reconfiguration model and experiments added for UC3 and UC9
- 4.5 – Reconfiguration of Time Sensitive Network (TSN)
  - Updated with the details of the mechanism and experimental numbers on performance in UC3 and UC9
- 4.6 – RIE-based reconfiguration method
  - Included a new RIE reconfiguration mechanism
  - Added remote component implementation description
- 4.7 – Reconfiguration in Managed-Latency Edge-Cloud
  - Moved description of MECE adaptive loop from D4.2 to D4.4
- 4.8 Situation-aware reconfiguration in closed-loop control
  - This is an additional work (with respect to D4.3) on situation-aware system reconfiguration for image-based control

2.2 Runtime monitoring

The following sections have been updated with respect to D4.3:

- 5.1.1 - FIVIS data storage, visualization and analytics platform
  - Corresponding Section in D4.3
    - 4.2.1 – FIVIS data storage, visualization and analytics platform
  - Updates
    - The section has been updated to reflect the status of FIVIS
- 5.2.1 - Monitoring in 3D industrial inspection system
  - Corresponding Section in D4.3

        - 4.3.1 - Monitoring in 3D industrial inspection system
    - Updates
        - PoCL Telegraf JSON serializer plugin description added.
- 5.2.2 - Heterogeneous Distributed Computing Adaptation Monitoring
    - Corresponding Section in D4.3
        - 4.3.2 - Heterogeneous Distributed Computing Adaptation Monitoring
    - Updates
        - Monitoring Infrastructure: two new PoCL FIVIS tracing plugins described.
        - Data Storage, Analytics and Visualization: Described a new FIVIS PoCL visualization plugin.
- 5.2.6 - Monitoring of Distributed Execution in the Virtual Reality Use Case
    - Corresponding Section in D4.3
        - 4.3.6 - Monitoring of V-PCC in Virtual Reality
    - Updates
        - Description of the monitoring aspects and results of the augmented reality distributed rendering demo.
- 5.2.5 - Monitoring of 4x2 array of 8xSIMD Floating point Accelerators
    - Corresponding Section in D4.3
        - 4.3.5 - Monitoring of 8xSIMD Floating point Accelerators
    - Updates
        - This chapter describes how, UTIA SW support developed in Y3 of the project for the 4x2 array of 8xSIMD floating point accelerators for larger Zynq Ultrascale+ device ZU15-EG and developed run-time support for parallel execution if these accelerators implemented as Debian OS posix pthreads.
        - Chapter describes SW frame for control of accelerators computing in parallel with the running, HW-accelerated Full HD video processing pipeline.
        - Performance results for sequence of floating point matrix multiplications are compared with Xilinx HLS HW accelerator with same count of floating point ADD and MULT units.
        - Details related to the run-time programming of 8xSIMD HW accelerators are removed. (This was already described in D4.3).
- 5.2.3 - Monitoring systems for reconfiguration for Habit Tracking and Smart Grid
    - Corresponding Section in D4.3
        - 4.3.3 - Monitoring systems for reconfiguration for Habit Tracking and Smart Grid
    - Updates
        - New hardware requirement qualities and reconfiguration modes added
- 5.1.2 - QRML extension to express monitoring requirements
    - Corresponding Section in D4.3
        - 4.2.2 - DSL extension to express monitoring requirements
    - Updates
        - Several examples have been added
- 5.1.3 - JOINTER framework to build custom hardware monitoring systems
    - Corresponding Section in D4.3

        - 4.2.3 - AIPHS framework to build custom edge monitoring systems
    - Updates
        - Changed the structure of the whole framework to better meet state-of-art requirements about monitoring systems and to build monitor also for processors, memory, and interconnections.
- 5.2.11 - Monitoring of high-performance embedded applications for Water-Supply maintenance
    - Corresponding Section in D4.3
        - 4.3.11 - Monitor in Processor-Coprocessor systems
    - Updates
        - Added information about Water-Supply Use-case target.
- 5.2.4 - Monitoring capabilities for object recognition in space applications
    - Corresponding Section in D4.3
        - 4.3.4 - Monitoring capabilities for object recognition in space applications
    - Updates
        - The monitor description has been improved with the redefinition of the monitor mechanisms and the description of the tracing strategies. Additionally, the monitor requirement and physical platform lists have been extended and improved.
- 5.2.7 - Monitoring in Salmi-Care System
    - Corresponding Section in D4.3
        - 4.3.7 - Monitoring in Salmi-Care System
    - Updates
        - The section has been updated to reflect the current status of Salmi Care Platform.
- 5.2.10 - Pose and facial recognition in Habit Tracking with edge-cloud adaptivity
    - Corresponding Section in D4.3
        - 4.3.10 - Pose and facial recognition in Habit Tracking with edge-cloud adaptivity
    - Updates
        - HIB has detailed the integration of the previous developments in the edge devices with their new approach for cloud processing that incorporates new pose estimation engine (PoseNet instead of openpose) and new layer of action recognition with artificial intelligence based on LSTM networks
- 5.2.9 - Monitoring systems for localization in space applications
    - Corresponding Section in D4.3
        - 4.3.9 - Monitoring systems for localization in space applications
    - Updates
        - Added the monitors regarding the video services provided by TASE components. The previous contributions only had monitors at system level and now TASE added monitors to study the image quality.

3. **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 is an extension of D4.3 which reported an abstract view of reconfiguration and monitoring defined under the FitOptiVis reference framework followed by various instances developed by various partners. In D4.4, we mainly report further improvement and refinement of the technologies and methods reported D4.3 and use-case specific adaptation of them with results. Obviously, some new technologies are also developed over the last year of the reporting period which are also reported.

In Chapter 4, we report of the reconfiguration mechanisms developed by various partners within the FitOptiVis project. Section 4.1 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 4.2 presents a deployment framework a specific instance of the reconfiguration mechanism on the CompSOC platform (real-time mix-criticality platform) for adding/removing components at runtime. Section 4.3 presents a Multi-Dataflow Composer (MDC) tool-based reconfiguration mechanism targeting HW accelerators that allows for quality and budget adaptation in runtime and falls under second category of reconfiguration (i.e., changing the component configuration). Section 4.4 presents a reconfiguration mechanism for changing quality and budget (i.e., changing the component configuration) in runtime on Nvidia Jetson embedded devices and evaluated in Use case 3 (UC3) and Use case 9 (UC9). This falls under second category of reconfiguration. In Section 4.5, 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). This falls under second category of reconfiguration. The method is applied to UC3 and UC9 and results are presented. In Section 4.6, we present a RIE (of Embedded systems) based reconfiguration library which provides a general DSL framework for component implementation and reconfigurations. In Section 4.7, we present a reconfiguration mechanism for admission and deployment services between edge and cloud considering latency requirements. This falls under the first category of reconfiguration, i.e., adding and removing components. Finally, in Section 4.8, we present a situation-aware reconfiguration mechanism (under the second category) for embedded visual control for quality enhancement.

In Chapter 5, we report on the runtime monitoring, profiling and measuring mechanisms developed by various partners within the FitOptiVis project. A table highlighting the usage of runtime monitoring systems within the FitOptiVis UC is also reported. Section 5.1 highlights the importance of the runtime monitoring action with respect to the DSL

developed in WP2. In Section 5.2, three enabling solutions to perform monitoring in FitOptiVis are reported: enabling solution means that they are more identifiable as framework to support in the runtime monitoring actions, rather than specific instances of monitoring systems. Enabling solutions have been used by FitOptiVis partners to implement their desired monitoring systems. In Section 5.3, instances of monitoring systems, namely concrete solutions where monitoring systems have been adopted within FitOptiVis UC, are described.

Connections with WP2

In WP2, a reference architecture has been introduced that describes FitOptiVis systems in terms of their platform and application components, resource budgets, configurations and qualities. Additionally, a reference QRM framework is introduced in that WP that selects optimal configurations based on available resources and application requirements. Most of the aspects that are featured in this model and architecture are subject to dynamic variation and change. For various reasons their concrete values or states need to be observed and collected at run-time by a run-time monitoring infrastructure. It is important to know current availability of resources, status of a component, use of a budget, or application qualities, e.g., for active QRM and reconfiguration or to verify if requirements are sufficiently met. Monitoring further provides access to recent or long-term historic data for adaptive, calibration or verification purposes.

Systems and components may be modelled in the QRML domain specific language. Such models provide a machine-readable, mathematical model of a system, its components, configurations and the various qualities and budgets. This enables data collected from a run-time monitoring infrastructure to be related back, automatically, to the QRML specification. This enables, e.g., automated verification of the requirements specified in the model, but also visualization of current or historic monitored data in the context of the QML system model. FIVIS includes visualization of monitored data in use case specific dashboards, but also the visualization of the structure of the QRML specification through the QRMLVis visualization component so that monitored data can be related to their counterparts in the formal QRML model.

4. Runtime reconfiguration

4.1 FitOptiVis reconfiguration framework

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/remove components 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, Quality and Resource Management (QRM) components, or 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 the budget requirements of an application, a 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 a certain budget (e.g., an Nvidia GPU) 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 the output resolution of a video stream on surgeon's demand. This affects the 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).

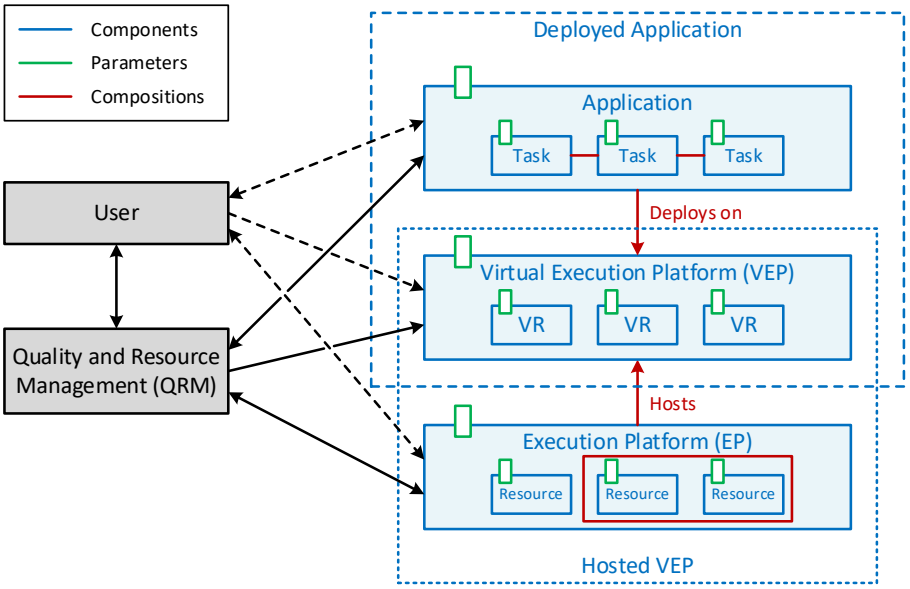


Figure 1: Overview of reconfiguration categories considered under FitOptiVis

- 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 the 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 the 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 hostings (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 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.

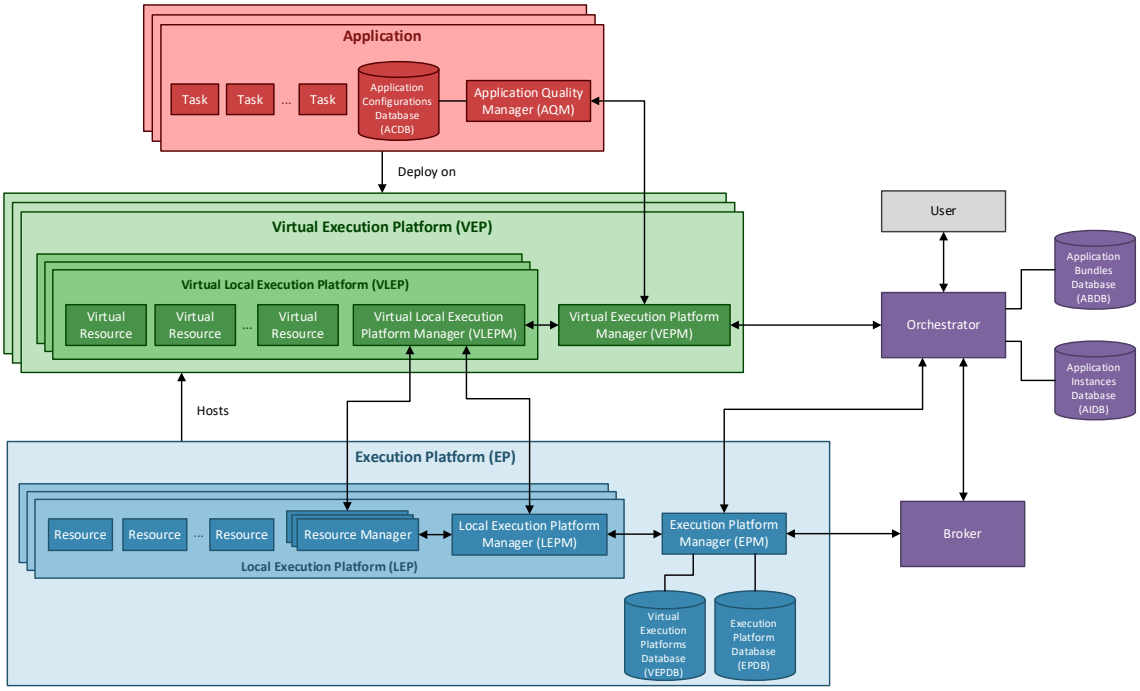


Figure 2: Block diagram of the proposed QRM architecture in CompSOC

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 adding/removing VEPs done by users.
- Adding/removing 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.
- Reconfiguring the 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 seven reconfiguration mechanisms developed targeting the above three categories of actions in Section 4.2-4.8. It covers a wide range of platforms and applications subject to reconfiguration mechanisms -- Real-time, mixed criticality embedded systems (Section 4.2 on CompSOC), HW acceleration (Section 4.3 on MDC), industrial embedded platforms and visual control (Section 4.4 and 4.8 on NVIDIA), time-

sensitive networked systems (Section 4.5), edge-cloud systems (Section 4.7) and software systems (Section 4.6).

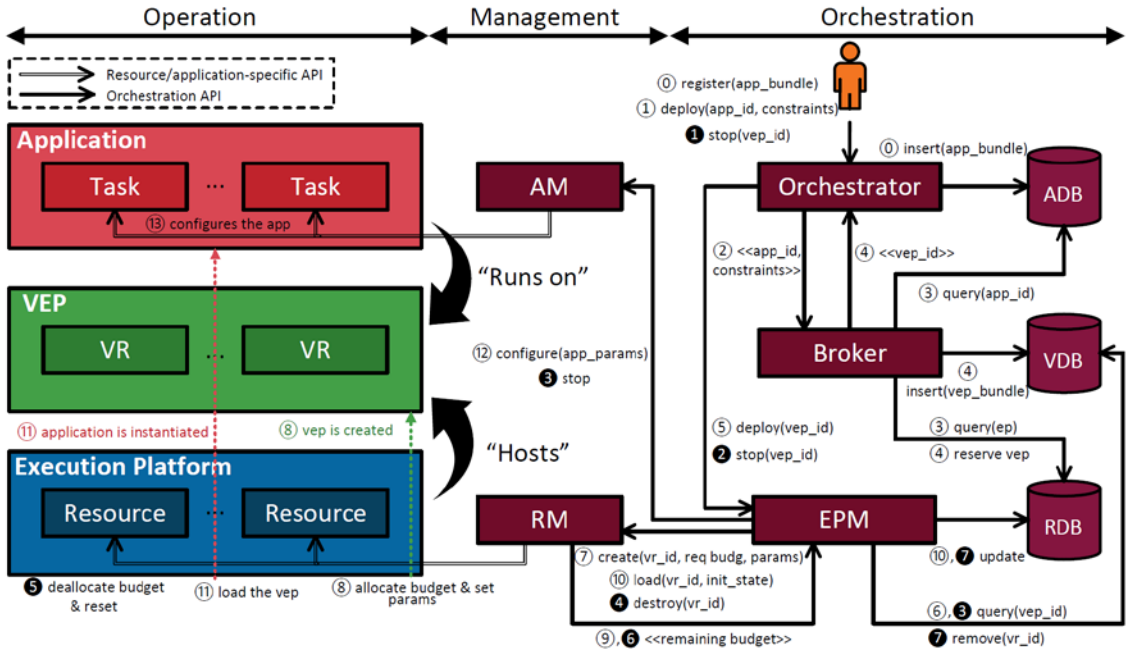


Figure 3: The architecture of the proposed framework and the deployment flow

4.2 Dynamic Reconfiguration in CompSOC

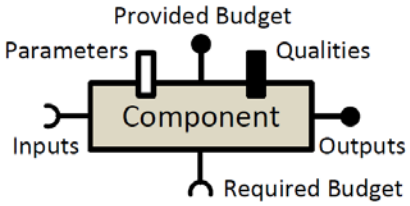
We employ the QRM architecture depicted in Figure 2 to perform dynamic reconfiguration in CompSOC platform. The architecture is designed in such a way that performing any type of reconfiguration action is possible. In the following, we first introduce the framework architecture. Then we discuss its two key operations, namely application deployment and Pareto optimization.

4.2.1 Architecture

The framework contains three layers (from left to right), namely i) operation, ii) management, and iii) orchestration, which are introduced in the following.

Operation: This layer contains software and hardware components of three types, namely i) *Applications*, which are made up of *Tasks*, ii) *Resources* that are hardware/software components on top of which applications execute, and iii) *Virtual Resources (VRs)* that are spatial and/or temporal resource partitions to share resources. VRs are often implemented by schedulers, microkernels, virtualization, or (RT)OS on top of a hardware resource. Each application runs on a dedicated set of VRs called a *Virtual Execution Platform (VEP)*. A VEP is a hierarchical component containing VR components. In this framework, we use the FitOptiVis component interface model proposed in WP2, shown in Figure 4, due to its flexibility in modeling applications and resources as well as its suitability for quality and resource management [HENDRIKS20]. In this modeling framework, component configurations are modeled with a set of points

defined on $Q_{inputs} \times Q_{outputs} \times Q_{reqBudget} \times Q_{prvBudget} \times Q_{qualities} \times Q_{params}$ space where each dimension is a *partially-ordered set (poset)* and corresponds to one of the component interfaces. A partial order represents how quantities of a component interface are better than, worse than, or incomparable to other quantities of the same interface. For example, higher application quality levels, smaller required budgets, and larger provided budgets are considered better.




```
"components":
[
  {
    "id" : "Task1",
    "configurations":
    [{
      "inputs":[{"raw_frames" : "30Hz"}, ...],
      "outputs":["processed_frames" : "30Hz"}, ...],
      "parameters":[{"resolution" : "720p"}, ...],
      "qualities":[{"framerate" : 30}, ...],
      "required_budget":
       {
        "TILE": { "RISCV":
                  [{
                    "unit" : "cycles",
                    "type" : "average_rate",
                    "value" : 100K
                   },... // other services from RISCV
                  ], ... // other resources from TILE
                  }, ... // other resources besides TILE
       },
      "initial_state":[{"IDMEM" : ".../task1.hex"}, ...]
     }, ... // other application configurations
    ]}, ... // other components
],
"compositions":
[
  "App1 = Task1 => Task2",
   ...
]
```

Figure 4: Component model and bundle

Management: The management layer prepares the operation layer for execution and consists of *Application Managers (AMs)* and *Resource Managers (RMs)*. AMs are responsible for configuring, booting, starting, and stopping applications. This can be as simple as setting the image resolution or as complex as booting a virtualized OS. RMs, on the other hand, monitor resources and perform lifecycle management of VRs including creating/destroying (by allocating/deallocating budgets), configuring (by setting parameters), initializing/resetting VRs (by programming resources).

Orchestration: The objective of the operation and management layers is to realize application deployments, which includes creating VEPs by partitioning resources, configuring the VEPs by setting resource parameters (e.g., processor frequency), initializing the VEPs (e.g., loading applications), configuring applications (e.g., setting parameters of an image filter), and executing them. Taking these steps requires knowing the size of resource partitions, application-to-resource bindings, resource and application parameters, and initial data of VEPs. All these are determined by the orchestration layer besides its other objective that is automating the deployment process. The orchestration part is comprised of several functional blocks briefly discussed in the following.

- The *Orchestrator*, the entry-point of the framework, accepts deployment requests and coordinates the operation of other functional blocks to plan and realize deployments.
- The *Broker* determines the optimal application deployments including the optimal component configurations (i.e., application and resource configurations) and compositions (e.g., VR-to-resource bindings).
- The *Execution Platform Manager (EPM)* coordinates creation, configuration, and destruction of VEPs by RMs.
- Based on the component types, three *databases* exist that store *component bundles*. A component bundle contains the model of the component as well as initialization data required to load component instances with (e.g., application instructions and data, VM image, hardware parameters). As shown in Figure 4, component models are stored in the JSON format due to its readability and simplicity. Each database is composed of two sets, *components* and *compositions*. The former contains bundles of atomic components (e.g., application tasks) and the latter demonstrates how composite components (e.g., applications) are made up of other components (e.g., *A1 = T1 => T2*, which means *A1* is the horizontal composition of *T1* and *T2*). A bundle of an atomic component contains a unique identifier and a set of *configurations* describing component properties.

Distributing the orchestration tasks among multiple functional blocks lets us pipeline deployments, which improves the responsiveness of the system. Additionally, separation of resource management and coordination of RMs enables us to dynamically add resources and their managers to the system by just hooking RMs to the EPM at run-time (using Data Distribution Service). This is highly beneficial for fog/edge settings where extra nodes can be dynamically added to the system.

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.

4.2.2 Application Deployment

We assume the deployment command is issued by the end user. In other words, the user decides to execute/stop an application. As shown in Figure 3, the flow is composed of ⑬ steps for deploying and ❼ steps for stopping an application. The separation of orchestration and management layers enables us to execute them on different platforms. For example, we can run the orchestration layer, which is more computationally intensive, on more powerful resources to speed up the deployment process. Additionally, in distributed systems, this design lets us have an RM for each subsystem, enabling them to operate in parallel and speeding up the deployment of distributed applications. The orchestration tasks are also distributed among three functional blocks, pipelining the deployment process. For instance, the deployment (done by EPM) and brokering of two consecutive deployment requests can be done in parallel. The deployment steps are explained in the following.

Deploying Applications:

⓪ An application deployment starts with registering the application bundle into the *Application Database (ADB)*. This lets users dynamically introduce new applications to the system. Each application bundle can be instantiated multiple times independently.

① Once the bundle is stored in the database, a deployment request containing the application identifier and deployment constraints including set-points for application parameters (e.g., image resolution), minimum quality levels (e.g., minimum frame rate), and maximum deployment costs (e.g., overall power) is sent to the Orchestrator.

②,③ The deployment request is forwarded to the Broker for deployment planning using the Pareto Calculator tool (http://www.es.ele.tue.nl/pareto/calc/) [GEILEN07]. The Broker builds the exploration space by fetching application and resource bundles from the ADB and Resource Database (RDB). This ensures that deployments use the latest system state.

④ The optimization result is a VEP configuration (stored as a VEP bundle in the Virtual Resource Database, VDB) containing the optimal application configuration, resources the application is bound to, their optimal configurations, and the budget that must be allocated to the VEP. Updating the VDB is followed by budget reservations done by updating the remaining budget of resources in the RDB. Since remaining steps may take time, the status of resources are updated first to make sure that the subsequent deployment plannings are done based on the latest state of the system so that they can be done in parallel (pipelined). Additionally, the Broker sends the identifier of the reserved VEP (*null* in case of no feasible solution) to the Orchestrator. This lets the Orchestrator know the feasibility of deployment and it can inform the user.

⑤ In case of a feasible solution, a deployment request containing the VEP identifier is sent to the EPM.

⑥ The EPM fetches the VEP bundle from the VDB.

⑦ It sends VR creation requests to RMs. Requests are sent per VR and contain an identifier (used for future reference), the budget that the VR is supposed to provide, and possibly existing VR parameters to set (e.g., vCPU frequency). Note that the VR-to-resource bindings are included in the budget.

⑧ Upon receiving the creation requests, RMs create VRs using resource-specific southbound APIs (concurrently).

⑨ Once VRs are created, RMs respond with the remaining budget of resources expressed in possibly more detailed abstractions to retrieve the possibly lost budget details.

⑩ The EPM sends loading requests for VRs that require to be initialized for their operation. Since loading VRs (⑪) is often slower than creating them (⑧), this step is done in parallel with updating the state of the system (i.e., remaining budget of resources).

⑪ Once the VRs are initialized, the application is instantiated and ready to start.

⑫,⑬ If the application has parameters to set, the EPM sends them to the AM and it configures the application using application-specific interfaces.

Stopping Applications:

Stopping entails first asking the AM to stop the application. Then all steps are reversed, omitting brokering.

4.2.3 Brokering: Pareto Optimization

As explained above (Steps③,④), the goal of brokering is to determine a VEP configuration such that deployment constraints are satisfied and a cost function is optimized. To do so, the component configurations in the bundles are retrieved. The Pareto optimization is the following poset-algebraic expression:

$$vep^* = Min(D_C \cap D_Q \cap (\cup_i (vep_i \Uparrow (app \cap D_P))))$$

where all the operations are performed on the component interfaces (i.e., required and provided budgets). First, parameter constraints (D_P) are applied to the application configuration points (app). Next, a set of *VEP candidates* are built where each candidate (vep_i) is a component containing all the resources that the application requires and corresponding to an application-to-platform binding. All the candidates are vertically composed to the application (to perform budget matching) to build a configuration space containing feasible bindings, deployment costs, and application quality levels. Next, the possibly existing quality (D_Q) and cost constraints (D_C) are applied (e.g., desired quality), resulting in a set of Pareto points. Finally, the Pareto frontier is minimized to one configuration (vep^*) by considering a certain policy (e.g., workload balancing) or by arbitrarily picking a point.

Note that, as shown in Figure 5, the vertical composition operator (⇑) requires addition and subtraction operations to be defined on posets (i.e., component interfaces such as provided and required budgets). Hence, to find the optimal deployment, we need to know how to i) compare (≼), ii) add (+), and iii) subtract (−) two component interfaces (e.g., required/provided budget), which makes the framework generic and suitable for heterogeneous components. To achieve this, all the component interfaces are modelled with simple *(name, value)* pairs (e.g., *(resolution, 720p)*). A partial order is defined on the values of each quantity type as well as any Cartesian product of component interfaces. Budgets have an additional hierarchy (encoded as nested JSON objects). The hierarchical structure lets us model how the resources are coupled together without adding adhoc optimization constraints. For example, if we ask for a processing tile comprised of a processor and a memory, we want them to be mapped on the same tile; however, asking for a processor and a memory without the notion of tile does not enforce this constraint.

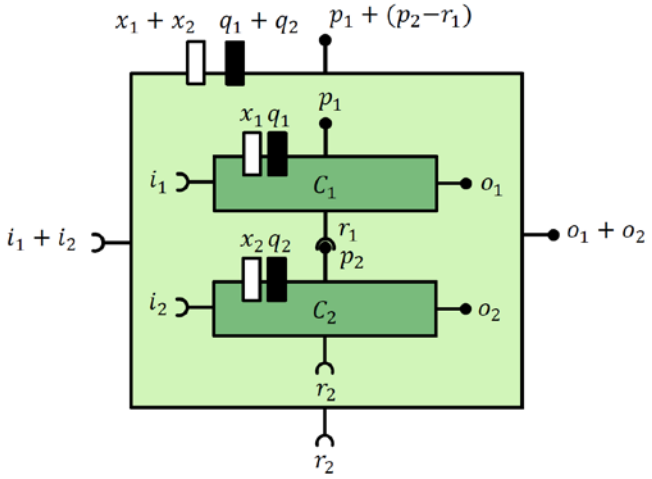


Figure 5: The vertical composition of two components

An atomic budget is modeled with a *(unit, type, value)* tuple specifying the unit of the budget (e.g., *cycles* for processing power, *bytes* for memory capacity), the model of the budget (e.g., *average_rate*, *TDM_Slot*), and quantity of the budget (e.g., 20% for the CPU share). Similar to *names*, *units* of operands must be identical. Various resource models and abstractions are proposed in the literature (e.g., Latency-Rate, Service Curves). Budget abstraction is a trade-off between analysis time and accuracy (e.g. overhead). More specific budgets may be less likely to be granted. Our framework allows RMs to use specific *provided budgets* (for high efficiency) and at the same time allows applications to *require abstract budgets* (for mapping flexibility) by automatically converting between abstraction levels of budgets of the same type.

For example, we still can do the budget matching if an application is profiled with average resource requirements (e.g., average processing rate) while resources are abstracted with Time-Division Multiplexing (TDM) tables. However, this comes at the cost of losing information when we add/subtract two budgets of different abstractions. For instance, if the required and provided budgets are expressed with the number of memory blocks and the address of blocks respectively, we only know the number of remaining blocks after subtracting the two budgets, unless we know how the memory manager allocates blocks to applications. However, including the allocation policies and implementation details slows down the optimization process. On top of that, the allocation algorithms may be proprietary and not be available. Therefore, we only allow budgets with less or the same details to be subtracted from (or compared with) other budgets and the abstraction of the result is similar to the less-detailed budget. In the allocation phase, RMs *refine* the abstracted budgets, and to solve the information loss problem, they report the remaining budget after allocating/deallocating budgets to/from VRs, and the RDB is updated with the retrieved abstractions (i.e., Steps ⑩ and ❼ in Figure 2).

4.3 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 at 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 configurations), 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 6) – the accelerator offers different functionalities (e.g. different image processing algorithms).

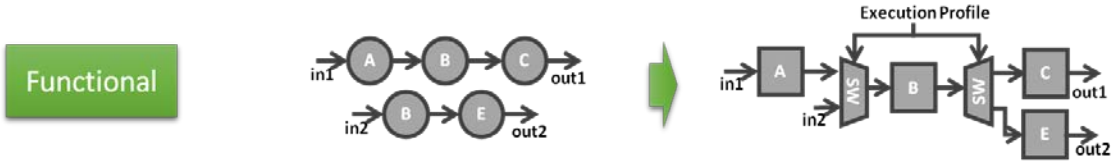


Figure 6: Functional-oriented reconfiguration

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 **Errore. L'origine riferimento non è stata trovata.**, 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).

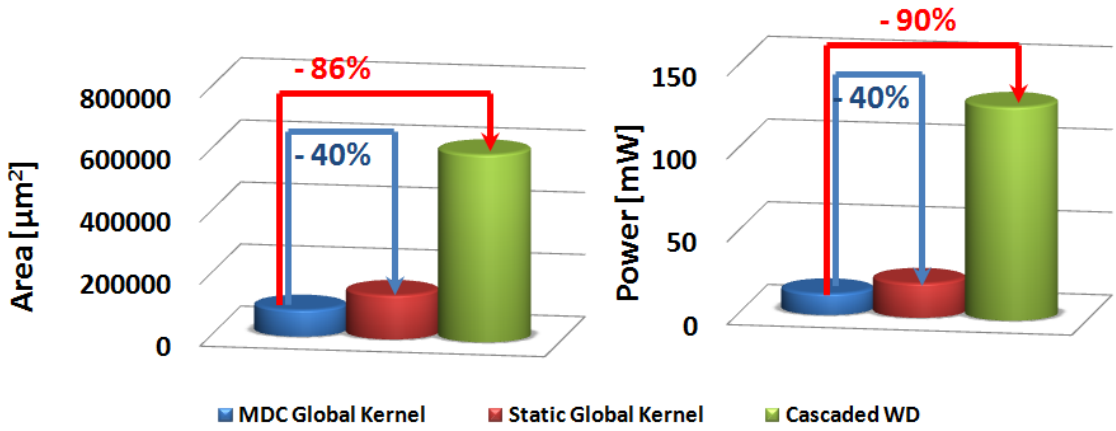


Figure 7: Area and power consumption histograms of the MDC generated denoiser (MDC Global Kernel) with respect to the corresponding non-reconfigurable denoiser (Static Global Kernel) and with respect to a unrolled atomic denoiser implementation (Cascaded WD).

- Working point-oriented (see Figure 8) – 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).

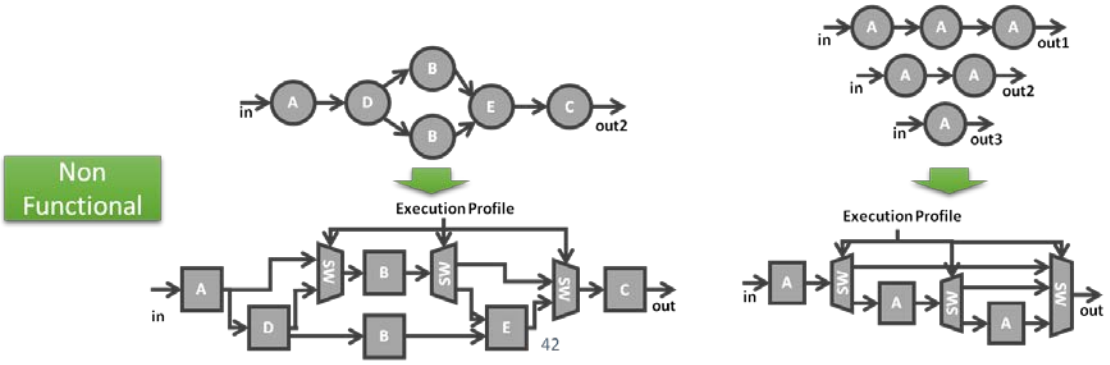


Figure 8: Working point-oriented reconfiguration

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 9, 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.

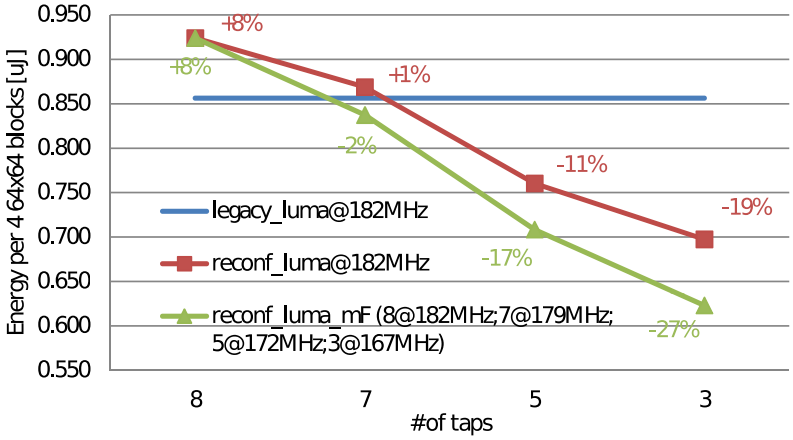


Figure 9: Reconfigurable HEVC interpolation filter: the supported working points provide different energy versus quality (Inv. proportional to # of taps) trade-off.

Dynamic Parameters

The reconfiguration capabilities delivered by MDC accelerators have been enhanced by providing support for dynamic parameters. In the past, having parameters on the dataflow models taken as input by MDC was allowed, but these parameters mapped only static parameters on the final hardware specification. With static parameters we intend parameters that are fixed at design time and that cannot be changed at runtime.

Basically, they reflect Verilog parameters or VHDL generics, and they can also have an impact on the resulting system architecture, meaning that adopted resources can change if static parameters change.

With dynamic parameters, we instead intend parameters which can be changed at runtime, then can create reconfigurations/modifications in the computation in a dynamic way. Those dynamic parameters are then reflected in the hardware specification as datapath inputs directly connected to the accelerator configuration registers. Such inputs, unlike the ones related to dataflow connections, are placed only on actors presenting them on the model, and do not follow a dataflow FIFO based protocol, but are driven directly by the values written in dedicated configuration registers. **Errore. L'origine riferimento non è stata trovata.** depicts how static and dynamic parameters are mapped into hardware.

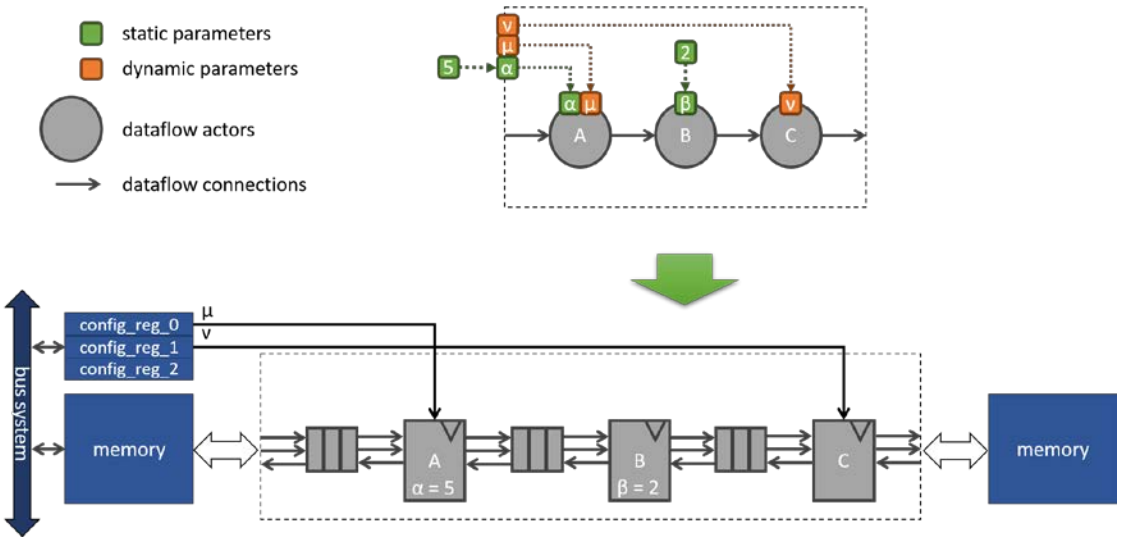


Figure 10: Overview of static and dynamic parameter mapping from a dataflow graph to the corresponding hardware datapath.

To better understand how dynamic parameters are specified in the practice, it is necessary to give some details about the adopted dataflow formalism, the RVC-CAL, where the network is specified through XDF, an XML dialect, while actors are specified through CAL. On the XDF network it is possible to define variables and parameters, while on the CAL files only parameters are available. To support both static and dynamic parameters, according to the previous definitions, network level variables are used to specify exclusively static parameters, while network level parameters specify exclusively dynamic parameters. On the actor side, dataflow parameters are used to define both hardware static and dynamic parameters. The differentiation is made according to name matching with respect to variables and parameters defined at the network level. If no matching is found, the parameter is interpreted as static and the default value is assigned on the generated Verilog top module.

Dynamic parameter support has been added to the MDC 0.0.2 release on 14/01/2021, available on https://github.com/mdc-suite/mdc.

4.4 Reconfiguration in Nvidia Jetson embedded devices

UC3 (Habit tracking) and UC9 (Smart grid) use the NVidia Jetson embedded devices Jetson Xavier, Jetson TX2, and Jetson Nano. 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. Critical action recognition is crucial. Therefore, we take advantage of the knowledge extracted by the Optical Flow stream to confirm whether potentially life-threatening situations occurred. Combining the RGB and Optical Flow streams leads to a more complex and accurate solution for action recognition (Two Stream I3D). Thus, the reconfiguration follows the dataflow represented in F.

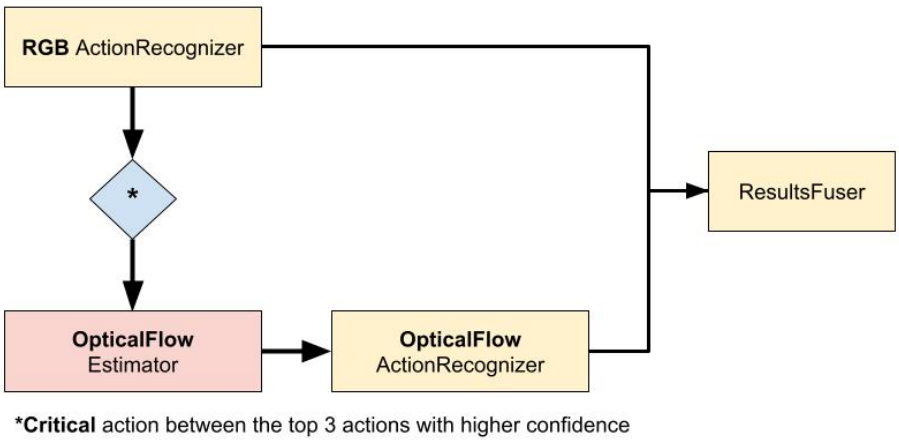


Figure 11: Diagram of tasks involved in action recognition (Two Streams)

For instance, if we detect that the active model cannot properly distinguish whether a critical action such as *falling down* or *lying on the floor* occurred, 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 TAU, 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 CUDAcores.

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. In Figure 12, it shows that processing time improvement is about 12x estimating Optical Flow through pocl-remote compared to using the Jetson TX2 edge platform.

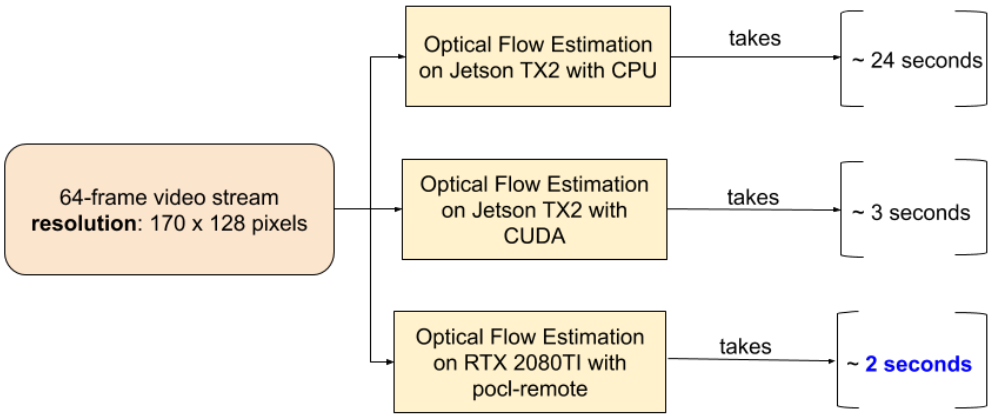


Figure 12: Compare time required to estimate Optical Flow TV-L1

After the initial tests with **pocl-remote**, the integration on the Jetson devices was optimized through OpenCV to accelerate the Optical Flow estimation even more. For example, for the NVidia Jetson Nano device, which is the most efficient and resource limited Jetson device, it obtains a **70%** improvement with respect to using the device GPU when estimating Optical Flow, taking less than **0.8 seconds** (See Figure 13). For testing purposes, we use a video configuration of 64 frames with 112x112 resolution.

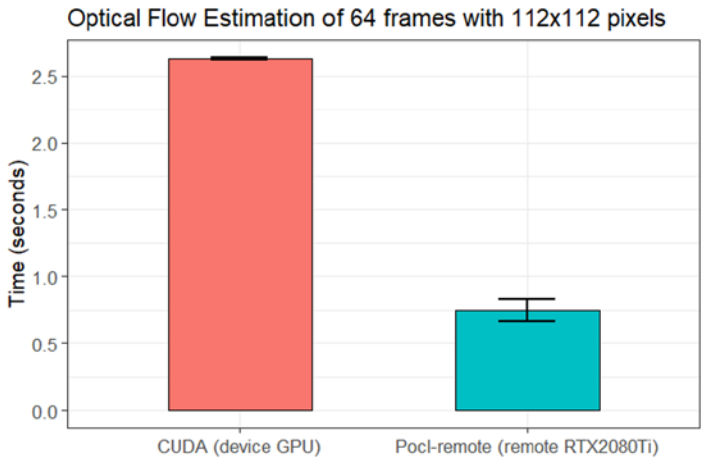


Figure 13: Optical Flow TV-L1 estimation comparison: only CUDA device vs pocl-remote implementation

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.

The improvements obtained with **pocl-remote**, the integration with the **FIVIS** platform and the development of new Deep Learning alternatives that offer a different accuracy vs power consumption trade-off fostered the introduction of this new run-time reconfiguration for the reconfigurable action recognition system, that runs on Jetson embedded devices (see Figure 14). In the proposed system architecture, **FIVIS** is our Quality and Resource Management (QRM) tool that monitors the system qualities and triggers reconfigurations based on it. With the integration of these tools, we expect to close the loop for run-time reconfigurations.

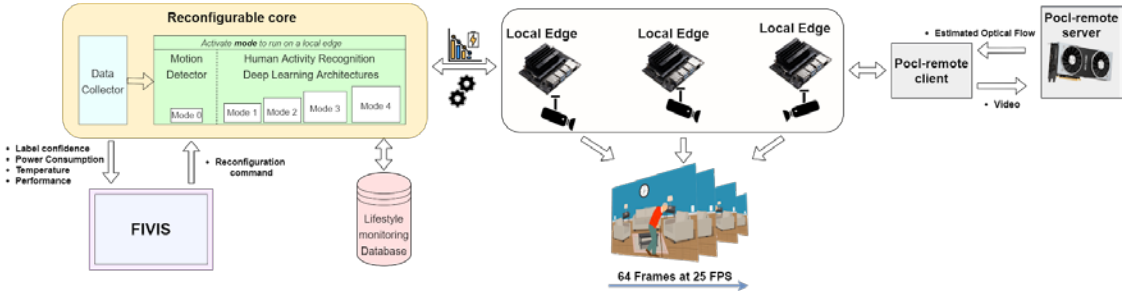


Figure 14: Reconfigurable CPS architecture for action recognition using FIVIS and pocl-remote

As described above, this run-time reconfiguration follows two objectives: 1) Reduce power consumption; 2) Confirm **critical** actions, reducing false alarms.

Firstly, we categorize the activities between temporally long (eg. *cleaning the floor*, *eating* or *sleeping*) and brief actions (*standing up or sitting down*). We switch to more efficient DL models when the system detects continuous long actions to reduce the power consumption and extend working time at the expense of losing some accuracy.

Secondly, we designed a complex and more accurate DL model (*Two Stream I3D*) in order to confirm critical actions when they occur. This DL model analyzes the video using a RGB and a Optical Flow stream. In particular, the use of **pocl-remote** enables the

offloading of the Optical Flow estimation to a remote GPU, reducing resource usage on the node allowing the integration of the Two Stream architecture on the Jetson devices. Figure 14 shows how the use of pocl-remote accelerates the inference time by **47%** compared to computing the whole Two Stream network on the Jetson Nano GPU with CUDA (also, reduces power consumption at the edge).

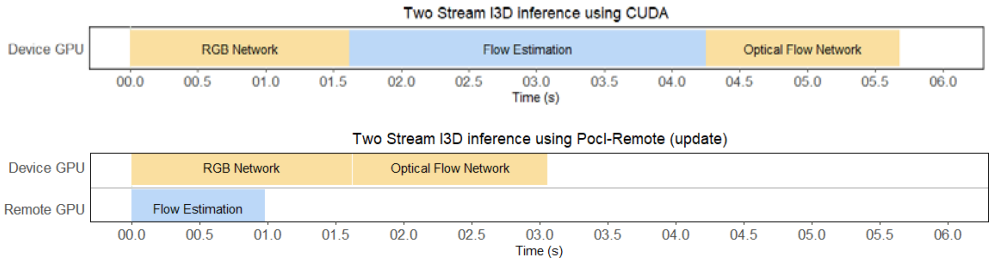


Figure 15: Two Stream inference times when using pocl-remote compared to only using the device GPU

Based on this, we designed the reconfiguration policy shown in the state transition diagram in Figure 15. The action recognition architectures involved in the presented system, in order from least to most accurate (and from most to least energy efficient), are as follows: 1) *RGB 16 - 112* (Mode 1); 2) *RGB 32 - 112* (Mode 2); 3) *RGB 64 - 112* (Mode 3); 4) *Two Stream 64 - 112* (Mode 4).

- The most power efficient alternative (*RGB 16 - 112*) is used when continuous (long) activities are recognized with high confidence. This enables reducing the system power consumption at the expense of losing some accuracy.
- Mode 2 and 3 are active when identifying brief actions in regular operation. Depending on the remaining battery, a different Mode is selected. Note that Mode 3 (*RGB 64 - 112*) reaches higher accuracy at a greater energy cost. Switching to Mode 2 (*RGB 32 - 112*) when low battery levels are detected contributes to extend working time.
- Finally, *Two Stream 64 - 112* (Mode 4) is activated when FIVIS notifies that a critical action was identified with low levels of confidence. Therefore, the video again is analyzed with the most accurate alternative to confirm whether a life-threatening situation really occurred.

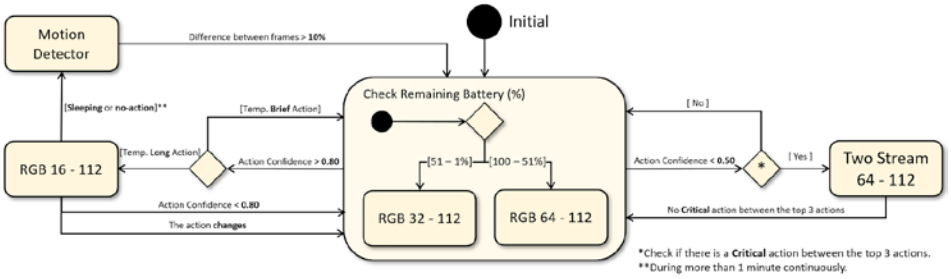


Figure 16: State diagram of run-time reconfiguration policy

Figure 16 shows an example of a reconfiguration triggered to confirm whether a *fall* occurred. Firstly, if only the RGB video is analyzed the system outputs that someone felt down with 0.27 confidence. Then the *Two Stream* network processes the RGB video, and the estimated optical flow (via pcl-remote) inferring that someone *fell down* with 0.99 confidence. With this, false alarms are reduced and critical actions are correctly classified.

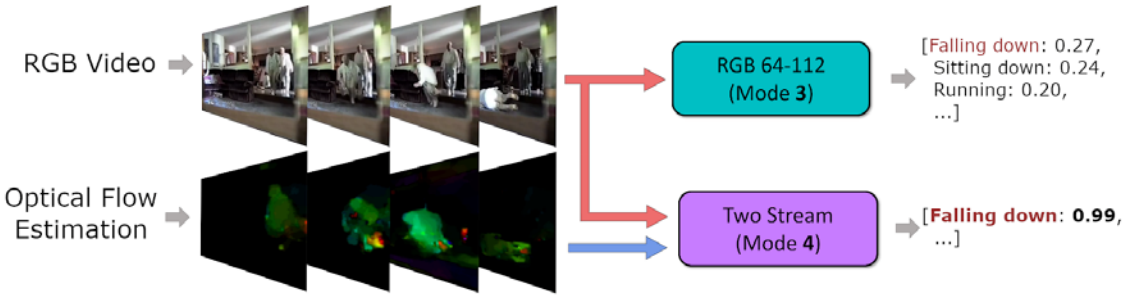


Figure 17: FIVIS reconfiguration to confirm critical actions

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 18). 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 18). Also, the execution of this other sub-component can trigger the generation of an alarm through the AlarmGenerator sub-component (red area in Figure 18). More details are given in Deliverable 5.2.

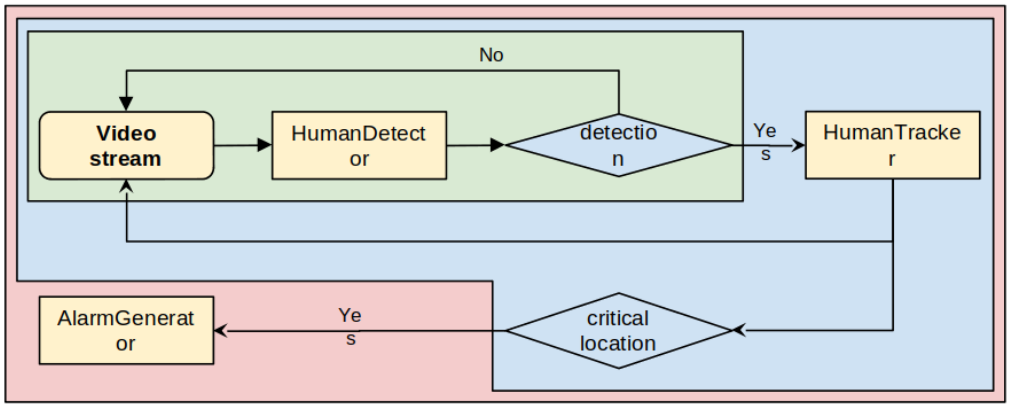


Figure 18: Smart-Grid surveillance system component composition

Our CPS has distributed processing between local nodes at the edge (Jetson platforms) and the central cloud server. At the edge, the local nodes are responsible for processing the video and streaming it to the cloud server. Figure 19 shows how the surveillance tasks and the processing are distributed between the nodes at the edge and the cloud server.

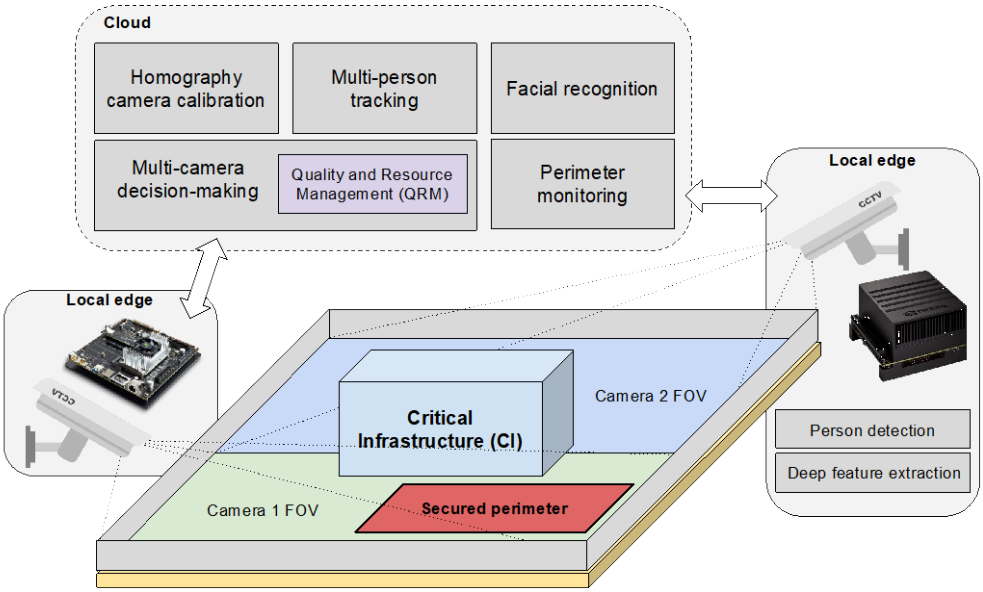


Figure 19: Re-configurable CPS architecture for Smart Grid CI protection

In a CPS where all subsystems are networked, bandwidth is a critical resource for an operation that must be successful and reliable. However, video processing demands high data bandwidth, and it could cause delays in the transmission of alarms, or even video-surveillance results. In order to overcome this issue, the Smart Grid protection CPS is reconfigurable, including a QRM (Quality and resource management) component. In other words, our system changes its operation mode to adapt the amount of required resources according to the task or operation context. Table 1 lists the different operating modes, varying the video acquisition quality on each edge local node depending on the substation context. The QRM process is performed both at the cloud server and the local edge node levels:

- **QRM core**: The decisions about the functioning mode of each of the local nodes are taken at the cloud server (Cloud in Fig. 18). This process takes into consideration aspects such as location and trajectory of the tracked person within the substation, distances to monitored perimeters and protection zones, distance from each camera and likelihood of finding the track within its FOV. Each time a decision is made about how each local node will operate; a command is sent specifying the required reconfiguration mode.
- **Video scaler**: It is responsible for the adjustment of the quality of the captured video (spatial resolution and frame rate) from the surveillance camera on the local node according to the reconfiguration instruction received from the QRM core.

As mentioned above, the different commands triggered by the QRM core of the cloud server result in re-configurations on the local nodes. These different re-configurations or operation mode changes of the local nodes video scaling component are shown in Table 1.

Table 1: Edge re-configuration modes

Edge operating modes	Resolution (fps)	Livestream video bandwidth	Event(s) that triggers re-configuration
Mode 2	.1.1 1280x960 (30)	.1.2 8,40 MB/s	.1.3 Confirmed intrusion
			.1.4 Broken perimeter
Mode 1	.1.5 640x480 (15)	.1.6 1,90 MB/s	.1.7 Posible intrusion
			.1.8 Detection
Mode 0	.1.9 320x240 (5	.1.10 0,41 MB/s	.1.11 Nothing relevant occurs

Figure 20 shows a reconfiguration example for the Smart Grid protection CPS: if nothing relevant happens (t1 in cam1 and t3 in cam2), the node operates in mode 0; if a person is detected (detection or pre-diction of the trajectory within the FOV of at least one camera), mode 1 is activated (t2 in cam1- blue area on the map - and t1 and t2 in cam2- red area on the map); finally, mode 2 is activated with the intrusion in the secured perimeter (t3 in cam1- green area on the map). The aspects considered to carry out the reconfiguration of the nodes are the following: When a person is detected within the node FOV (t1 person within the FOV of the camera of local node 1 (cam1); blue area in Fig. 19-bottom), it changes its operation mode from 0 to 1. Next, when the trajectory of a tracked target points to the area of the predicted region with 95% confidence (Ellipses show the potential region in which with 95% confidence the track will be found in the next 200ms) being within the FOV of another camera (t2 with predicted trajectory within cam2 FOV; red area), the operation mode goes up to 2. Similarly, when the trajectory of a tracked target entries the perimeter of a secured area, the operation mode changes to 2 (t3; green area).

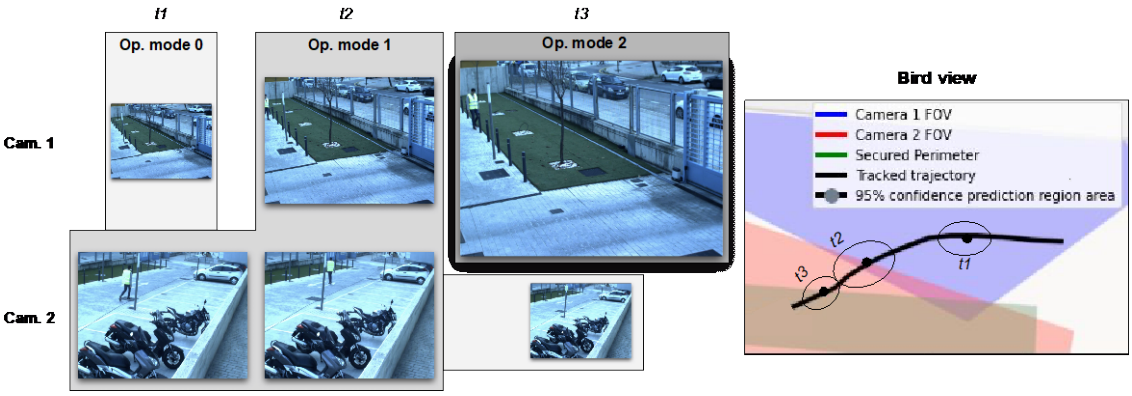


Figure 20: Reconfigurations example

With respect to the overall system, reconfiguration is triggered to optimize the use of data bandwidth of the shared communication network in our multi-camera system. The use of data bandwidth directly depends on image resolution that is context-aware as shown in Table 1. Figure 21 shows a reconfiguration example for CPS cloud-edge

bandwidth use. Three local edges are dynamically reconfigured according to the events (arrow) of the monitored environment. The gray area represents the bandwidth reduction (≈75%) with respect to no reconfiguration scenario.

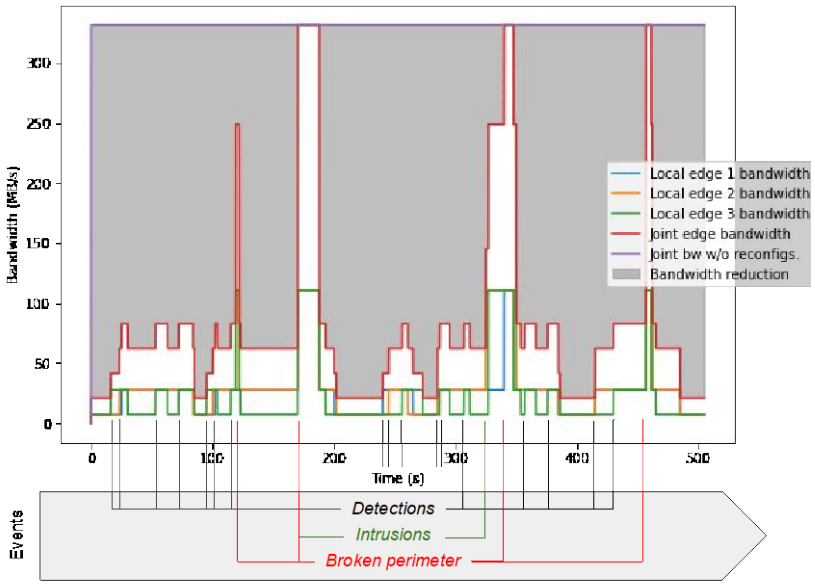


Figure 21: Bandwidth use vs reconfigurations

4.5 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 5.2.8. 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 the so-called Best Master Clock Algorithm (BMCA).

The Best Master Clock Algorithm

The Best Master Clock Algorithm (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 below:

Table 2

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.

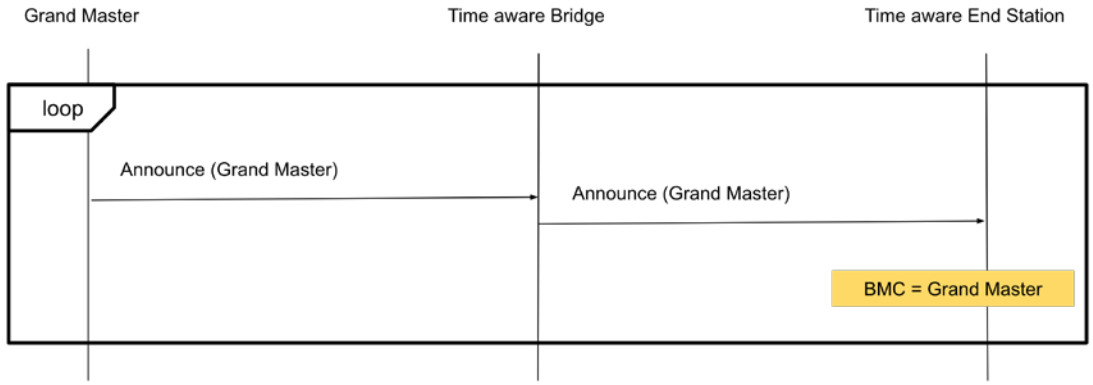


Figure 22: Announce message exchange and Best Master Clock election

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 following roles:

Table 3

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.

Best Master Clock Algorithm application on UC3 and UC9

The runtime configuration and the failover provided by the BMCA has been validated through different scenarios, considering the three-node ring deployment used on UC3 and UC9.

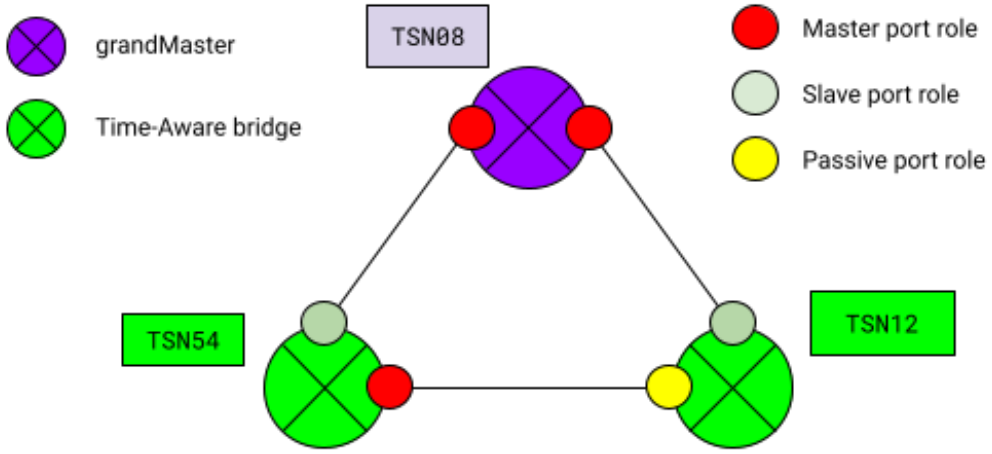


Figure 23: TSN ring deployment on UC3 and UC9.

The gPTP configuration is summarized on the table below. The three TSN bridges have the same configuration and offer the same time reference (the internal oscillator). Consequently, the grandMaster eligibility is given by the lowest clockIdentity value. In normal function, this corresponds to TSN#08 node. If unavailable, the TSN#14 will take the grandMaster role and source timing to TSN#54.

Table 4

Parameter	TSN#08	TSN#54	TSN#14
priority1	250	250	250
clockClass	13	13	13
clockAccuracy	34	34	34
offsetScaledLV	14208	14208	14208
priority2	248	248	248
clockIdentity	aa:bb:cc:ff:fe:dd:ee:08	aa:bb:cc:dd:ff:fe:dd:54	aa:bb:cc:dd:ff:fe:dd:14

To validate the BMCA, the port roles and the node configurations will be checked. Besides, the failover time will quantify the reliability against network failures. This measures have been considered along the time:

To check the configuration and failover times, three different scenarios have been considered:

a) Normal function. The default network configuration and the initial convergence measures will be presented.

b) Link TSN#08 – TSN#12 failure. In this scenario, the default synchronization path to the grandMaster is unavailable for TSN#12. The TSN#12 will readapt the port roles to receive synchronization from TSN#08 through TSN#54.

c) Link TSN#08 - TSN#54 failure. This scenario is similar than the previous one. However, the passive port on TSN#54 will automatically back the failed slave link, so the recovery is shorter.

d) TSN#08 failure. This is the worst case scenario. In this case, the TSN#12 should become grandMaster and TSN#54 will readapt to receive synchronization from this node.

4.6 RIE-based reconfiguration method

Reconfiguration provides the principle on which autonomous systems can adapt to their changing environments. In a component-based system, the reconfiguration process consists mainly of adding or removing components, changing the connections among them or modifying their functional code.

RIE (Runtime reconfiguration Implementation of Embedded systems) is a component-based implementation methodology. It allows creating C++ components from an extension of the QRML DSL (SDSL, Service-oriented DSL) that was proposed in WP2 [BERG20_1]. 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 include 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.

In Figure 24, 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. From this basic component, all the implementation are derived. 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 presented in this code example:

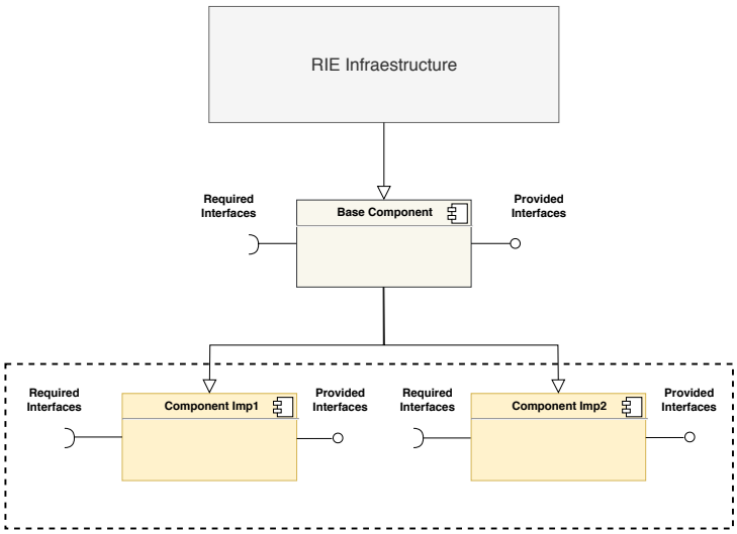


Figure 24: RIE-based component implementation strategy

```
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 provide 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.

During the last year, the RIE reconfiguration strategy has been improved. RIE-based reconfiguration approach is based on the instance concept. An instance is associated with a base component and acts as pointer to the current component implementation. Therefore, service interfaces remain connected in the same way during the whole execution process, while instances may change its associated component dynamically. This approach improves other strategies that requires redefine services during reconfiguration.

Components may have several set-points or configurations. During the component initialization process, a default set-point is assigned. RIE reconfiguration process

consists in changing the current component set-point to another configuration from configuration list.

The system execution process starts with the initialization and blocking of the components, therefore they still cannot provide services. Then the component execution is initialized, so all component services become active. When reconfiguration begins, only components which must be reconfigured are stopped. Afterwards, a new instance of the reconfigured component is associated to the new component version. To begin the execution of the reconfigurated component, it is necessary to wait until old component provided services have already finished. This even is indicated with an specific flag. When the flag indicates that old component provided services have ended, the new components begin the execution, starting to provide services. Finally, the old component version is removed.

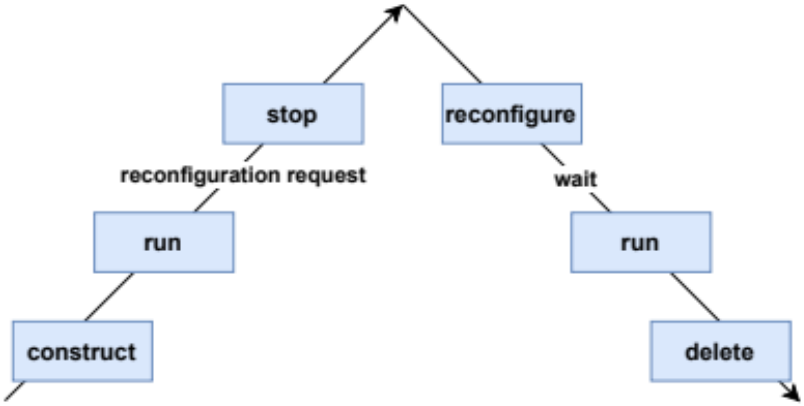


Figure 25: RIE Reconfiguration mechanism

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. For this purpose, grpc services are used. https://grpc.io/
- 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 in which the component is implemented.

RIE-based methodology supports remote component implementations and its dynamic reconfiguration. There are two different platforms (local system and remote component server) and a DNS server to translate from the identifier that is included in the component configuration parameter to the component server IP address and port where component is implemented, as shown in next Figure.

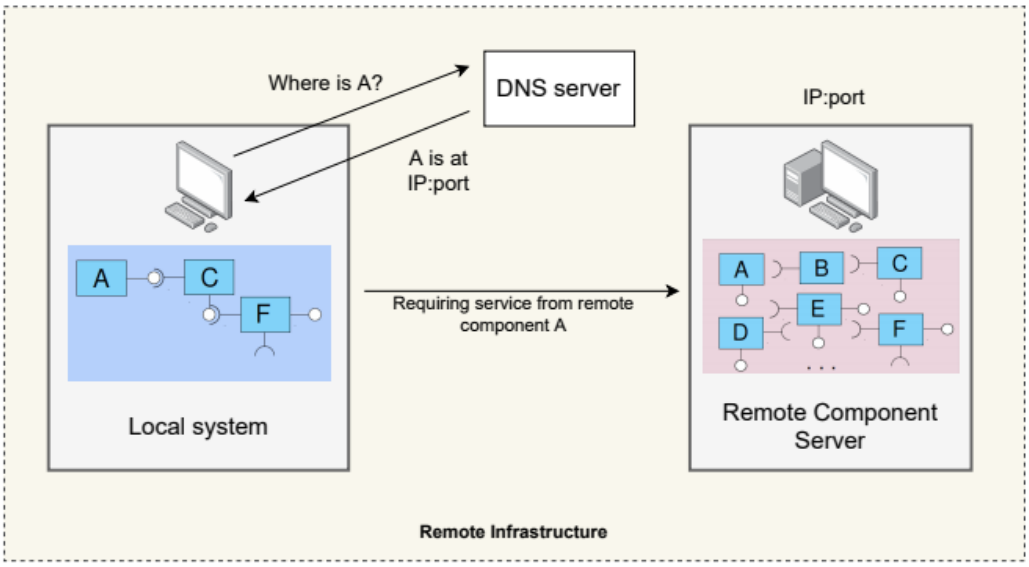


Figure 26: Remote component infrastructure

Local system is implemented in the host platform that manages system reconfiguration, while the remote component server implements a set of components that may be used on local platform. Component server IP address and port are unknown at design time, since components are developed independently from the application. To solve this problem, a configuration parameter ("urlSink") is included to specify the component remote configuration. Furthermore, component configuration may include the remote implementation set-point with the configuration parameter "RIE_Impl_Remote".

At runtime, a DNS server receives queries with an "urlSink" component parameter and returns the IP address and the port to establish the connection between local platform and remote component server. After that, the communication between both platforms is initiated. By default, component implementation connections uses gRPC services for remote procedure calls, although other approaches (e.g. Linux sockets) are also available.

RIE methodology introduces two elements (RIEInterfaceSource and RIEInterfaceSink) for remote component connection. A RIEInterfaceSink object performs a rpc call received by a RIEInterfaceSource object and vice-versa.The approach is presented on Figure 27.

The RIEInterfaceSource class receives an rpc call that is transformed into a call to a certain component interface. The RIEInterfaceSink class is a specific component interface that receives requests to provided services and transforms them into remote procedure calls. This approach facilities automatic component implementation generation. In fact, the remote versions can be automatically generated from the component SDSL model.

The remote component implementations use remote interfaces to provide and require services. These remote interfaces are classes deriving from the component interfaces, where service calls are implemented with RIEInterfaceSource and RIEInterfaceSink. Therefore, the remote interface transforms a remote call into a local service call, due to the existence of a component implementation both on the local platform and on the server.

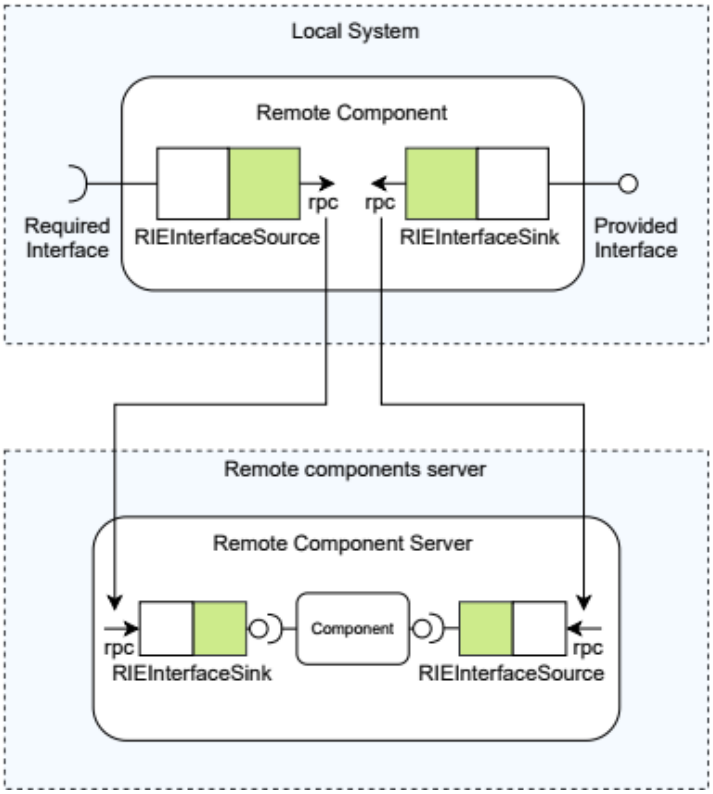


Figure 27: Remote component calls

The local platform implementations are common to all remote implementations because the remote implementation is an instance of the local component in another platform. In fact, they are also managed in the same way that local instance. For this reason, there is a RIE class, RIERuntimeInterface, that is responsible for executing the remote component runtime management functions on the component server, allowing local platform to access the remote component in the same way as if it was a local component.

In 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 address 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. After this, it will manage in the same way that other local components.

Although the current RIE implementation version only supports a remote-procedure-call infrastructure (e.g. gRPC), the methodology can integrate different communication strategies in the same application (e.g gstreamer for some interfaces and gRPC for others).

4.7 Reconfiguration in Managed-Latency Edge-Cloud

The Managed-Latency Edge-Cloud platform (see deliverable D4.5 for more details) is a self-adaptive system which aims at providing soft real-time guarantees on response time to services deployed in an edge-cloud infrastructure. To this end, the platform controls admission and deployment of services submitted for execution in the cloud, periodically monitors performance of the deployed applications, and make short-term predictions of service performance. This allows the system not only to intervene after detecting a violation of application requirements, but also to act proactively ahead of time if needed.

At the highest level of abstraction, the managed-latency edge-cloud infrastructure implements a MAPE-K self-adaptation loop (shown in Figure 28). A single adaptation loop is used to manage both the initial deployment as well as redeployment of services. In fact, redeployment is nearly identical to initial deployment—calculation of real-time requirements is done periodically and takes into account the current placement of services to prevent unnecessary relocations.

Each phase of the adaptation loop has a distinct responsibility:

- **Monitoring**. The monitoring phase is responsible for keeping the internal model of the system up-to-date. In the context of the edge-cloud platform, the controller monitors the state of the Kubernetes (K8S) cloud (nodes, pods, and other entities such as services and deployments) as well as the state and performance of individual applications, e.g., how often.
- **Analysis**. The analysis phase is responsible for finding a deployment configuration (an assignment of application components to nodes in the cloud) that satisfies performance guarantees. A Constraint Satisfaction Problem (CSP) solver is used to find feasible solutions (in which timing requirements can be expected to hold), while the controller is responsible for evaluating the feasible solutions and choosing from among them.
- **Planning**. In the planning phase, the controller determines if the desired configuration differs from the actual configuration and if necessary, prepares a sequence of actions (tasks) to bring the cloud to the desired state.
- **Execution**. In the execution phase, the controller makes actual changes to the cloud platform, following the plan of actions produced in the planning phase. In many

cases, the actions can be executed in parallel, except when there are explicit precedence constraints among tasks.

The four phases execute simultaneously, sharing data through a central **knowledge** component. In its simplest form, the knowledge component can be represented by a single centralized database. However, it is entirely possible for the knowledge component to interface with several storage back-ends that can be used for different purposes.

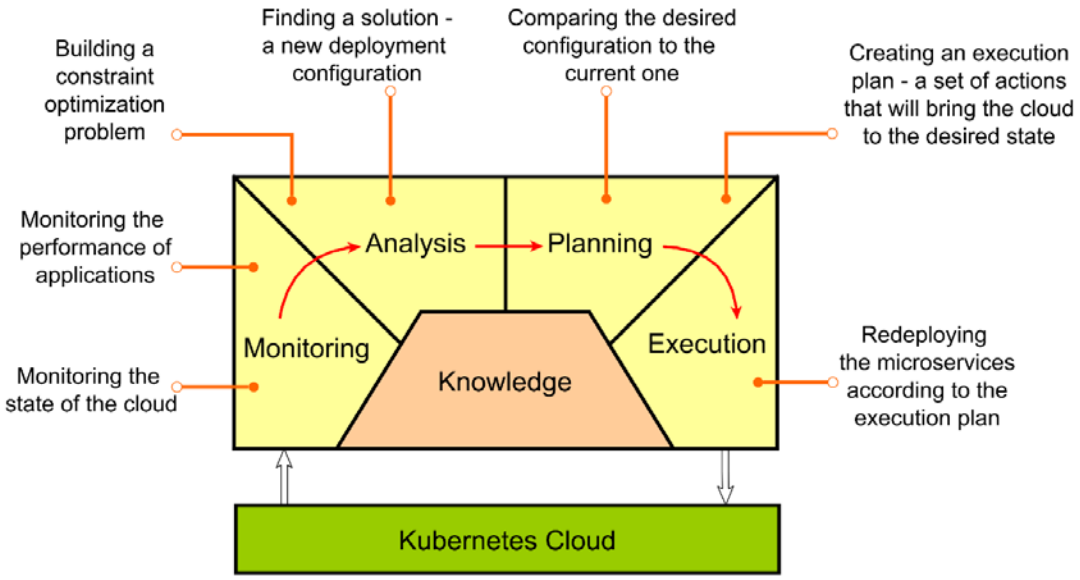


Figure 28: Self-adaptation loop of the managed-latency edge-cloud platform.

Note that this control loop applies only to management of latency in the edge-cloud platform. FitOptiVis systems in the role of edge-cloud applications will implement application-specific higher-level (higher-latency) control loops responsible for configuring the set-points (e.g., resource limits, desired framerate) for a lower-level (low-latency) control loop responsible for achieving the desired set-points on the hardware components.

4.7.1 Detailed Platform Architecture

The architecture of the edge-cloud platform shown in Figure 29 comprises a number of modules, each with distinct responsibilities in the control loop. Yellow modules (need to) run on the master node, green modules do not (need to) run on the master node, and blue modules represent a middleware layer. We now elaborate on the role of individual modules and their interaction with other modules:

- **Event Cache**. The module is responsible for persistent storage of important events, such as changes in application deployment (requests to deploy or undeploy an application) and connections from unmanaged components. Unmanaged components execute outside the edge-cloud platform (e.g., a hardware accelerator) and connect (as clients) to the managed components executing in the cloud.

- **Knowledge**. Provides data storage and query capabilities to modules directly responsible for implementing the MAPE-K control loop. Knowledge data generally concerns cloud nodes (and their subtypes), application types and instances, and component types and instances.
- **Cloud Monitor**. Implements the monitoring phase of the MAPE-K control loop by periodically collecting information about the state of the nodes in the cloud, network latencies, and unmanaged components.
- **Analyzer**. Implements the analysis phase of the MAPE-K control loop and is responsible for finding an application deployment plan that satisfies the timing requirements of all deployed applications. The module is internally split into Solver and Predictor submodules.
  - **Solver**. Responsible for finding the best deployment plan within a given time limit. Takes into account node utilization, network latencies, and predictions of component performance in deployment scenarios considered.
  - **Predictor**. Predicts performance of managed components, taking into account the hardware they are running on and the load induced by other components running on the same hardware.
- **Planner**. Implements the planning phase of the MAPE-K control loop, which means identifying differences between the current application deployment and the desired deployment. Constructs an ordered execution plan of tasks that need to be executed to transition the system to the next state.
- **Cloud Executor.** Implements the execution phase of the MAPE-K control loop by executing planned tasks either on the Kubernetes cloud, or on the other (Managed and Unmanaged) controllers.
- **Managed Controller**. Responsible for invoking probes on managed components and for reconnecting dependencies of managed component instances. Can access all Node Controllers at runtime.
- **Unmanaged Controller**. Responsible for reconnecting dependencies of unmanaged component instances from one managed instance to another, invoking probes on the client (which invoke managed components) to observe managed component performance including communication latency, and monitoring the state of unmanaged components.
- **Node Controller**. Runs on each node and monitors the utilization of a particular node and of all the components executing on that node (using standard K8S facilities for resource monitoring). In addition, it serves as a proxy to managed component instances for the Managed Controller.
- **Probe Controller**. Serves as a central entity through which all requests for probe invocation (on Managed and Unmanaged components) have to pass. Caches and forwards the results of probe invocations.
- **Network Controller**. Responsible for making changes in network configuration and for collecting network utilization data and connection latencies.

On each node, the information about a service obtained during the assessment phase is used to assign each deployed service the resources needed to perform its tasks within the timing constraints. This resource allocation is strictly enforced using features of the operating system, containerization technology, or the virtualization platform. Specifically, we rely on resource allocation features of Docker and Linux cgroups. This is necessary to prevent services from exceeding their allocated share of resources (due to, e.g., a

sudden spike in the number of clients), which could have a negative impact on the execution time of other services.

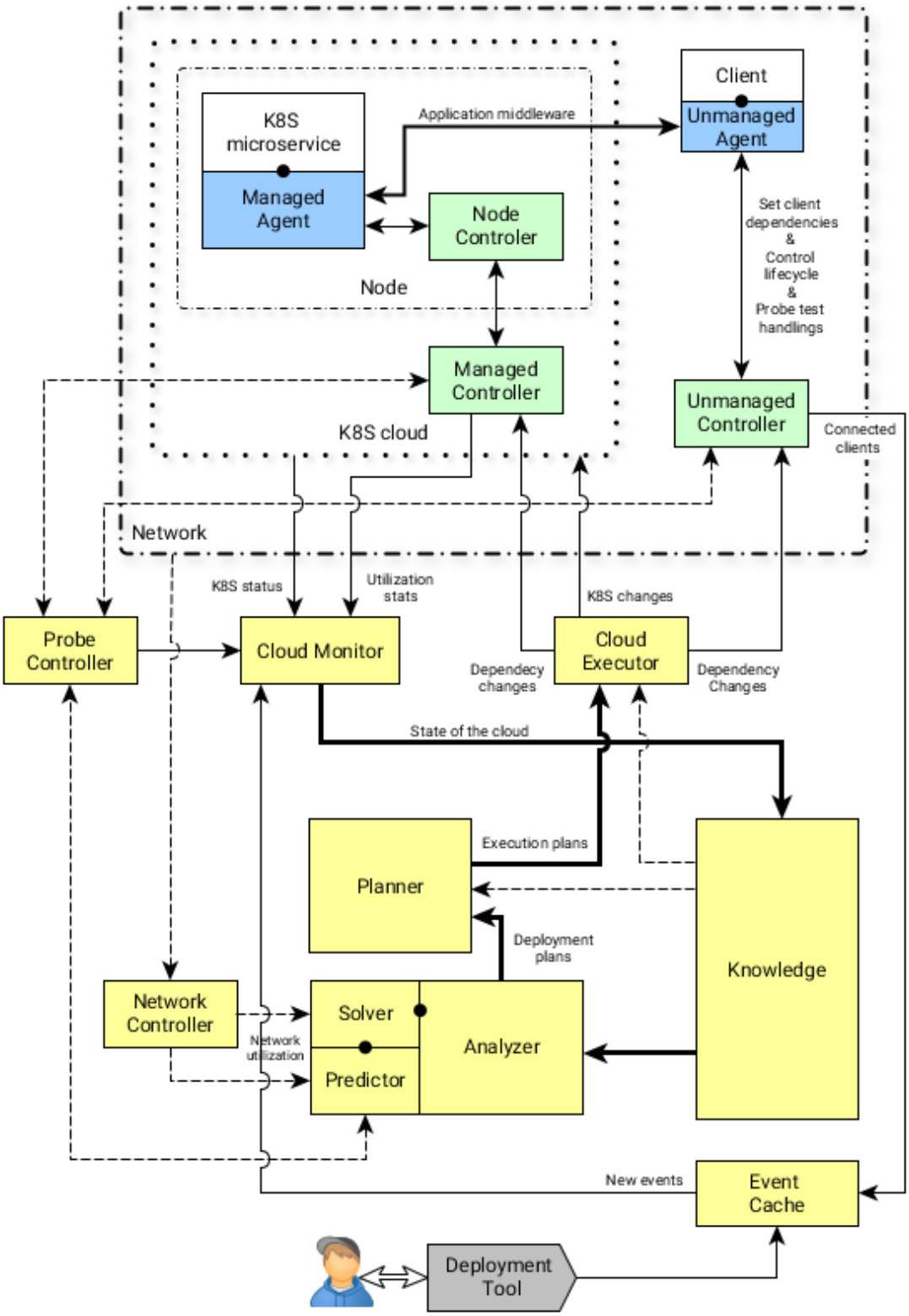


Figure 29: Detailed architecture of the managed-latency edge-cloud platform.

4.7.2 Performance and Interference Models

To adaptively control deployment and redeployment of components in edge-cloud and thus to probabilistically guarantee end-to-end response time, the platform needs to build a model of application performance. This model needs to capture several modes of execution: baseline performance, when the application is exercised in isolation, performance under constrained resources, and performance in presence of other co-located applications sharing the physical hardware through virtualization.

Because we do not require the developer to provide the platform with apriori knowledge about application performance and resource requirements, the cloud platform needs to build the application performance model using experimental evaluation.

The model then is used to predict application performance in different situations, especially during admission control (when deploying a new application), and when optimizing the deployment of existing applications (to ensure that real-time guarantees are met, or to manage the utilization of cloud resources).

An important aspect of performance that the cloud platform needs to take into account is performance interference on shared resources (CPU caches, memory and IO bandwidth, etc.) when co-locating multiple virtual machines and/or containers on the same physical machine.

On the other hand, we generally consider the underlying network bandwidth unlimited for modelling purposes. The rationale behind this assumption is that edge-cloud applications are likely to be latency-sensitive, but not necessarily bandwidth-intensive—that would defeat the primary purpose of edge-cloud, which is to reduce communication latencies due to distance.

We also assume that edge-cloud infrastructure can generally be private, i.e., with significant level of control (like in hospital use cases). Consequently, we assume that the network infrastructure can be configured to assign time-critical network traffic a QoS class with high priority; that latency-sensitive services with guaranteed response time requirements will not saturate the network with bulk transfers; and that applications with excessive bandwidth requirements can be dealt with by proper network infrastructure design. In particular, if latency-sensitive traffic needs to coexist with bulk traffic on the same network infrastructure, we assume that solutions based on Time-Sensitive Networking will be used.

4.7.3 Performance Prediction of Co-located Workloads

One of the key responsibilities of the Analyzer module (see platform architecture in Figure 29) is finding and analysing deployment alternatives. The analysis primarily concerns application performance prediction, providing the adaptation controller with data for making decisions—both when considering an application for admission as well as when reacting to violation of application's timing requirements.

The Predictor part of the Analyzer module uses a novel performance prediction algorithm which is based on statistical characterization of application performance measurements followed by a similarity comparison, revealing performance dependencies between background workloads (i.e., services).

We first use performance measurements to build a structured data set and the, whenever a performance prediction of a particular scenario is needed, the relevant prediction data are extracted into a linearized data-fitting model. This model is then solved by a constrained least-squares method, giving a reliable order statistics estimate of application performance, including its fidelity.

To build the initial data set, we perform a number of measurements for a number of scenarios involving one or more workloads. There is always a scenario in which each workloads executes in isolation, without any other workloads running in the background. For each workload, we also include various combinations of background workloads. Because this may quickly become computationally infeasible, we generally focus on collecting information for pairs of co-located workloads, which reveals first-order performance impact, i.e., how applications influence each other on given hardware platform. Scenarios involving three or more workloads are sampled depending on available resources.

For each scenario, we collect measurements on a number of parameters which characterize the application behaviour. In addition to response time, this includes CPU utilization, number of I/O operations, and memory utilization. To ensure robustness of the predictor, each scenario is measured multiple times to properly sample the influence of factors that can influence the measured parameters, but are beyond our control, such as virtual memory layout, file system state, or just-in-time compilation. With the initial data collected, we can start predicting application performance in different scenarios.

The prediction algorithm consists of three phases, and is summarized in the schema shown in **Errore. L'origine riferimento non è stata trovata. Errore. L'origine riferimento non è stata trovata.**. Here we discuss the individual phases in more detail:

1. **Data pre-processing**. The first phase represents all computations that can be performed apriori to save the computational costs in later phases. The goal is to compute a number of statistical characteristics (for each of the given scenarios) in order to capture dependencies of all parameters of interest on the measurement conditions. This includes information about statistical distribution of the measurements, i.e., the sample mean and median, selected sample percentiles, standard and relative deviations, standard error, and the difference between the sample maximum and minimum values.
   While the characteristics such as mean or median capture typical behaviour, the sample maximum and minimum capture information about extremes. The difference between the typical and extreme behaviour is used to effectively penalize measurements with lower fidelity, improving performance prediction reliability.
   We also compute various quantities that allow revealing dependencies between performances of different workloads. In particular, these include slow-down parameters corresponding to the difference between sample percentiles of measured parameters for cases when a workload executes in isolation and when it executes together with other workloads.
2. **Task fitting**. Given the initial data, their statistical characterization, and a user-specific prediction requirement (i.e., a question), we first need to detect precomputed scenarios relevant for the prediction. We allow two types of scenario questions:
   - Q1: performance prediction for one of the already tested workloads, $\mathbf{W}_i$.

- Q2: performance prediction for a new workload, $\mathbf{W_{n+1}}$, for which we have data measured in isolation.

The situation is simpler for Q1. The prediction must be based on the statistical characteristics of the scenarios involving $\mathbf{W_i}$. We therefore build a prediction model using all scenarios involving $\mathbf{W_i}$, except the one in which $\mathbf{W_i}$ executes in isolation. This gives us a data fitting problem, modelling the unknown correlation between the question and the preselected initial scenarios, which we then solve using the constrained least-squares method with non-negative constraints (NLS).

For Q2, the prediction is based on finding an existing workload $\mathbf{W_j}$ that most closely resembles the new workload $\mathbf{W_{n+1}}$. To find such a workload, we first compute the statistical workload characterization (see phase 1) for the scenario in which $\mathbf{W_{n+1}}$ executes in isolation and compare it to characterizations of other workloads executing in isolation. Using some similarity measure, e.g., a weighted vector norms of the difference between mean, median, and deviation for the most relevant measured parameters, we look for the lowest difference (best match), producing $\mathbf{W_j}$. Finally, we incorporate the statistical characterization of $\mathbf{W_{n+1}}$ in the data set and "rephrase" Q2 as Q1 with $\mathbf{W_j}$ in the role of $\mathbf{W_i}$ serving as a proxy for the new workload $\mathbf{W_{n+1}}$.

3. **Data-based prediction**. In the last phase, we use a weighted combination of workload dependencies to predict the behaviour in the scenario from Q1 or Q2. Specifically, we estimate percentiles of expected performance of $\mathbf{W_i}$ in Q1 by shifting the percentile observed for $\mathbf{W_i}$ executing in isolation by a linear combination of estimated weighted slowdowns.

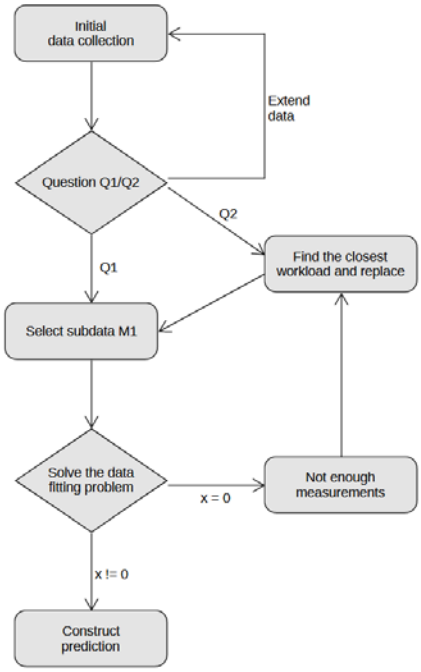


Figure 30: Overview of the performance prediction algorithm.

The interactions among co-located services sharing the underlying physical resources are generally complex, and often non-linear—especially when the physical resources are nearing exhaustion. Consequently, the prediction accuracy varies with different combinations of applications and resources used, and cannot provide actionable results for all possible scenarios.

To ensure that the predictor can be used with confidence within the adaptation loop, it is critical to establish the predictor's operational boundaries and ensure that the managed system stays within the boundaries. The boundaries can be expressed as limits on the utilization of the CPU, memory, and IO resources used to characterize the workloads.

While our system currently does not support automatic discovery of the operational boundaries, our initial evaluation indicates that they could be established experimentally for a particular platform. We expect that this could be turned into an automated procedure.

The work presented here has been accepted for publication in the Journal of Systems and Software.

4.8 Situation-aware reconfiguration in closed-loop control

While vision is an attractive alternative to many sensors targeting closed-loop controllers, it comes with high time-varying workload and robustness issues when targeted to edge devices with limited energy, memory and computing resources. Replacing classical vision processing pipelines, e.g., lane detection using Sobel filter, with deep learning algorithms is a way to deal with the robustness issues. At the same time, hardware-efficient implementation is crucial for their adaptation for safe closed-loop systems. However, while implemented on an embedded edge device, the performance of these algorithms highly depends on their mapping on the target hardware and situation encountered by the system. That is, first, the timing performance numbers (e.g., latency, throughput) depends on the algorithm schedule, i.e., what part of the AI workload runs where (e.g., GPU, CPU) and their invocation frequency (e.g., how frequently we run a classifier). Second, the perception performance (e.g., detection accuracy) is heavily influenced by the situation -- e.g., snowy and sunny weather condition provides very different lane detection accuracy. These factors directly affect the closed-loop performance, for example, the lane-following accuracy in a lane-keep assist system (LKAS).

The key steps involved in the situation-aware design and reconfiguration are [DE21]:

1. Situation definition: Situations are combinations of environmental factors that potentially influence closed-loop performance. Figure 31 gives an example illustration of a situation definition for an LKAS considering lane types, road layouts and different weather or environmental conditions.

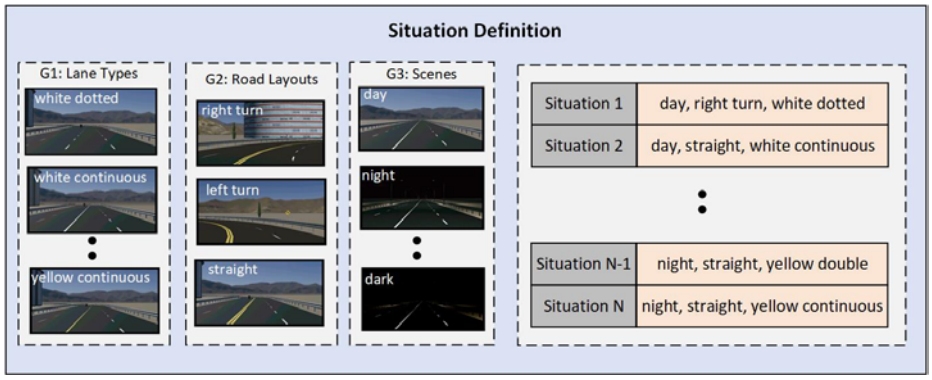
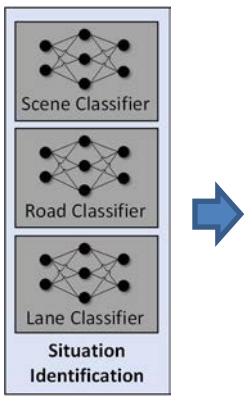


Figure 31: Situation Definition for an LKAS

2. Situation-aware Characterization: The goal is to identify the set of LKAS parameters that perform the best under a specific situation at design time. First, we determine the system parameters sensitive to the operating situation using Monte-Carlo simulations of the entire system using the HiL setup in the IMACS framework. We observe that the approximations in the image signal processing (ISP), region-of-interest (ROI) selection in perception and vehicular speed selection in the controller are the system parameters that heavily influence the closed-loop QoC. We will refer to these parameters as configurable knobs.
3. Situation identification: For selecting the best-tuned knobs, we need to identify the situations under which LKAS is operating at runtime. For this, three different light-weight CNN classifiers (scene, road & lane) are considered based on the Resnet-18 architecture, as shown in Figure 32 (which also gives a brief overview of the three classifiers implemented in NVIDIA AGX Xavier).



Classifiers	Details
Road Classifier	*dataset*: 5866 images (train: 5353, val: 513) *output classes*: straight, left turn, right turn *classification accuracy*: 99.92% *Profiled runtime on NVIDIA AGX Xavier*: 5.5 ms
Lane Classifier	*dataset*: 4781 images (train: 3939, val: 842) *output classes*: white continuous, white dotted, yellow continuous, yellow double *classification accuracy*: 99.97% *Profiled runtime on NVIDIA AGX Xavier*: 5.5 ms
Scene Classifier	*dataset*: 4703 images (train: 3892, val: 811) *output classes*: day, night, dark, dawn, dusk *classification accuracy*: 99.90% *Profiled runtime on NVIDIA AGX Xavier*: 5.5 ms

Figure 32: Situation identification and classifiers details.

4. Dynamic runtime reconfiguration: For runtime reconfiguration, the input frames are analyzed first, and the operating situation is determined using the classifiers identified for a situation. Post situation identification, best situation-specific knob tunings are selected based on the situation-aware characterization. Knobs are then (re-)configured based on the situation as shown in Figure 33.

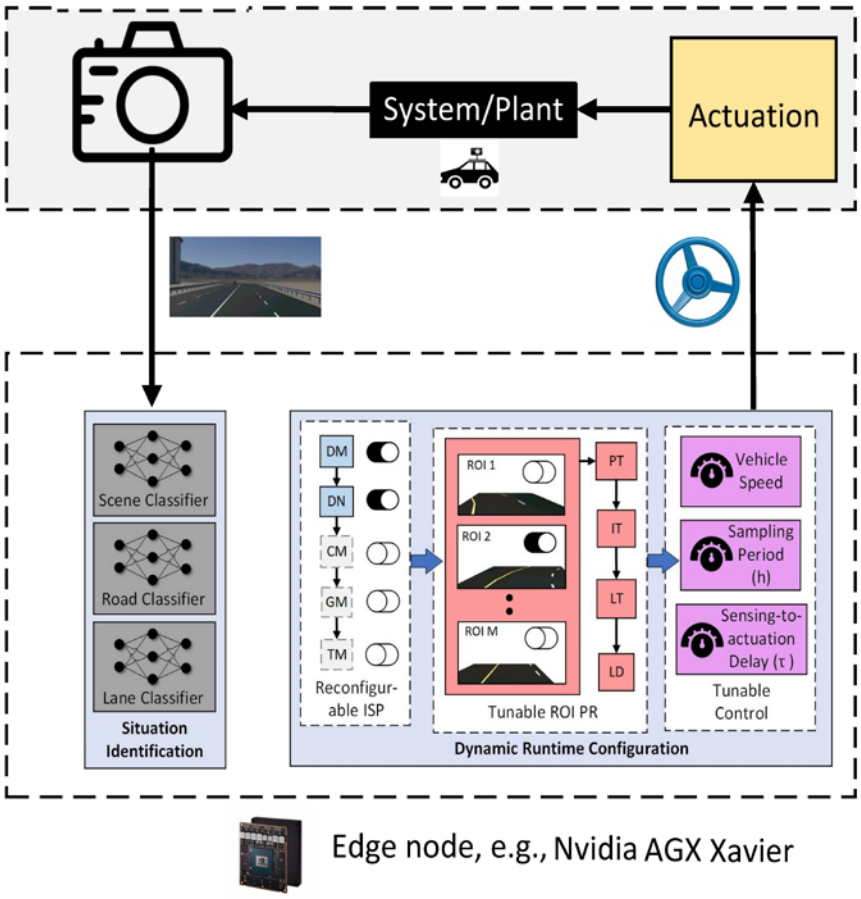


Figure 33: Dynamic runtime reconfiguration and HiL setting for LKAS

A situation-aware design of AI perception where the idea is to define the situations by a set of relevant environmental factors (e.g., weather, road etc., in an LKAS, shown in Fig. 30) provides robust LKAS designs with 32% better performance compared to traditional approaches. We design the learning algorithms and parameters, overall hardware mapping and schedule, taking the situation into account. We show the effectiveness of our approach considering a realistic LKAS case study on heterogeneous NVIDIA AGX Xavier platform in a hardware-in-the-loop (HIL) validation using the IMACS framework (illustrated in Figure 34).

The IMACS framework (http://www.es.ele.tue.nl/ecs/imacs) consists of a simulation engine, an interface between the simulation engine and design-time tools like MATLAB and SDF3, and an interface between the simulation engine and run-time implementation either as software or in hardware. The simulation engine simulates the physics, environment, sensors and actuators. Algorithm design and analysis are done using design-time tools like MATLAB and SDF3. Once we have a good design, the codes are generated or developed and simulated in a software-in-the-loop framework. Further, the codes are validated by an implementation in the NVIDIA platform and an HiL simulation.

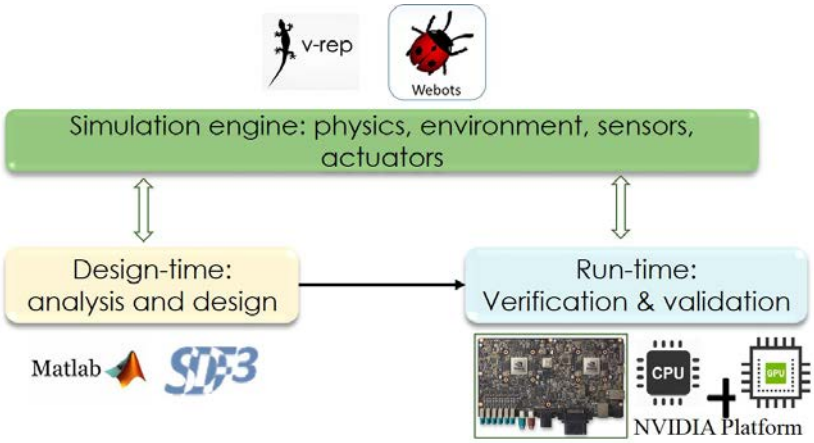


Figure 34: IMACS framework

Situations are created in the physics simulator, and the software simulated in IMACS adapt to the situations by dynamic runtime reconfiguration.

5. Runtime Monitoring, Profiling and Measuring

This chapter describes the monitoring, profiling and measuring support developed within the FitOptiVis project, and also reports practical setups related to FitOptiVis use cases. First, a reference platform for monitoring systems in FitOptiVis is described: 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: their development comes 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.

With the goal of properly identifying the monitoring techniques developed within the FitOptiVis project, a reference architecture of a cloud-edge computing system has been considered, shown in Figure 35. 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. For these reasons, a reference model of a monitoring action has been developed, with the sake of clarity. It is reported in Figure 35 and shows the actors involved in a monitoring action.

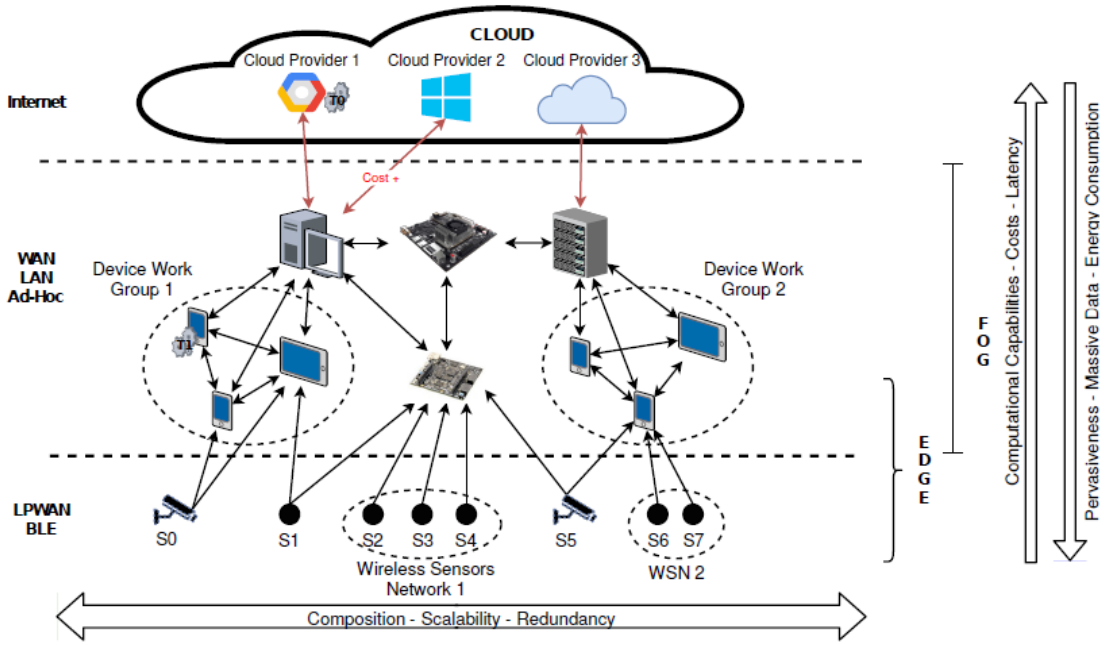


Figure 35: A reference architecture for Cloud-Edge computing systems [ZAN18]

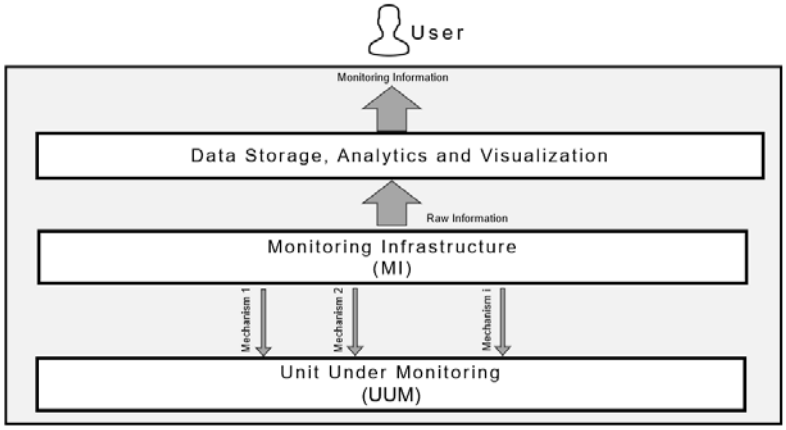


Figure 36: Reference model of a monitoring action. Independently on how many layers are involved, some components can be always identified: a *Unit Under Monitoring* (UUM), a *monitoring infrastructure* that extracts raw information from the UUM by means of hardware/software mechanisms, and a *Data Storage, Analytics and Visualization* part that organizes, filters and parses the raw information to obtain the monitoring information

An overview of the connections between the work of Task 4.2 partners and the FitOptiVis use-cases is reported in Table 5.

Table 5: Task 4.2 participant works vs Use-cases

	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8	UC9	UC10
ITI				X						
CUNI		X	X	X					X	X
TUT		X								
UC										X
UNIVAQ	X									
UNICA	X									
UNISS	X									
UTIA								X		
UGR			X						X	
TASE										X
HURJA		X	X							
NOK		X								
HIB			X							
7SOLS									X	

The work of Task 4.2 provides the foundation for the development of a tool to guide designers in the selection of monitoring systems for their applications. The tool is available at https://monicatool.cloud [VAL2_21].

5.1 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 JOINTER, a framework to build custom monitoring systems for edge-computing platforms. This section reports details about these enabling solutions.

5.1.1 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 set of system-specific 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 case, 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 acted upon in an autonomous fashion. This requires that the monitoring system also provides means for accessing the data (e.g., through time-based queries) to external agents.

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 provides 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.

5.1.1.1 Overview

The FIVIS system provides support for aggregating data from multiple sources and enables executing customized analyses on the data to provide both 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 and widgets implementing control elements in dashboard user interface (e.g., time range selector, signal selector, chart

legend). 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.

Architecture

The architecture of the system is shown in Figure 37Figure 37. The system consists of a server part (hosted by the system provider) 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 be customized to use an application-specific “data pump task” to pull data from an external system.

Signal data obtained through observation (and pushed to the system) represents *master data*. The system keeps the master data in a MySQL database, but only works 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 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.

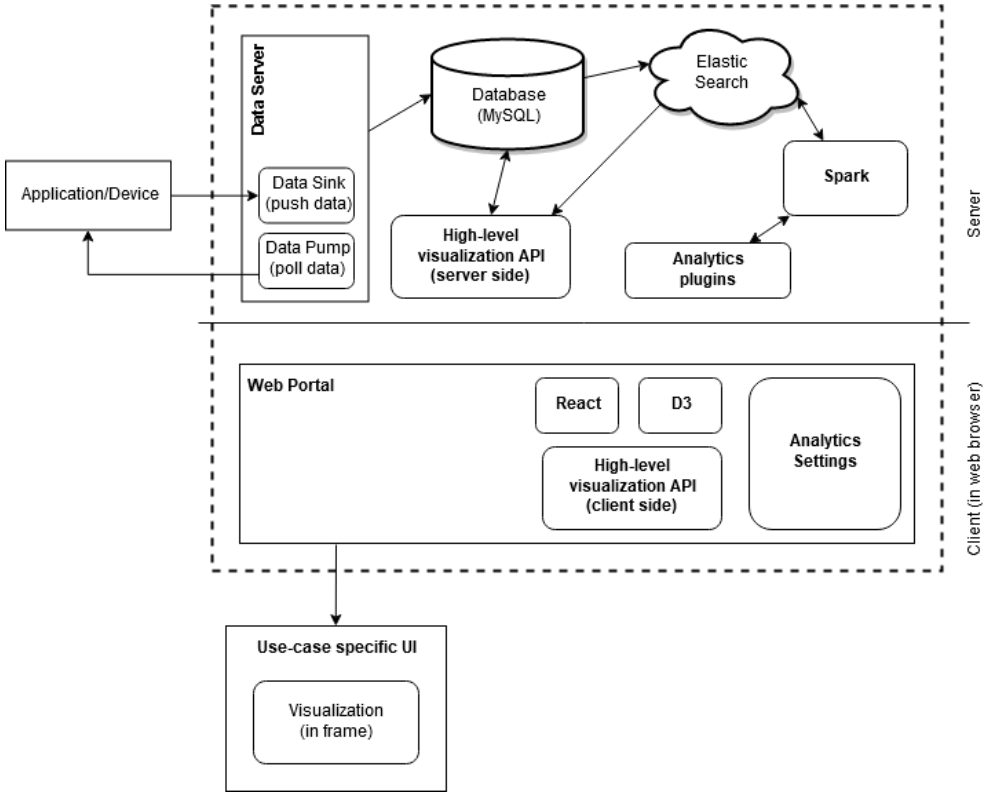


Figure 37: Architecture of the FIVIS system

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 38 below. The data is conceptually organized into uniquely identified signal sets. Each signal set groups one or more signals, where each signal represents a stream of typed values, e.g., readings from a sensor.

Each signal set is described by a *schema*, which consists of an ordered set of *signal descriptors*, one for each signal. A signal descriptor captures the signal name and the type of values represented by the signal. The system currently supports logical, numeric, textual and temporal values, as well as base-64-encoded binary objects.

The actual signal data is 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 for each record to have a unique identifier for which ASCII ordering is well-defined. The ordered identifiers allow the system to determine if and where new records with signal data were inserted, which is necessary for proper scheduling of data analysis tasks—if new records are appended after the existing data, the system may only schedule an incremental analysis for the new records. If (for some reason) new records are inserted in between existing records, the system may need to recompute the analyses for all records.

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 visualization templates.

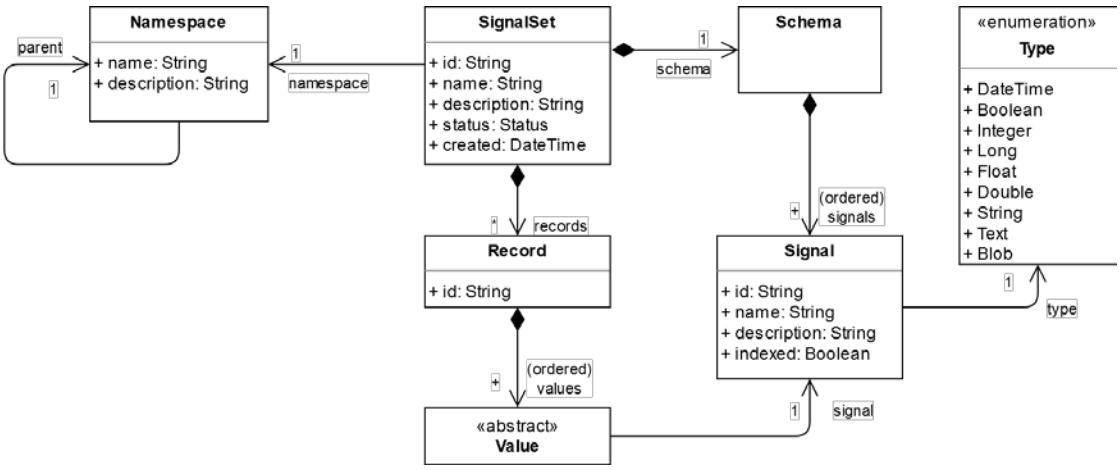


Figure 38: Meta-model of master (sensor) data in persistent storage

Schema Design

Apart from the basic structuring of data into signals and signal sets, the system does not impose any other constraints on the data. However, it is important to keep in mind that the system is primarily intended for storing time-series data and to **avoid** treating signal sets as tables in a relational database.

The key difference is that in a relational database, a table typically represents an entity type, and each row of a table represents (part of) a single entity (uniquely identified by the values of the primary-key attributes). The purpose of queries in a relational database is to find (or update) the state of the entities matching criteria expressed as restriction over attribute values.

In a time-series database (FIVIS), a signal set represents an actual entity (a data-producing process with its own timeline), uniquely identified by the signal set name. Each record (row of signal values) in a signal set then reflects the state of the entity at a certain moment. The purpose of queries in a time-series database is to restrict the set of signal records for a known entity to a specific time frame. This difference needs to be taken into account when designing the schema for storing sensor data in FIVIS.

The best way to structure data into signal sets is to think of signal sets as values of signals produced "at the same time" by a single process. This means that all signal values are temporally-correlated, i.e., associated with the same moment. If there are multiple processes producing different signals, they should be stored in separate signal sets, because a signal set identifies a process or an independent timeline.

For example, let us consider that we need to store data from a system monitoring agent. The agent monitors the utilization of system resources (e.g., CPU utilization, I/O bandwidth, memory, etc.) and uses different threads to sample performance counters corresponding to different resources. In this case, each thread represents an independent activity with its own timeline. Consequently, the data produced by each thread should be stored in a different signal set, because each thread will associate its own timestamp with the values stored in a record. In contrast, if the agent sampled all

performance counters from a single (main) thread, then the corresponding schema would consist of a single signal set. If such a single-threaded agent ran on multiple devices, then the schema should contain one signal set for each device.

In summary, the entities in a time-series database correspond to data-producing processes. The state of an entity at a given time is represented by a set of attribute values, and a sequence of such sets captures the evolution of a single entity in time. This can be directly mapped to FIVIS concepts of a signal set (a data-producing process with its own "timeline"), a signal (named process attribute with a specific value type), and a record (set of attribute values at a given time).

Just like it is possible to store time-series data in a relational database (including timestamp in the primary key along with entity identifier), it is also possible to store data from multiple data-producing processes in a single signal set. This is, however, not a good or recommended schema design. The signal records in such a signal set would be sparse (i.e., only a subset of signals would be valid at the time associated with the record), which greatly complicates time-series data visualization and analysis, because data processing code cannot rely on all signal values to be valid all the time (e.g., when computing derived signal from two other signals in a record). Visualization queries then need to perform database-like filtering to obtain the required data, which may turn out to be either impossible or extremely inefficient on storage backends optimized for time-series data.

5.1.1.2 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 lack resources to issue an HTTP request with JSON payload. In this case, the system can be customized to poll the device through a remote agent (e.g., IoT gateway) 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 required so far. Partners interested in using the system in this mode should contact 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 are optional (depending on the 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 values for each signal in a signal set. Schema is an object with named slots, where the name of a slot represents the signal name and the string value of a slot denotes the signal type. The following types are currently supported:
  - `boolean`, a logical type, accepts boolean values `true` and `false`,
  - `integer` and `long`, a 32-bit and 64-bit signed integer types, respectively,
  - `float` and `double`, a 32-bit and 64-bit floating point types, respectively,
  - `string`, represents a short string value (typically used for keywords or enumerations) limited to 255 characters ,
  - `text`, represents a text document of unlimited length,
  - `datetime`, an ISO8601-formatted string representing absolute time with millisecond resolution,
  - `blob`, a BASE64-encoded string representing arbitrary (often binary) data; the data is not interpreted by the storage backend (the signal cannot be used in queries), but can be used by visualization.

  Note that the values for signal types `string`, `text`, `datetime`, and `blob` are all transported as JSON strings, but the system will treat them differently. Notably, the length will be limited for the `string` type, a specific format is expected for the `datetime` type, and specific encoding is expected for the `blob` type.

  The `schema` field can be omitted in most 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.

  Each signal set has an implicit signal named `id`, which represents a unique identifier for a set of signal values (a record). The identifier is a free-form string and the sender of the data is responsible for providing the identifier value for each record. The only requirement is that ASCII ordering must be well defined on the identifier, because the system uses it to establish record ordering.

  The `id` signal is not part of the user-defined schema and MUST NOT be included in the schema object, but MUST be included in each data record.
- `data: object[]`, an array of records 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 the record's unique identifier.

The following listing shows an example of sensor data payload in the JSON format. The `schema` object defines four signals with different types (note the absence of the implicit `id`

signal), and the `data` field contains two records, each including a record identifier (the `id` signal) in addition to the values of the four user-defined signals:

```
{
    "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
        }
    ]
}
```

Figure 39. Example of sensor data payload in JSON format.

Posting Signal Data

Sending data to FIVIS requires sending a `POST` request containing a JSON document with the signal data payload to the `/api/signals` API 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 with the payload 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 FIVIS system user interface through the **API** submenu of the **Account** menu. The following listing shows the payload from the above example being posted to FIVIS 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
    }
  ]
}'
```

Figure 40. Payload posted to FIVIS using the `curl` utility.

We provide a simple FIVIS client library (which internally uses `libcurl`) for resource-constrained devices with limited support for shell or Python. The library (together with a sample CPU monitoring application supporting batched/delayed data transmission) is available on GitHub: https://github.com/d-iii-s/FIVIS-client

Alternatively, ITI provides a plugin to the Telegraf system agent, which allows system-level and application-specific 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 allows keeping the master data immutable, yet visualizing 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 (without touching the master data), 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 would 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;
```

Deleting Signal Sets

The entire signal set can be deleted by sending a `DELETE` request with an empty body to the `/api/signals/<sigset-cid>` API endpoint, where `<sigset-cid>` is a fully qualified signal set identifier containing both the signal set namespace and the signal set identifier.

In most cases, the namespace will correspond to the partner identifier, because each partner has access to a partner-specific namespace in which it can create and delete signal sets. For example, if a partner with an identifier `'cuni'` wanted to delete a signal set with an identifier `'test'`, the corresponding fully qualified signal set identifier would be `'cuni:test'`. The following listing shows how to delete such a signal set using the `curl` utility:

```
curl https://api.fivis.smartarch.cz/api/signals/cuni:test
  --request DELETE \
  --header 'access-token: 8e3f4bf6bec954b40a0ec08ab0dc0c11d0d18fed'
```

5.1.1.3 Data Processing

The IVIS framework underneath FIVIS provides the concepts of tasks and jobs. 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.

5.1.1.4 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, which 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 are 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.

5.1.1.5 System Status

An instance of the system hosted by CUNI has been set up and is available to project partners. Parts of FIVIS dealing with data ingestion, storage, and visualization are fully operational, which allows partners to send monitoring data to FIVIS and visualize it in custom dashboards. CUNI has developed use-case-specific dashboards for several partners involved in use cases UC-3 and UC-9 (c.f. Section 5.2 below).

Some parts of the system are being finalized, specifically the integration of QRML component model with monitoring data (which will enable interactive exploration of application architecture and associated quality attributes), and the integration of customized analysis tasks that can produce reconfiguration triggers to other systems, which will allow FIVIS to serve as a controller in a high-level adaptation loop.

5.1.2 QRML extension to express monitoring requirements

UC has extended QRML [D2.1][BERG20] to SDSL in order to support monitoring requirements. The main objective is to provide automatic monitor code generation from SDSL.

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

The monitor type declaration identifies the monitor and define 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.

The following figure 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 41. Monitor declaration using SDSL.

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, next figure 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 42. Example of monitor instance in a component.

UC implementation methodology support the Linux LTTng framework and the FIVIS infrastructure. System monitoring is necessary in resilient systems since they need some mechanisms to guide reconfiguration process.

Figure 41 shows an example of FIVIS monitoring usage, where some latency parameters are represented. Additionally, Figure 42 shows an example of LTTng monitoring traces.

Figure 43 FIVIS example graph

```
fernando@fernando-HP-Compaq-Pro-6300-MT:~/Escritorio/BAC_Example_TFM/Local/BAC_Final/example/fitoptivis/Monitors/lttng$
 babeltrace -f trace --input-format=lttng-live  net://localhost/host/fernando-HP-Compaq-Pro-6300-MT/live_test_session --no-delta
[10:51:48.234485982] VideoTrace:Performances: { cpu_id = 2 }, { field_Comp = "FaceDetection", field_latency = 31060 }
[10:51:48.245128240] VideoTrace:Performances: { cpu_id = 2 }, { field_Comp = "AccessControl", field_latency = 7 }
[10:51:48.248736023] VideoTrace:Performances: { cpu_id = 2 }, { field_Comp = "FaceRecognition", field_latency = 45254 }
[10:51:48.278070802] VideoTrace:Performances: { cpu_id = 2 }, { field_Comp = "FaceDetection", field_latency = 29325 }
[10:51:48.289120059] VideoTrace:Performances: { cpu_id = 2 }, { field_Comp = "AccessControl", field_latency = 7 }
[10:51:48.292814033] VideoTrace:Performances: { cpu_id = 2 }, { field_Comp = "FaceRecognition", field_latency = 44012 }
[10:51:48.322987103] VideoTrace:Performances: { cpu_id = 2 }, { field_Comp = "FaceDetection", field_latency = 30163 }
[10:51:48.333870582] VideoTrace:Performances: { cpu_id = 2 }, { field_Comp = "AccessControl", field_latency = 8 }
[10:51:48.337493208] VideoTrace:Performances: { cpu_id = 2 }, { field_Comp = "FaceRecognition", field_latency = 44613 }
[10:51:48.366407710] VideoTrace:Performances: { cpu_id = 2 }, { field_Comp = "FaceDetection", field_latency = 28904 }
[10:51:48.377235507] VideoTrace:Performances: { cpu_id = 2 }, { field_Comp = "AccessControl", field_latency = 7 }
[10:51:48.380680534] VideoTrace:Performances: { cpu_id = 2 }, { field_Comp = "FaceRecognition", field_latency = 43121 }
[10:51:48.409985076] VideoTrace:Performances: { cpu_id = 2 }, { field_Comp = "FaceDetection", field_latency = 29294 }
[10:51:48.420910537] VideoTrace:Performances: { cpu_id = 2 }, { field_Comp = "AccessControl", field_latency = 7 }
[10:51:48.424617658] VideoTrace:Performances: { cpu_id = 2 }, { field_Comp = "FaceRecognition", field_latency = 43872 }
[10:51:48.454089751] VideoTrace:Performances: { cpu_id = 2 }, { field_Comp = "FaceDetection", field_latency = 29462 }
[10:51:48.464938308] VideoTrace:Performances: { cpu_id = 2 }, { field_Comp = "AccessControl", field_latency = 8 }
[10:51:48.468943475] VideoTrace:Performances: { cpu_id = 2 }, { field_Comp = "FaceRecognition", field_latency = 44253 }
```

Figure 44 LTTng monitoring traces example

5.1.3 JOINTER framework to build custom hardware monitoring systems

5.1.3.1 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: JOINTER (acronym of *JOINing flexible monitoring with heTERogeneous architectures*) allows the generation of monitoring systems targeting architectures implemented on FPGAs. JOINTER starts from a basic library of element, takes as input the monitoring requirements and the hardware architecture, providing as output a monitored system.

5.1.3.2 Monitoring system composition

JOINTER 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.

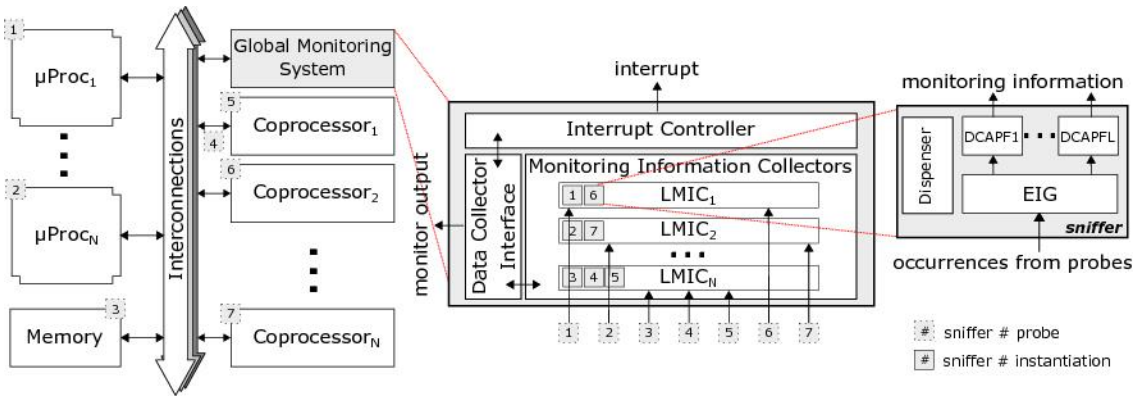


Figure 45: Monitor generation in JOINTER

The monitoring systems automatically generated by JOINTER are based on a number of *sniffers* distributed within the hardware architecture of the system under monitoring. The generation of monitoring systems is reported in Figure 45: sniffers are referred as small boxes with a number, distributed inside a hardware reference platform for heterogeneous embedded systems, shown on the left.

Each sniffer is composed of an Event Instance Generator (EIG) and multiple Data Capture and Filters (DCAPFs). The EIG allows to connect to specific interconnection points of the monitored platform, producing event instances associated to events to be monitored. Event instances are then managed by DCAPFs to compute metrics: each DCAPF is associated to a metric.

Sniffers are managed by means of some Local Monitoring Information Collectors (LMICs), that allow their initialization, their control, and the collection of results. LMICs share a unique register space.

The register space is exposed to other system components by means of a Data Collector Interface (DCI): the role of DCI is to make the monitoring results from sniffers, stored in LMICs register space, available to be accessed by other components (for example, by means of AXI-based protocols).

All the structure is highly customizable, and it is provided as open source at the following link: https://github.com/alkalir/jointer.

5.1.3.3 Interface

JOINTER is provided with bare-metal APIs to interact with the generated monitors. In the next future, Linux based APIs will be provided.

5.2 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 (referred to as instances) to satisfy the different requirements coming from use cases, WP3, and WP4 tasks, are reported.

5.2.1 Monitoring in 3D industrial inspection system

ITI has developed a 3D Industrial Inspection system (Zero Gravity 3D) which uses sixteen edge computer boards connected to the same number of cameras to obtain the images required to compute a 3D reconstruction of a given object. In order to monitor the state of this hardware, Telegraf, an open source server agent, is used to collect the required data. Each required device or application pushes 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 requires a monitoring system at the edge, featuring minimal intrusiveness and very small bandwidth consumption. The level of interruption depends on the time interval of monitoring events, however, even with a small interval, this 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 by dividing the monitoring application into two parts. First, the client-side agent which is responsible for 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 being gathered, Zero Gravity 3D collects the following data for monitoring its state:

- Network bandwidth.
- Throughput, measured as parts per minute.
- CPU load and temperature.
- GPU load and temperature.
- RAM 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.

The Telegraf agent has been installed on all the sixteen edge boards of the 3D Industrial Inspection Case plus an external Raspberry Pi with internet access to send the collected data to FIVIS, the software that plays the role of the aforementioned storage server. To send this data, ITI has created a serializer plugin. The plugin is based on a Telegraf JSON serializer so that it will be consistent with the Telegraf architecture and provides the serialization of data into the JSON payload format required by FIVIS. In the Raspberry Pi, serialized data is sent to FIVIS through the Telegraf HTTP output plugin, using a POST method as required by CUNI's monitoring software.

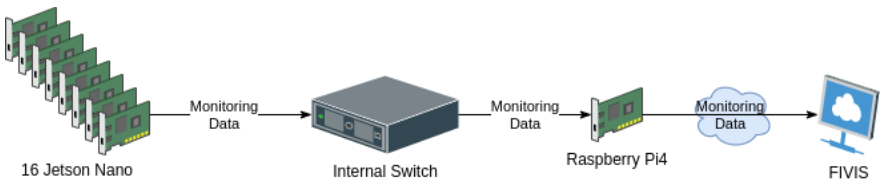


Figure 46 Monitoring infrastructure

Similar monitoring scenarios are being studied by other FitOptiVis use cases. The Telegraf agent and the serializer plugin can be installed on almost any device capable of running Linux and, in this way, easily provide the device with the capacity to send the information collected in 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 Telegraf plugin which can be easily installed on any device capable of running Linux. This solution provides a way to send data to FIVIS monitoring tool reducing the development effort and giving access to many data sources such as databases, network, CPU, memory usage and board sensors.

In order to display the stored data in a dashboard, FIVIS requires the creation of templates adapted to the needs of the monitored system. These templates make use of external parameters in JSON format, specifying properties such as type and label.

According to the specifications described in the previous point, a template to show current values of a capturer (an Nvidia Jetson card) was defined. Afterwards, this panel was replicated to all the 16 boards. Using the menu on the left, any of the devices can be selected to display its dashboard.

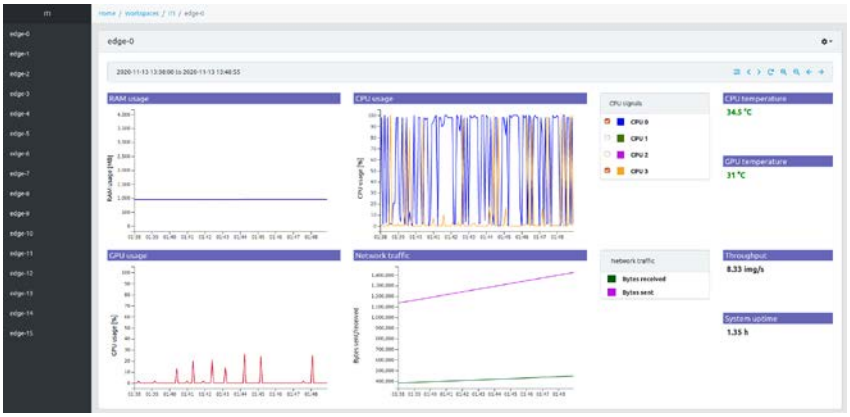


Figure 47 Fivis dashboard

5.2.2 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. 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.

Within the schedule of this report, device occupancy was implemented to a demonstratable level, network monitoring was implemented to a degree with a time out – based detection of network outage to switch between local and remote execution whenever network goes out of reach. Computer cluster probing is a work in progress, aiming for demonstration level before the end of the project.

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

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, PoCL was extended with a flexible tracing infrastructure, which allows creating data collection plugins for different types of output.

A simple text-output tracing plugin was created to demonstrate the 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).

In the final year of the FitOptiVis, two additional PoCL tracing plugins were created to feed PoCL tracing data to FIVIS servers. Both of these plugins use background OS threads to process and send the collected data, to minimize interference with the application.

The “influx“ plugin provides output in `influxdb line format`, a simple text format with one line per event. The output can be sent to either a file, or a TCP/UDP/UNIX connection. When using the connection-type output, the other side of the connection should be a Telegraf agent with a PoCL Json serializer plugin.

The “direct connection” plugin provides output in JSON format, suitable for directly uploading to FIVIS servers. The advantage of this plugin is that it does not require extra infrastructure (a running instance of telegraf with FIVIS plugin).

Data Storage, Analytics and Visualization

The data collection server now provides a graphical user interface for visualizing the data which will be utilized in the application optimization. FIVIS was extended with a specialized dashboard plugin to display tracing data collected by PoCL FIVIS tracing,

using swimlane visualization. It supports zooming, scrolling, manual time frame input, swimlanes of multiple devices, and visualizing event dependencies.

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) could 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.

5.2.3 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 frequency of the hardware components at any given time in order to find out what resources are being used. Each performance mode has a unique *id* and we measure the hardware status values every time it is changed.

- **Bandwidth usage:** we monitor the network incoming and outgoing bandwidth usage on this platform. We analyze it taking into account the network usage when sending critical (e.g. alarms) or best effort (e.g. video) data.

Habit Tracking use case (UC3)

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, monitoring the label confidence of the detected actions enable reconfigurations to reduce the power consumption and improve the recognition of critical activities. It is measured every time an inference over the video is performed.
- **Active operating mode:** The action recognizer runs on the node devices. Particularly, it can run using different operating modes. Each operating mode uses a different DL model to recognize the activities with a different accuracy vs energy consumption trade-off. The operating modes are identified with numbers from 0 to N, ordered from the least to the most accurate (models for testing on the datasets). Note that it is very important to identify which operating mode is active when performing reconfigurations. It is measured when reconfiguration commands are received.

Smart-Grid Surveillance system use case (UC9)

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.
- **Location of tracks**: In order to know whether a given intruder/worker is within a secure perimeter, its coordinate value (latitude, longitude) is transmitted.

Tasks assigned to each worker: In order to know whether a certain worker is assigned a specific task that allows him/her to stay in a given perimeter, the schedule of each worker is transmitted.

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 like:

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.

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.

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 Dataset	[OH11]	This surveillance video dataset is characterized by collecting data from natural scenes that showed people 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

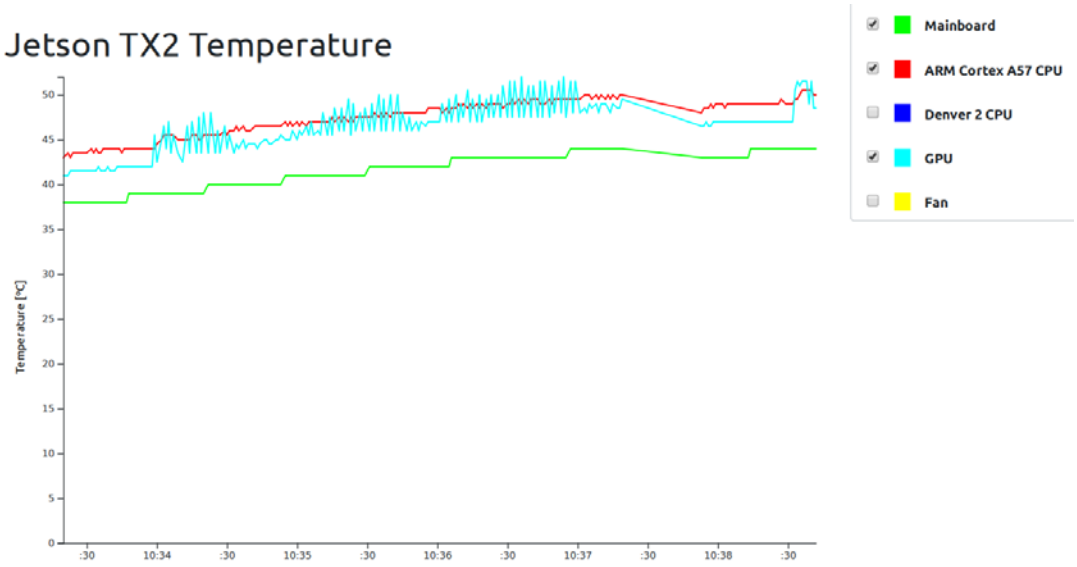
		synchronized surveillance video from 8 cameras at 1080p and 60 FPS, with over 2 million frames of 2,000 students walking to and from classes.

- **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.

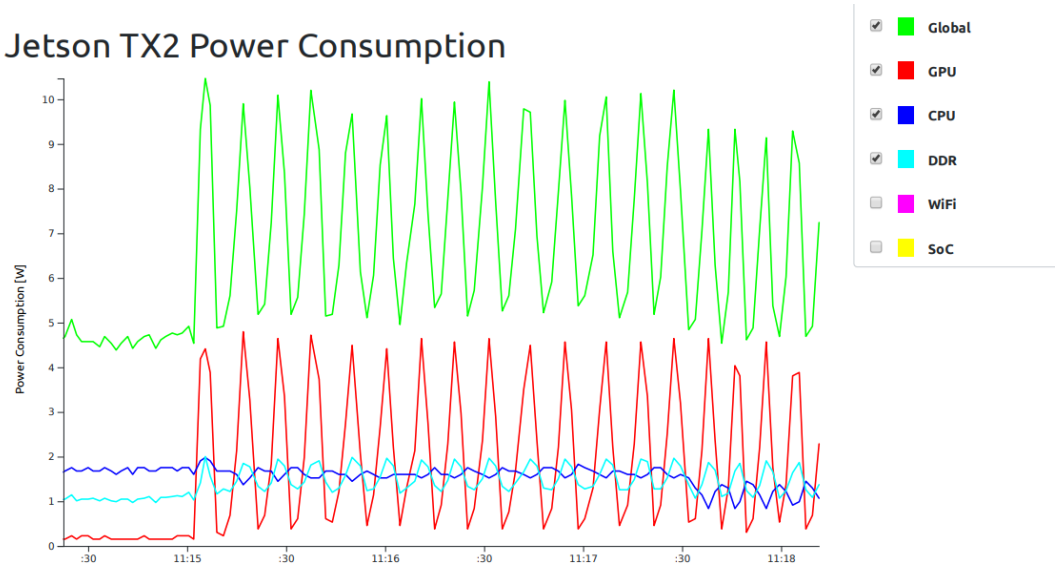
Data Storage, Analytics and Visualization

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:** Here we can see the temperature of the mainboard, the CPU and the GPU in Celsius during a concrete period of time.



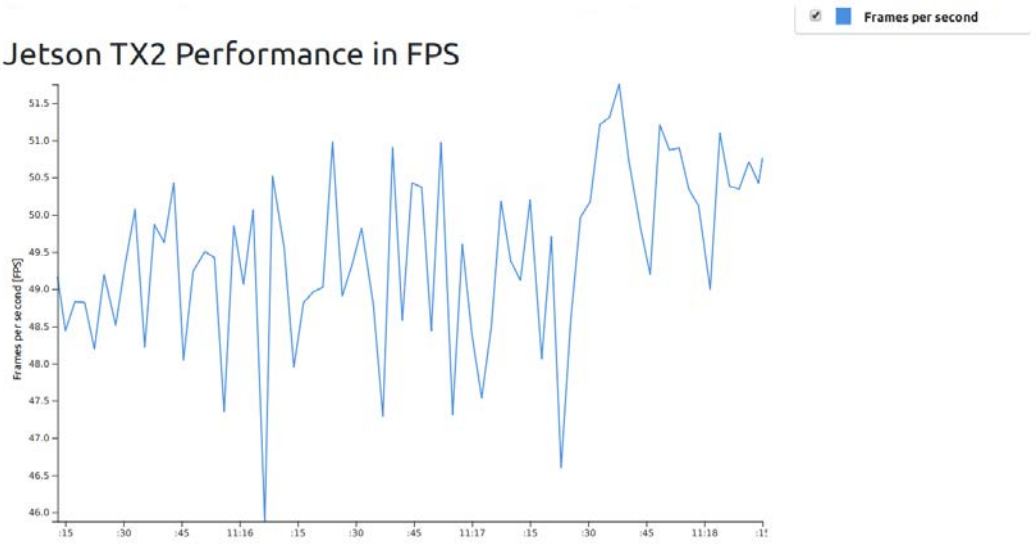
- **Power Consumption:** This visualization shows the power consumption at each second while our system is running.



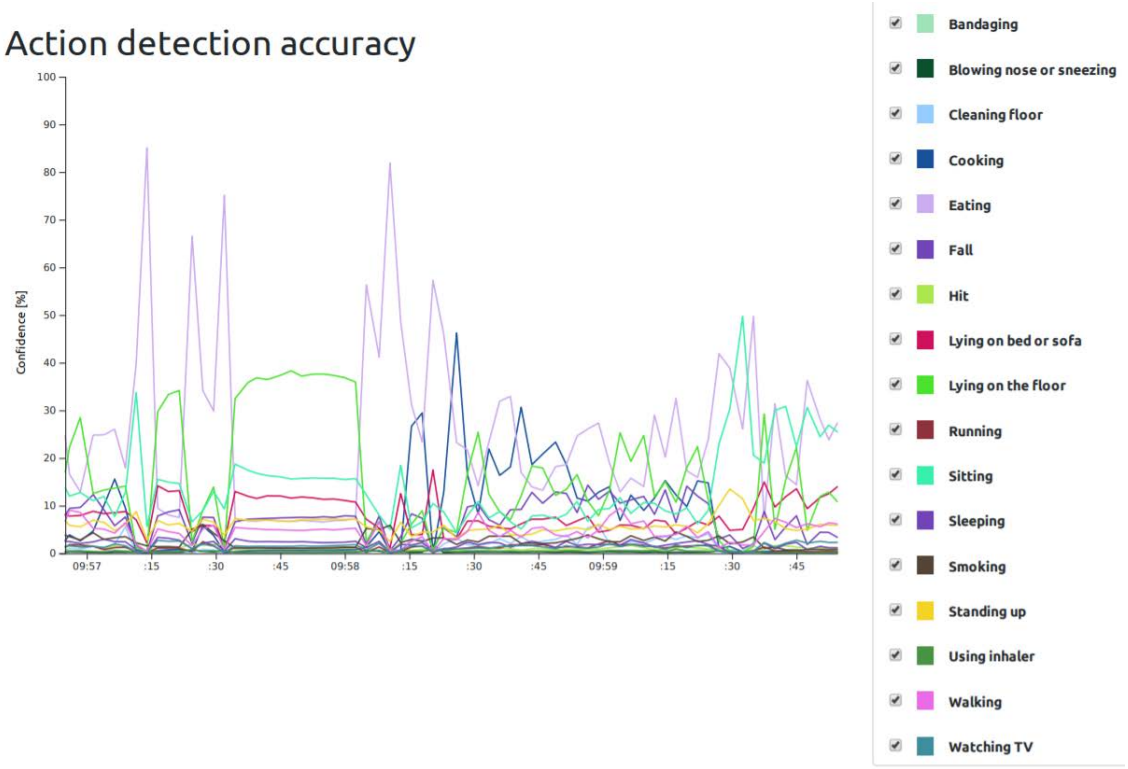
Habit Tracking

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.



- **Confidence of recognized actions:** In the next visualization, 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.

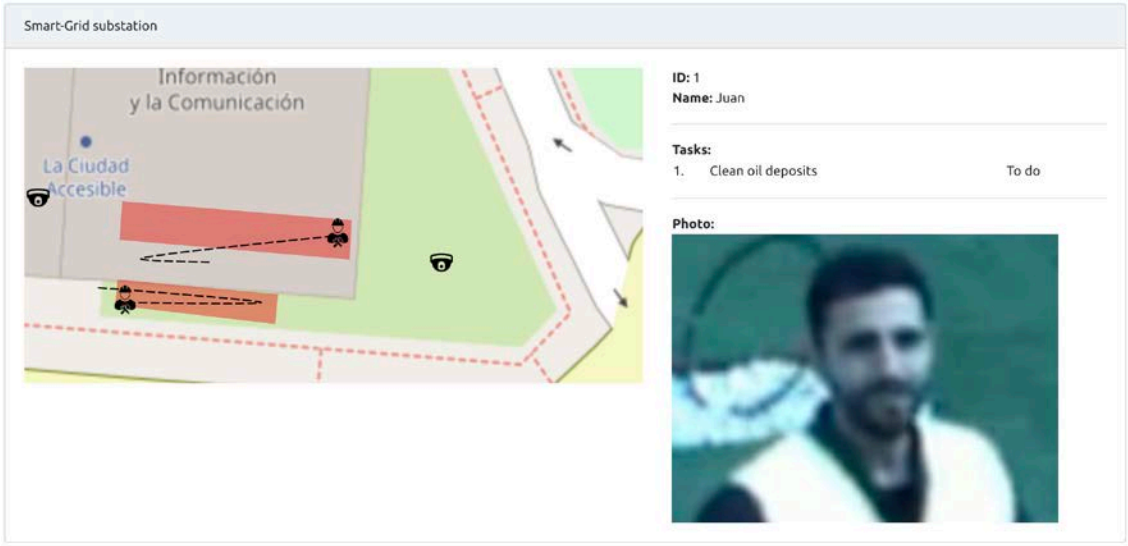


The same data can be visualized in different ways. In the following picture, we show a stacked variant of the above picture, which allows to quickly discern the dominating recognized action.

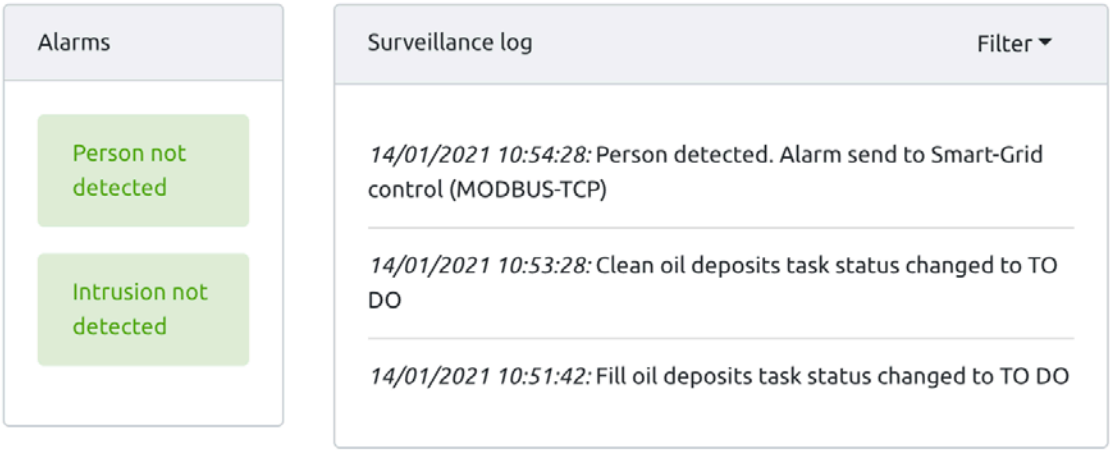
Smart Grid Surveillance

In the Smart Grid use case, we have metrics that are shown in the visualizations below.

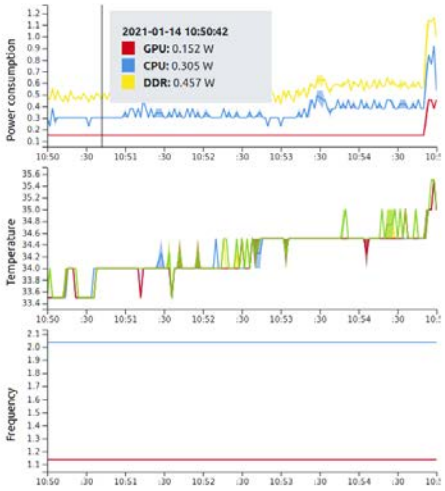
- **Location and identity of each track**: This visualization shows the real-time location and trajectory of the persons detected within the monitored infrastructure.



- **Alarm status and log information**: These visualizations represent the main events in the surveillance. On the one hand, the alarm panel shows the current status of the alarms. On the other hand, the information log includes the main events in terms of alarms triggering, installation perimeters exceeding, system reconfiguring, etc.



- **Camera node operation**: clicking on the camera elements provides information on the operational status of individual camera nodes, as shown below.



5.2.4 Monitoring capabilities for object recognition in space applications

In this section, the monitoring capabilities that will be used in the autonomous space exploration use case are described. The methodology provides two types of monitoring mechanisms:

1. Application quality parameter monitoring. This mechanism was described on section 5.1.2. The DSL component description defines component qualities. These qualities are assigned at runtime by the component code in order to provide information about component/system performance and status. A system component (runtime reconfiguration manager) uses these qualities for autonomous system configuration (resilience systems).
2. Platform resource monitoring. The platform-monitor components get information about resource usage in the hardware platform. Currently, Linux platforms are monitored with the libgtop library while NVIDIA platforms are monitored with NVIDIA tegra stats utility.

Monitoring results are managed using three different approaches:

- Tracing information is submitted to the FIVIS framework for data collection and visualization.
- Monitoring events are traced using the Linux LTTng framework.
- Monitoring qualities may be accessed or reported using the RIE infrastructure. Therefore, a system component (e.g., runtime manager) can access quality value in runtime.

Monitoring Requirements

Hardware platform parameter are monitored with the previously commented platform resource monitor. Specifically, the following parameters are monitored:

- Memory use: The memory size that is actually used by the application and the free memory size are reported.
- Available cores: Number of cores that are available to be used.
- CPU usage: For every core, percentage of use.
- Power consumption: this monitor reports the CPU power consumption.

Application components provide user-defined qualities that have to be monitored. They use the first approach (Internal quality parameter monitoring) and they are defined in the system DSL description. In the UC10 use case, the components use the following qualities:

- Frame rate: Video processing applications provide the frame rate to compare 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 necessary.
- 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 neural network or the trained weight in 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, therefore it has to be continuously traced.

Unit Under Monitoring

The space application is 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 applications that requires low power consumption. It includes and NVIDIA Maxwell family GPU, an ARM Cortex-A57 processor and 4 GB of memory.
- NVIDIA Jetson AGX Xavier: This board is equipped with a GPU Volta with specific Tensor cores, 32 GB of LPDDR4 memory and a memory storage of 32 GB. This board provides 20 times higher performance and 10 times higher energy efficiency compared to NVIDIA Jetson TX2.
- 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).

Next figure 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 48: 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 frames 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.

Monitoring data can be exported to FIVIS system in order to collect and visualize the data. FIVIS allows to represent data using several kinds of graphs, resulting in a more effective interpretation of the data rate.

5.2.5 Monitoring of 4x2 array of 8xSIMD Floating point Accelerators

In Y3, UTIA integrated 4x2 array of 8xSIMD floating point accelerators for larger Zynq Ultrascale+ device ZU15-EG and developed run-time support for parallel execution on these accelerators implemented as Debian OS POSIX threads.

Figure 49 presents complete system.

The 4x2 array of 8xSIMD, run-time reprogrammable, floating point HW accelerators is connected to ARM by zero-copy AXI-S HW data movers auto-generated by the SDSoC compiler. To reduce the count of needed AXI-S data movers to 8, the SIMD accelerators are arrange in the 4x2 array, with direct connections in HW. See Fig. 1. The SIMD accelerators perform floating point operations with 214 MHz clock. Accelerators are controlled from ARM SW by AXI-lite registers interfaced with 150 MHz clock.

This arrangement saves HW resources, but it also results in reduced data transfer performance in comparison to an alternative architecture with 8x1 SIMD accelerators. It also enables direct connectivity from accelerator 0 to 1, 2 to 3, 4 to 5 and 7 to 8. See Figure 49.

System contains Mult HW block generated by SDSoC 2018.2 compiler. It is designed to accelerate floating point (64x64) matrix by matrix multiplication. The accelerator is complemented by several alternatives of data movers (zero-copy or DMA or SGDMA) auto-generated by Xilinx SDSoC 2018.2 compiler.

The Full HD 60 FPS video input comes from video sensor to a chain of HW IPs converting it to 16 bit per pixel representation. Data are moved by Xilinx VDMA HW IP

core to frame buffers reserved in the DDR4 of the ARM A53 processor of the Zynq Ultrascale+ device. Video input comes via Avnet Imageon LPC FMC card. System works with 8 reserved video frame buffers accessible by the axi stream (AXI-S) VDMA data mover. Video output is using second AXI-S VDMA controller and generated output video frames from the same 8 video frame buffers.

The video image processing algorithm HW accelerator is compiled from Xilinx, OpenCV, HLS-compatible library of video processing algorithms. We work with two HW accelerated video pipeline algorithms:

- LK Dense Optical Flow algorithm as an example of large and complex video processing algorithm.
- Edge detection algorithm working with Sobel filter as an example algorithm.

Both algorithms use 214 MHz clock and operate on Full HD frames with performance above 60 FPS.

Monitoring Requirements

SW application running on ARM A53 processor has to control the parallel execution of the4x2 array of 8x SIMD HW accelerators. Each accelerator can possibly support different computing capabilities.

It is therefore required to provide mechanisms for the SW to identify in the run-time what are the capabilities of each HW accelerator currently present 4x2 array of accelerators in the programmable logic (PL) Zynq Ultrascale+ device.

Basing on this information, the processor SW application can decide how to program each individual HW accelerator in the 4x2 array and what internal 8xSIMD operations use in each accelerator.

Unit Under Monitoring

The unit under monitoring is one 8xSIMD run-time reconfigurable floating point HW accelerator present in 4x2 array of accelerators. See Figure 49.

If each of the accelerators in the 4x2 array 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. This information can be read by Arm SW application.

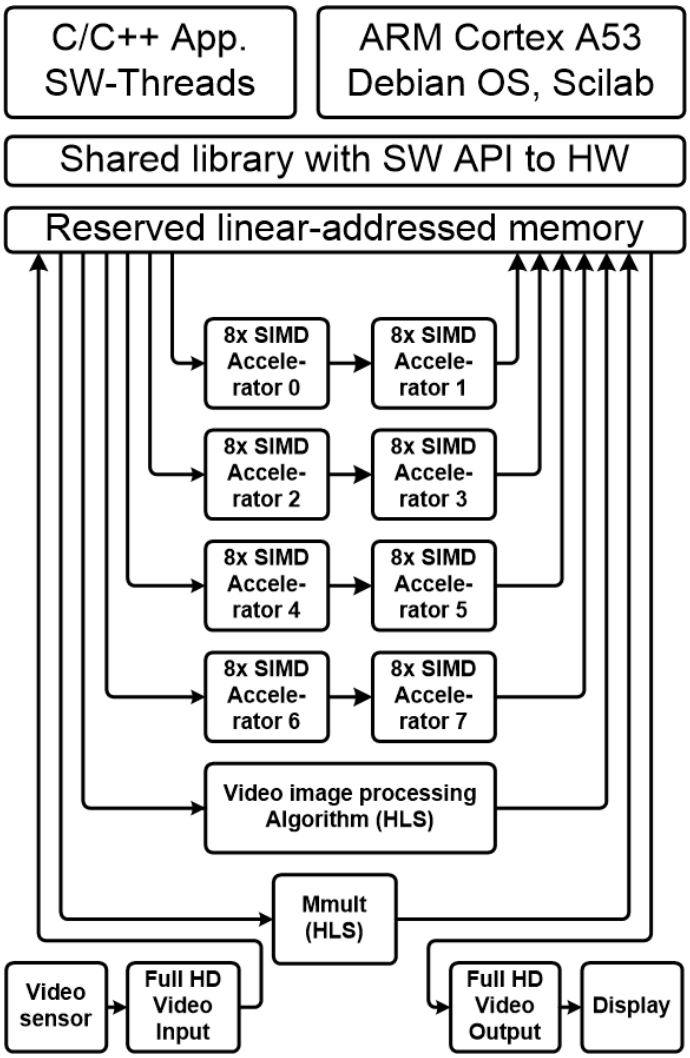


Figure 49 System with 4x2 array of 8xSIMD Floating point run-time reconfigurable accelerators

Implemented run-time infrastructure for parallel execution

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

- Firmware can be re-programmed at run-time using data streaming.
- Computation & data streaming can be performed in parallel.

Figure 50 documents the run-time execution loop and Figure 51 shows the code of one of the control threads.

```
while (!done) {

  //wait until parking frames are reached by VDMA HW engines
  reading_timeout = 0; writing_timeout = 0;
  while (readingFrame != XAxiVdma_CurrFrameStore(&vdma.inst, XAXIVDMA_READ)) {
        reading_timeout++;};
  while (writingFrame != XAxiVdma_CurrFrameStore(&vdma.inst, XAXIVDMA_WRITE)) {
        writing_timeout++;};
  if ((reading_timeout >= TIMEOUT_VAL) || (writing_timeout >= TIMEOUT_VAL)) {
        printf("VDMA parking - I/O error!!!\r\n"); break;}

// Advance frame buffer indexes
  advanceFStore(writingFrame);
  advanceFStore(nextInputFrame);
  advanceFStore(inputFrame);
  advanceFStore(outputFrame);
  advanceFStore(nextOutputFrame);
  advanceFStore(readingFrame)
  // Initiate parking to new positions (moved by one)
  XAxiVdma_StartParking(&vdma.inst, readingFrame, XAXIVDMA_READ);
  XAxiVdma_StartParking(&vdma.inst, writingFrame, XAXIVDMA_WRITE);

// Asynchronous call to start Sobel filter. Synchronise by: sds_wait(5);
  sobel_demo_processing((unsigned short*)vFrameStoreStartAddr[inputFrame],
                        (unsigned short*)vFrameStoreStartAddr[outputFrame],
                         numRows);
  // Demonstration of run-time change of processed microlines
  if (numRows == numRowsMax) { numRowsStep = -1;}
  if (numRows == numRowsMin) { numRowsStep =  1;}
  numRows += numRowsStep;
  // Sequence of matrix multiplications on 4x2 array of 8xSIMD accelerators
  // performed in parallel to HW accelerated video processing
  hw_sds_clk_start();
  pthread_create(&t,   NULL, hw_thrd,   &data);
  pthread_create(&t_1, NULL, hw_thrd_1, &data_1);
  pthread_create(&t_2, NULL, hw_thrd_2, &data_2);
  pthread_create(&t_3, NULL, hw_thrd_3, &data_3);
  pthread_join(t, NULL);
  pthread_join(t_1, NULL);
  pthread_join(t_2, NULL);
  pthread_join(t_3, NULL);

  // Measure time needed for the sequence of matrix multiplications
  hw_sds_clk_stop();

  // Wait here until the Sobel filter HW accelerator is done with currrent frame.
  sds_wait(5);
}
```

Figure 50. SW infrastructure for parallel execution of video pipeline and 4x2 array of accelerators

In Figure 50, the main application loop is controlling “round_robin” progress of actual frame buffer. The Full HD 60 FPS video data copied permanently from the video sensor to the video frame buffers by the VDMA HW controller.

Implemented SW infrastructure presented in Figure 50 selects the “actual” input frame buffer for input and actual output frame buffer for output video data by calls to function `StartParking()`.

The video processing function `sobel_demo_processing()` can be started at this stage, as the input and output video frame buffers are “parked” for SW access. The call is compiled for asynchronous. Function starts the Video processing HW accelerator for Sobel function based edge detection and returns immediately without waiting for processing of complete video frame.

Four threads controlling the 4x2 array of 8xSIMD accelerators are created and executed on four cores of the ZU15-EG Arm A53 device. Each thread controls two 8xSIMD accelerators organised in four rows, each row with 2 accelerators. See Figure 49.

The main SW loop waits until all four threads are joined and terminated. Time used for computation on 4x2 array of accelerators is measured.

Finally, the main loop calls the `sds_wait(5)` function, which blocks until the HW-accelerated Sobel video processing pipeline finishes. The Sobel accelerator first writes its computation results to the “parked” output video frame and then signals the `sds_wait(5)` function to return to the caller.

The `done` variable used in the top while loop is controlled by user Ctrl-C and serves for proper termination of SW. The VDMA HW data-movers are stopped in this case and performance results are reported. See Figure 51.

```
void *hw_thrd(void *ptr)
{
  int j;
  struct thread_args *data = (struct thread_args *)ptr;

  mat_mult_2s_stage_0(  data->fp03x8_0_inst, data->fp03x8_1_inst,
                        data->B_01, data->B_01_1);

  for (j = 0; j < MATRICES_IN_ONE_FRAME; j++){
    mat_mult_2s_stage_1(data->fp03x8_0_inst, data->fp03x8_1_inst,
                        data->B_02, data->B_02_1);
    mat_mult_2s_stage_2(data->fp03x8_0_inst, data->fp03x8_1_inst,
                        data->B_03, data->B_03_1);
    mat_mult_2s_stage_3(data->fp03x8_0_inst, data->fp03x8_1_inst,
                        data->B_04, data->B_04_1);
    mat_mult_2s_stage_4(data->fp03x8_0_inst, data->fp03x8_1_inst,
                        data->B_05, data->B_05_1);
    mat_mult_2s_stage_5(data->fp03x8_0_inst, data->fp03x8_1_inst,
                        data->B_06, data->B_06_1);
    mat_mult_2s_stage_6(data->fp03x8_0_inst, data->fp03x8_1_inst,
                        data->B_07, data->B_07_1);
    mat_mult_2s_stage_7(data->fp03x8_0_inst, data->fp03x8_1_inst,
                        data->B_08, data->B_08_1);
    mat_mult_2s_stage_8(data->fp03x8_0_inst, data->fp03x8_1_inst,
                        data->B_01, data->B_01_1);

    // Write matrix A to accelerator 0 and matrix A_1 to accelerator 1
```

```
    // Read matrix Z from accelerator 0 and matrix A_1 to accelerator 1
    mat_mult_write_a_read_z_2s(data->fp03x8_0_inst, data->fp03x8_1_inst,
                      data->A1_A2,   data->A3_A4,   data->A5_A6,   data->A7_A8,
                      data->A1_A2_1, data->A3_A4_1, data->A5_A6_1, data->A7_A8_1,
                      data->Z1_Z2,   data->Z3_Z4,   data->Z5_Z6,   data->Z7_Z8,
                      data->Z1_Z2_1, data->Z3_Z4_1, data->Z5_Z6_1, data->Z7_Z8_1);
    }
    return NULL;
}
```

Figure 51: One of the four threads controlling the parallel execution of 4x2 array of accelerators

The code of one of the four threads is shown in Figure 51. It performs a sequence of two floating-point matrix multiplications on two connected 8xSIMD HW accelerators `data->fp03x8_0_inst`, `data->fp03x8_1_inst`. These two instances correspond to 8xSIMD HW accelerators with index 0 and 1 in Figure 51.

Matrix B is loaded to accelerator 0 and matrix B_1 to accelerator 1 as 8 matrix slices in parallel to the actual floating-point computation on both accelerators. This is done by eight sequential calls to `mat_mult_2s_stage_1()` ... `mat_mult_2s_stage_8()` functions.

This masked write of 8 slices of B and B_1 is shorter than the actually performed floating point computation in accelerator 0 and accelerator 1.

Finally, the thread calls the `mat_mult_write_a_read_z_2s()` function. It downloads the result of the two matrix multiplications to the host memory (as matrices Z and Z1) and (in parallel) uploads a pair of new matrices A and A_1 from the host memory to the accelerators 0 and 1 for next computation in the loop performed by the thread.

Performance results for sequence of floating point matrix multiplications

The array of 4x2 8xSIMD on ZU15-EG, 214 MHz clock provides: **10.82** GFLOPs
The SDSoC 2018.2 HW on ZU15-EG, 214 MHz clock provides: **7.31** GFLOPs
The SW version 4 threads, ZU15-EG, 1,05 GHz clock provides: **0.64** GFLOPs

The run-time infrastructure for parallel execution of the 4x2 array of run-time reprogrammable 8xSIMD accelerators developed in Y3 of FitOptiVis outperforms (in case of computation of a sequence of floating point matrix multiplications) the HW accelerator generated for same task by the Xilinx SDSoC 2018.2 compiler.

Measured GFLOPs performance of the array of 4x2 8xSIMD accelerators, SDSoC accelerator and SW implementation of matrix multiplication running in 4 threads on Arm A53. Both HW accelerator based implementations use 64x ADD and 64x MULT HW floating-point, pipelined units.

Both compared solutions can be executed in parallel to the edge detection video processing by Sobel filter HW accelerator. The video pipeline works in Full HD, 60 FPS with HDMI video input and video output.

5.2.6 Monitoring of Distributed Execution in the Virtual Reality Use Case

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.

The real-time augmented reality rendering demo utilizes two remote OpenCL devices, one ("streaming custom device") provides the V-PCC video stream, while the other (a GPU, in this case Nvidia 1060) is used to improve the quality of the reconstructed point cloud. Execution time stamp data is sent to the FIVIS data collection server using PoCL-based Telegraf plugin. The visualization of the data in a device-swim-lane format is shown in Figure 52. Framerates measured from the application in its different execution configurations are shown in Figure 53, and the energy per frame in Figure 54. The reconfiguration modes and the overall effects are discussed in D4.5.

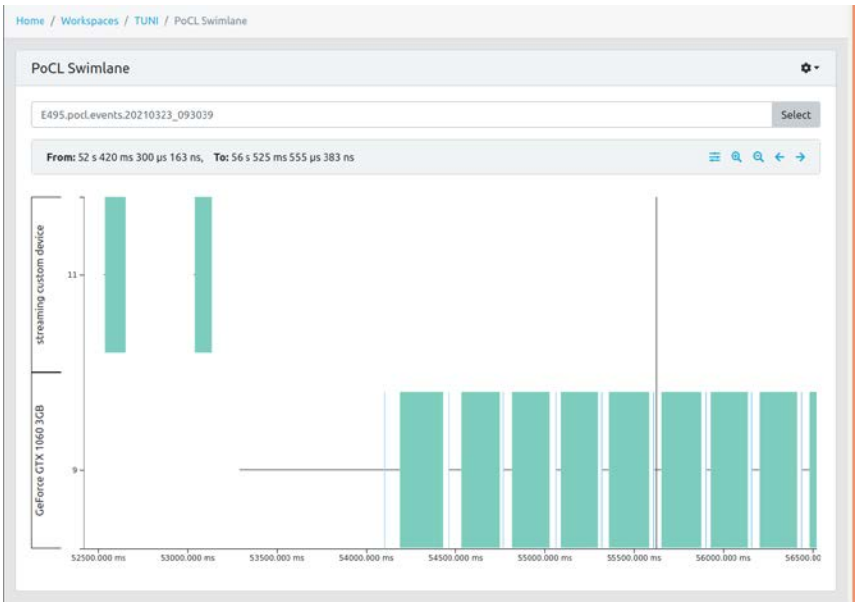


Figure 52: The swimlane visualization of monitoring data gathered during local+remote configuration execution of the AR demo.

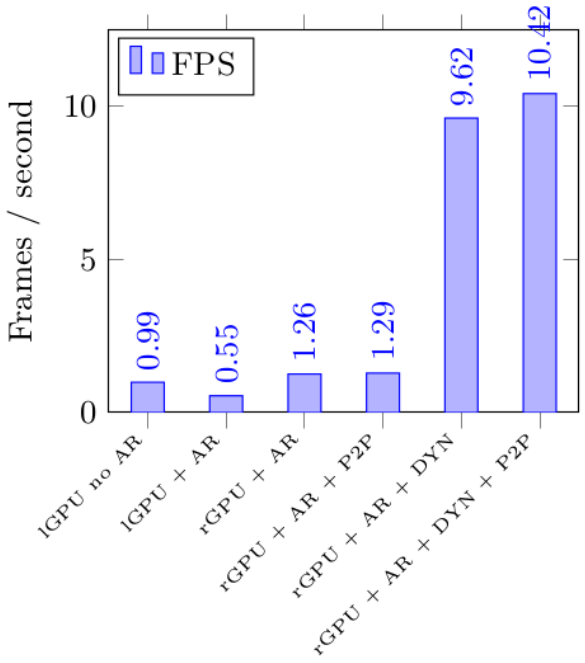


Figure 53: Monitoring results to the FPS in the different remote/local execution configurations in the AR demo.

The first two measurements are obtained using the local (mobile) GPU without AR (model only) and with AR. The next four measurements offload point sorting to a GPU on a PoCL-R remote server, with various PoCL features enabled.

The power usage of the smartphone was retrieved using Android's Power Stats HAL interface. Offloading the sorting of the point cloud compensates for most of the added energy consumption from AR positioning even without further optimizations.

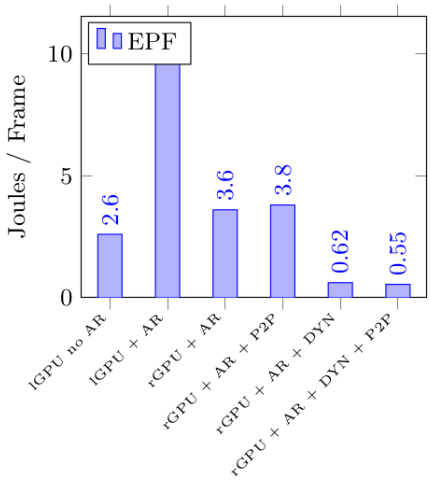


Figure 54: Monitoring results of energy consumption per frame in the different remote/local configurations of the AR demonstrator.

Figure 55: Screenshot of the AR demonstrator with on screen visualization of the monitoring data during **remote** execution.

Figure 56: Screenshot of the AR demonstrator with on screen visualization of the monitoring data during local execution.

5.2.7 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, eye-tracking

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 II AR-glasses to ensure that all possible use cases can be implemented easily. We have also investigated 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, but we concluded that it is best to utilize only HoloLens II AR-glasses in the FitOptiVis project for optimal efficiency and usability reasons.
- 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 performed only initial measurements on power usage and based on the achieved results, we have made 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 fulfil 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 was developed using C# programming language on top of the Unity 3D engine and the server back-end side was developed using PHP. All description packets are in JSON format.

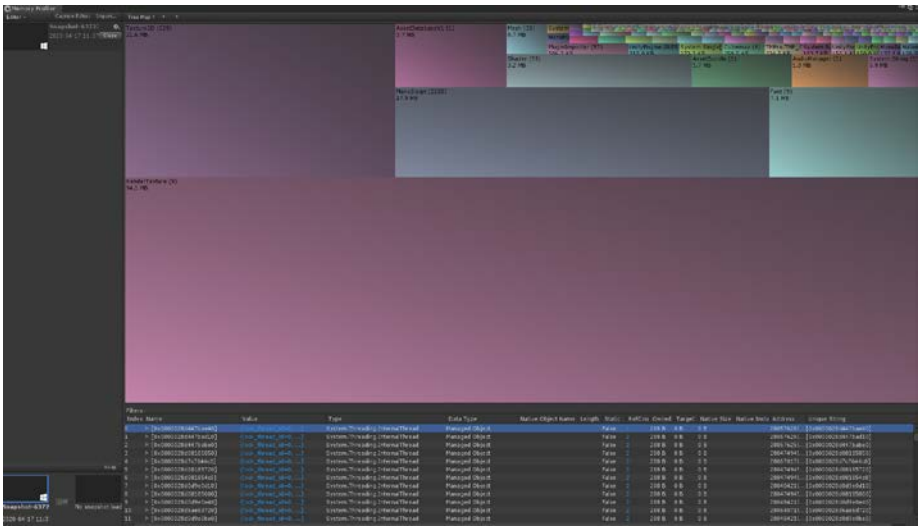
Monitoring Infrastructure

As of M35, 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 was 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 was monitored by utilizing the real-time development platform Unity's own tools. We were 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:



Data Storage, Analytics, and Visualization

Collected data will be stored in secure servers. Analytics and visualization was done by utilizing appropriate analytics/visualization tools (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 was used to keep track and compare how the user has progressed in the rehabilitation.

5.2.8 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.

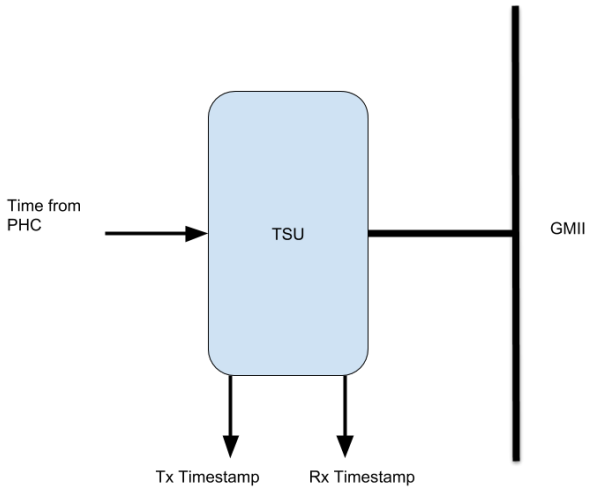


Figure 57: Timestamping Unit

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 depicted below.

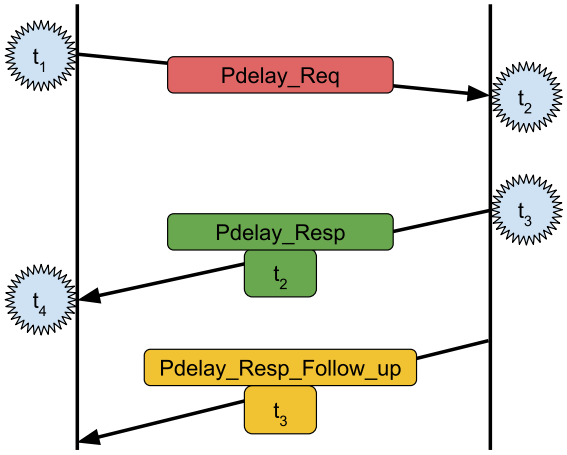


Figure 58: Regular Handshake on the Peer Delay Mechanism

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. 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.

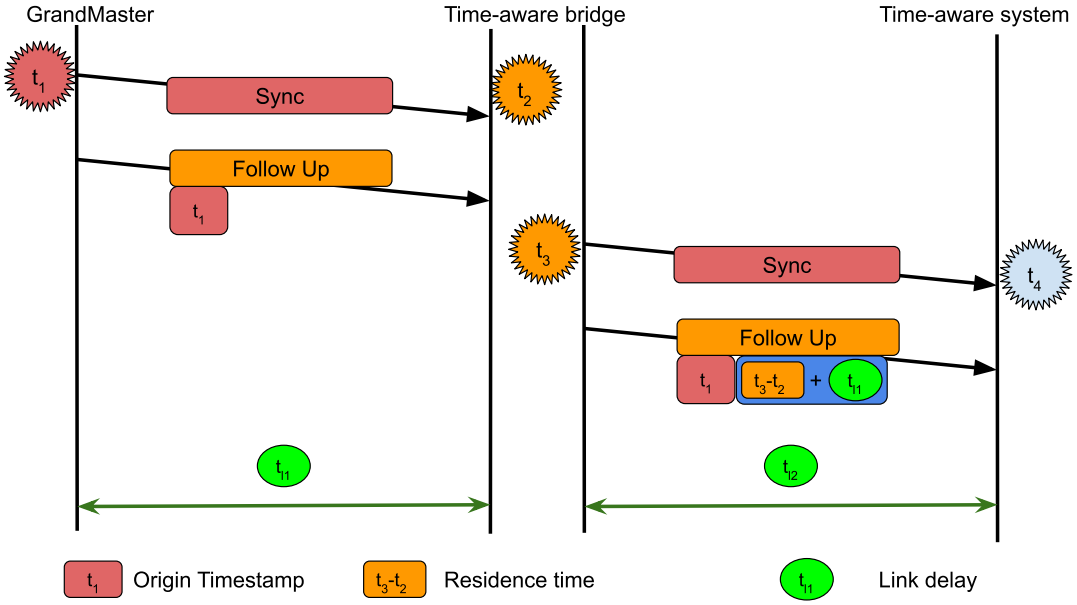


Figure 59: Two-step PTP mechanism, as defined on IEEE 802.1AS

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 in FIVIS and available for presentation using a custom panel, as shown in Figure 60. The panel can be embedded into any use-case-specific dashboard which needs display status of TSN nodes.

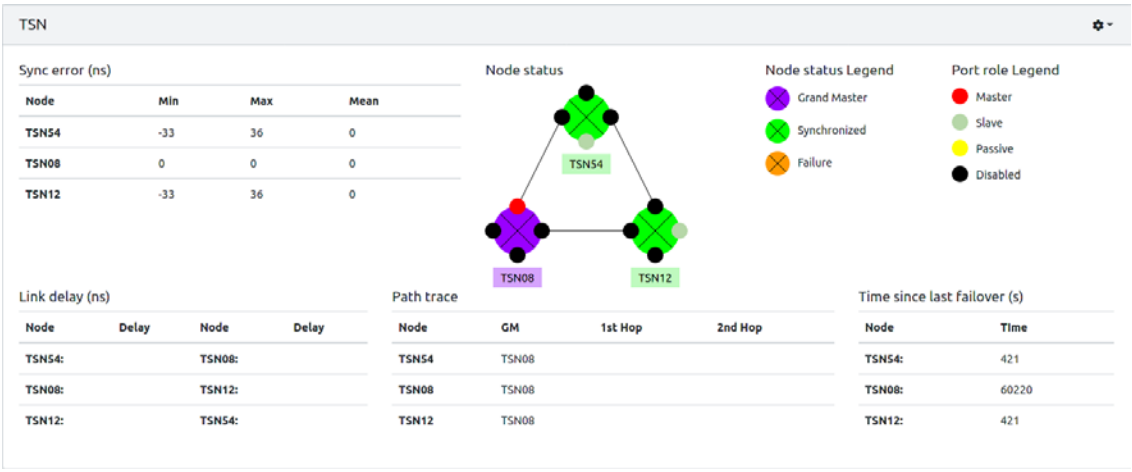


Figure 60. Dashboard showing status of TSN nodes in FIVIS

5.2.9 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 three main monitors to report at runtime in order to control the reconfiguration mechanisms will be:

- Radiation dose: off-the-shelf components 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.
- Image processing monitors: In the frame of the FitOptiVis project, two main components regarding the processing of the image have been developed by

TASE for UC10: The Space Image Processing chain and the Image Collection Interface. Both, combined, can obtain an image from a CMOS sensor (which are very new in the space industry) and send it to the components developed by the University of Cantabria.

- Monitors to study the quality of the image are currently being used. The goal of this monitors is to analyse how the components behave under the different setpoints of the system which are closely related with the possible stages of the mission of the spacecraft. The monitors for the image processing of the system are: fps, image resolution, image size.

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 the following 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
- The components developed in WP5 (Space Image Processing and Image Collection Interface) which are in charge of monitoring the image qualities.
- A host machine that gathers all the data coming from the previous blocks. This machine is in charge of putting together all the monitors together and storing them. Once that the monitors are correctly stored, the infrastructure described by University of Cantabria (UC) in Section 5.2.4 is in charge of receiving them and displaying them with FIVIS.

Data Storage, Analytics and Visualization

As described in the previous point, a host machine is in charge of receiving all the monitors coming from the different components and to store them into a text file. This machine is accessed by the infrastructure developed by UC and described in Section 5.2.4 in order to visualize the data with FIVIS

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

As part of the effort in Habit Tracking use case, HIB has developed 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. The edge device is more suited to the facial recognition while the cloud recognition engine is more suited for pose estimation and activity recognition using a LSTM network working on the joint position computed.

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 PoseNet[1] for TensorFlow which yields a wire-frame model of persons in still frames. This has been changed from the original openpose[2] component presented in D4.4. The module is functionally identical but the underlying neural network is different.
- **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

The following Figure represents the overall architecture of the system under analysis:

[1] https://www.tensorflow.org/lite/examples/pose_estimation/overview - PoseNet body wireframe deep learning system.
[2] https://github.com/CMU-Perceptual-Computing-Lab/openpose - openpose: multiperson human body posture detection by Carnegie Mellon University.

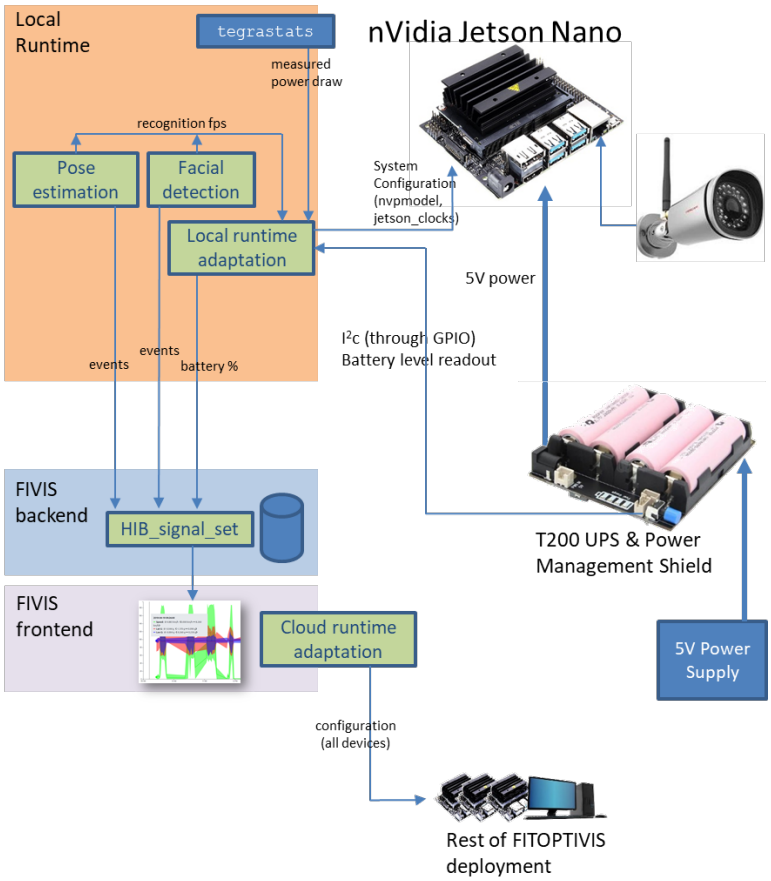


Figure 61 HIB architecture for UC3 Habit Tracking

The Figure 61 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 i^2C[3], 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 61. In addition to the aforementioned hardware units (the Nvidia Jetson Nano and the GeekwormT200 UPS kit with i^2C battery

[3] 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 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 and a more powerful GPU based activity detection system) can be switched on and off on demand.

5.2.11 Monitoring of high-performance embedded applications for Water-Supply maintenance

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 coprocessors systems, offering a coarse-grained functional and non-functional reconfiguration (Section 4.3). 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 JOINTER framework (Section 5.1.3). The framework to use JOINTER applied to MDC generated coprocessors can be accessed in the repository at the following link: https://github.com/alkalir/jointer. The final monitoring system has been applied to monitor the edge-computing platform associated to Water-Supply Use-case (UC1).

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 62 shows the IP generated by MDC.

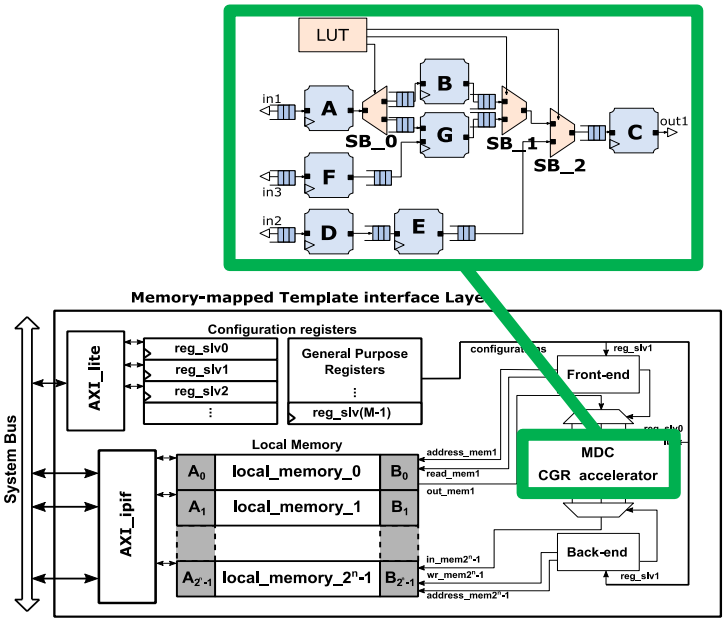


Figure 62: The figure shows the IP generated by MDC. The MDC CGR accelerator is a coarse grain reconfigurable datapath capable of executing different functionalities, by opportunely multiplexing resources in time. This datapath is automatically encapsulated into a ready to use Xilinx IP and into a processor-coprocessor system, according to the user choices during the design time.

Monitoring Infrastructure

The monitoring infrastructure has been generated using the JOINTER framework: this allows to satisfy the MON1 requirement, since JOINTER produces hardware monitoring systems that limit the timing impact on SW execution. Three different sniffers can be selected to monitor the coprocessor. Figure 63 shows the sniffers and their placement with respect to the internal circuitry 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, thus satisfying MON4.

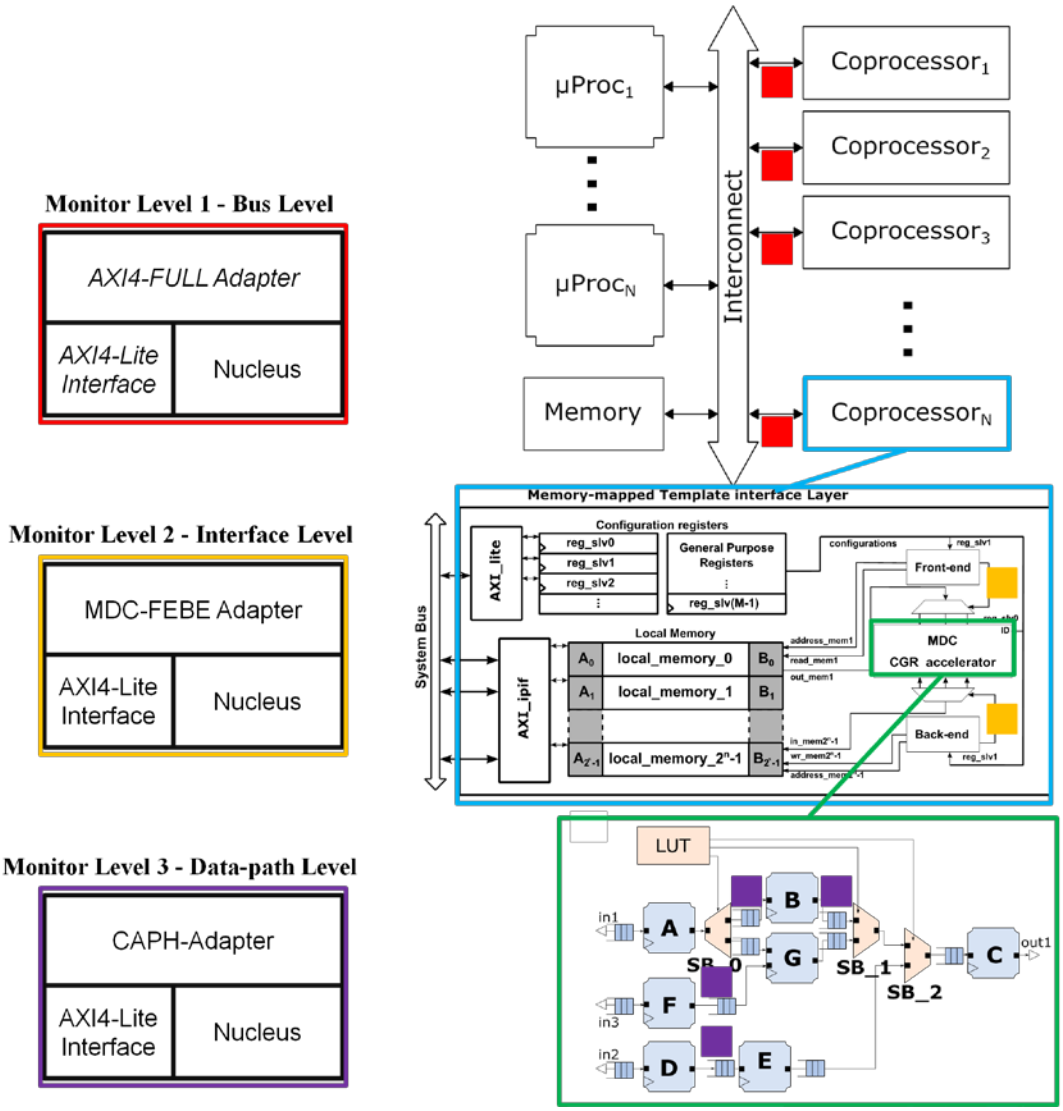


Figure 63: Sniffer generation for MDC coprocessors

Data Storage, Analytics and Visualization

Data output by the monitoring system are organized as reported in Table 2.

Table 6: Event instances of monitors. EVENT_ATTRIBUTE contains an attribute for internal usage, ACC_ID provides a code to indicate the ID of the monitored coprocessor, LEVEL_ID indicates the monitored level, EVENT_INFORMATION contains the raw information.

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 JOINTER for MDC and the application of JOINTER to generated monitoring systems for the Water-Supply Use-case (UC1), are reported in [VAL21][VAL20][MUT20] and in the project repository: https://github.com/alkalir/jointer.

6. **Conclusions**

In our summary of the final outcomes of Task 4.2 and Task 4.3 from the last year of the project, we reported specific developments of the technologies and methods reported in D4.3 and certain new technologies and methods developed over the last year to provide runtime support for adaptive applications. Many use-case specific results are reported in the direction of monitoring and profiling as well as runtime reconfiguration.

In line with D4.3, D4.4 reports specific extension and use-case-specific adaptation and results of various runtime reconfiguration mechanisms under three main categories of reconfigurations related to adding/removing components, changing the component configuration and composition. We reported seven reconfiguration mechanisms developed and extended over the reporting period targeting platforms and applications ranging from hard real-time (CompSOC, HW accelerator) to industrial platforms (NVIDIA Jetson and AGX), and to TSN-based distributed systems to edge-cloud systems. They are further evaluated, validated, and verified in various use case scenarios. In turn, these mechanisms enable deployment of the FitOptiVis QRM framework in the context of various use cases.

The reconfiguration process is activated using runtime monitoring, profiling and measurement mechanisms spanning over different levels. In line with D4.3, D4.4 reports different enabling technologies and instances of monitoring mechanisms under the FitOptiVis reference platform. In particular, three enabling technologies have been proposed withing FitOptiVis, FIVIS, QRML extension, and JOINTER, each one associated with various dissemination activities. The final obtained instances made use of these enabling technologies, and spanned at different levels, from cloud to edge, with solutions requiring the synergy between hardware monitoring systems and software ones. This was the key to properly control the intrusiveness of monitoring systems, and also to properly find the architectural trade-offs.

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.

[KEP03] Kephart, J., Chess, D.: The Vision of Autonomic Computing. Computer. 36, 1, 41–50 (2003).

[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.

[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.

[VAL21] G. Valente, T. Fanni, C. Sau, T. Di Mascio, L. Pomante, F. Palumbo, (accepted, in press). A composable monitoring system for heterogeneous embedded platforms. ACM Transactions on Embedded Computing Systems

[VAL20] G. Valente, T. Fanni, C. Sau and F. Di Battista, "Layering the monitoring action for improved flexibility and overhead control: work-in-progress," 2020 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Singapore, 2020, pp. 18-20, doi: 10.1109/CODESISSS51650.2020.9244018.

[MUT20] V. Muttillo, G. Valente, L. Pomante, H. Posadas, J. Merino and E. Villar, "Run-time Monitoring and Trace Analysis Methodology for Component-based Embedded Systems Design Flow," 2020 23rd Euromicro Conference on Digital System Design (DSD), Kranj, Slovenia, 2020, pp. 117-125, doi: 10.1109/DSD51259.2020.00029.

[VAL2_21] G. Valente, F. Caruso, L. Pomante, T. Di Mascio, “Monica - On-chip Monitoring Systems Characterization”, 2021 Design Automation and Test Conference (DAC 2021) (to be presented in December 2021).

[DE21] Sayandip De, Yingkai Huang, Sajid Mohamed, Dip Goswami, Henk Corporaal, “Hardware-and Situation-Aware Sensing for Robust Closed-Loop Control Systems”, 2021 Design Automation and Test of Europe Conference (DATE 2021)

[BERG20] F. van den Berg, V. Čamra, M. Hendriks, M. Geilen, P. Hnetynka, F. Manteca, P.P. Sánchez, T. Bureš, T. Basten. QRML: A Component Language and Toolset for Quality and Resource Management. In Forum on specification & Design Languages, FDL 2020, Proceedings, 8 pages. Kiel, Germany, 15-17 September, 2020. IEEE Computer Society Press, Los Alamitos, CA, USA, 2020.

[HENDRIKS20] M. Hendriks, M. Geilen, K. Goossens, R. de Jong, T. Basten. Interface Modeling for Quality and Resource Management. arXiv:2002.08181 [cs.LO], 19 February 2020.

[GEILEN07] M.C.W. Geilen, T. Basten. A Calculator for Pareto Points. In R. Lauwereins, J. Madsen, editors, Design, Automation and Test in Europe, DATE 2007, Proceedings, pages 285-291. Nice, France, 16-20 April, 2007. IEEE Computer Society Press, Los Alamitos, CA, USA, 2007.

[BERG20_1] F. van den Berg, V. Čamra, M. Hendriks, M. Geilen, P. Hnetynka, F. Manteca, P.P. Sánchez, T. Bureš, T. Basten. QRML: A Component Language and Toolset for Quality and Resource Management. In Forum on specification & Design Languages, FDL 2020, Proceedings, 8 pages. Kiel, Germany, 15-17 September, 2020. IEEE Computer Society Press, Los Alamitos, CA, USA, 2020.