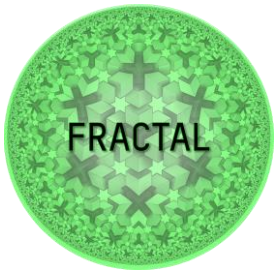

D4.2 FRACTAL low-power services

Deliverable Id:	**D4.2**
Deliverable name:	**FRACTAL low-power services**
Status:	**Draft**
Dissemination level:	**PUBLIC**
Due date of deliverable:	**2022-10-31 (M26)**
Actual submission date:	**2022-10-25**
Work package:	**WP4 "Safety, Security & Low Power Techniques"**
Organization name of lead contractor for this deliverable:	**ROT**
Authors:	Nadia Caterina Zullo Lasala, ROT Damiano Vallocchia, ROT Amal Alrish, ROT Daniel Onwuchekwa, SIEG Pascal Muoka, SIEG Juan Garcia, QUA Ankur Raj, QUA Berkay Enginoglu, PLC2 Alexander Flick, PLC2 Luca Bertaccini, ETHZ Leticia Pascual, SML Raúl García, SML
Reviewers:	Pietro Abbatangelo, MODIS Frank K. Gürkaynak, ETHZ

Abstract:
The goal of WP4 is to develop safety, security, and low-power techniques for individual FRACTAL nodes. In T4.1 we investigated low-power services in seven components implemented in the FRACTAL project, and this document will report the results of this work, extending the preliminary implementations reported in D4.1. The development includes both the node level (i.e., individual FRACTAL nodes) as well as the system level (i.e., distributed systems comprised of FRACTAL nodes) in accordance with the FRACTAL system architecture.

This project has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 877056

Co-funded by the Horizon 2020 Programme of the European Union under grant agreement No 877056.

Content

1 History

Version	Date	Modification reason	Modified by
0.1	2022-05-31	First contributions	All partners
0.2	2022-07-15	Refine contributions and outline tests	All partners
1.0	2022-07-20	Draft version	All partners
2.0	2022-08-01	New organization of the contents of the document	All partners
3.0	2022-09-06	Major contributions to all chapters	All partners
4.0	2022-09-20	Addressing HLAB feedbacks, final contributions	All partners
5.0	2022-10-07	Pre-final version of the document ready for internal review	ROT
Final	2022-10-24	Final version of the document ready for submission	ROT

2 Summary

This deliverable aims to report the outcomes of T4.1 on Low-Power services. The results of the implementations carried out in the task are presented according to the components developed, which reflect the objectives of the task.

Namely, the work in T4.1 was focused on suitable Data Compression techniques for low-power services, specifically the LZW compression techniques, described in Section 5 (ROT). Moreover, this Task developed a hierarchical architecture to facilitate low-power services for FRACTAL systems with a time-triggered Network-on-Chip, with the work on the Hierarchical Adaptive Time-triggered Multi-core Architecture (HATMA), which will be described in Section 6 (SIEG). In addition, energy measurement and energy-efficient computing for low-power services for PULP systems have been explored in Section 7 (ETH). The task also investigated low-power communication protocols for a wireless network, in the Agreement Protocol in Chapter 9 (QUA). Furthermore, RPU based access for Power Services on safety focused Versal platforms have been explored in Sections 8 and 10 (PLC2) respectively. Finally, following the outcomes of WP3, T4.1 validated the adaptation of the LEDEL Library for low-power services in Section 11 (SML).

3 Introduction

The goal of the 4th work package is to develop safety, security, and low-power services for individual FRACTAL nodes. In this document, we will shed light upon low-power services for FRACTAL systems extending their preliminary implementations reported in the deliverable D4.1. The development includes both the node level (i.e., individual FRACTAL nodes) as well as the system level (i.e., distributed systems comprised of FRACTAL nodes) in accordance with the Fractal system architecture depicted in Figure 1. To this aim, we will provide low-power services for multi-core chips that use a time-triggered Network-On-Chip (NoC) to interconnect heterogeneous types of computing resources such as general-purpose processor cores. Moreover, we are developing low-power services for the interconnected FRACTAL nodes based on a hierarchical system concept with wire-bound and wireless time-triggered off-chip networks. Furthermore, we will extend the low-power services realized on FRACTAL nodes to the system level by investigating low-power communication protocols for wireless and wired networks. In addition, we will investigate new aggregation and compression algorithms to reduce the amount of time and data needed to transmit information over a channel. Finally, the validation process of LEDEL, the library to implement Machine Learning algorithms as a low-power service, is detailed in accordance with the results of WP3.

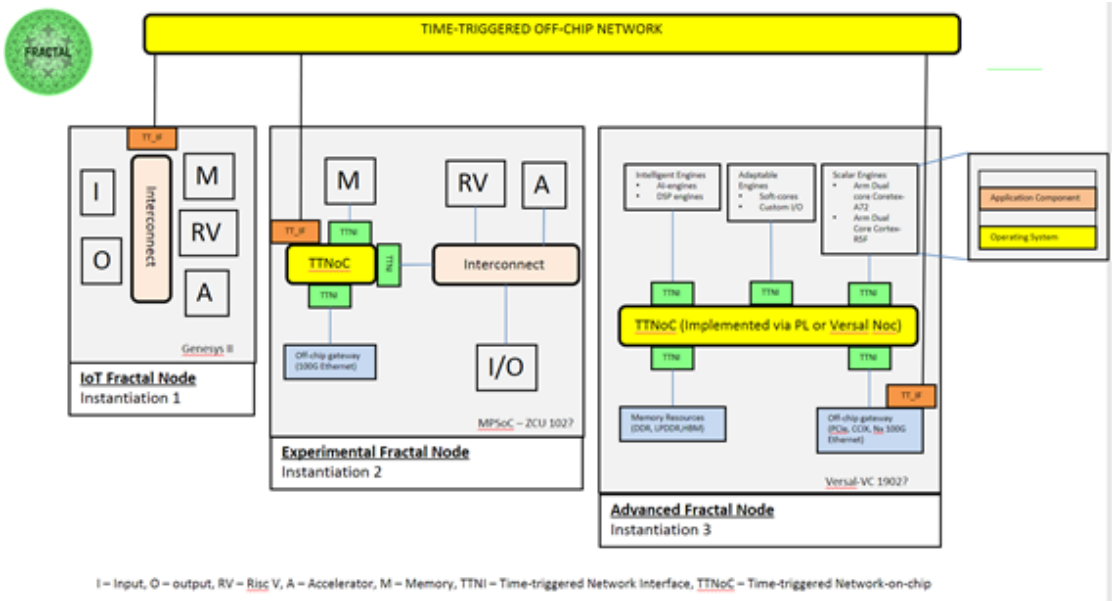

Figure 1 Fractal system architecture

The strategic objective of this task is to guarantee the energy efficiency of the FRACTAL system. To achieve this objective, the task was realized by several building blocks/components which contribute to fulfill the T4.1 objectives and are reusable and could be demodulated by any use case (UC).

A brief description of each of the T4.1 components and how they contribute in achieving the Task objectives are reported in Table 1.

Table 1 Brief description of the components and contribution T4.1 objectives

Data Compression for low-Power Services - WP4T41-01	
Description	It is a data compression technique for low-power devices, applied at the system level. In particular, it is a software library developed in C++ which performs operations of data compression and decompression making use of the universal lossless data compression algorithm LZW.
Contribution to achieving the T4.1 objectives	This component satisfies the need to investigate compression techniques for energy-efficient data compression to reduce the amount of data transfer for low-power services. This data compression component contributes to improving nodes utilization, increasing network lifetime, and improving bandwidth efficiency.
HATMA - WP4T41-02	
Description	It is a Hierarchical Adaptive Time-triggered Multi-core Architecture used to facilitate services at the different hierarchies. It delivers adaptation services in a distributed system of multiple nodes and within the nodes.
Contribution to achieving the T4.1 objectives	This component facilitates adaptability features of the FRACTAL node through aligned switching of time-triggered schedules in response to context events. HATMA delivers low-power services by leveraging context events such as dynamic slack arising from the execution of application tasks. Techniques such as Dynamic Voltage and Frequency Scaling (DVFS) and clock and power gating are used to reduce the node's power consumption or efficiently utilize the available power during periods of dynamic slack.
Low Power services for PULP systems - WP4T41-03	
Description	It aims to enhance existing platforms to increase energy efficiency and provide low-power FRACTAL services. Specifically, microarchitectural modifications and extensions to the RISC-V platform are explored to specialize the architecture for a specific application scenario, thus increasing its energy efficiency. Additional components can be placed in different power domains, which can be powered down during the sleep mode of the IoT device. Different power gating granularities are investigated.

Contribution to achieving the T4.1 objectives	The component satisfies the objective of the task by exploring fine-grained power gating and increasing the energy-efficiency of the platform through microarchitectural modifications and specialized architectures
Versal RPU access for Power Services - WP4T41-04	
Description	The component enhances the safety-centric Versal platform to provide access to power control and monitoring services through underlying HW accesses as provided by Versal Isolation Design. This service proxy code is running on the real time processing units (RPU) in the Versal processing system. The mission mode (FRACTAL node) services use this component along the provided protocol to retrieve a predefined set of control and monitoring features.
Contribution to achieving the T4.1 objectives	This component offers power control and status monitoring for the Versal FRACTAL node by setting the power domains, Dynamic Voltage and Frequency Scaling (DVFS), and clock gating. This provides the Versal-based low-power services and so enables the physical node adaptivity for the HATMA services.
Agreement Protocol for Low-Power Services - WP4T41-05	
Description	It is an implementation of the agreement protocol on a wireless network on low-power devices. This protocol aims at having clock synchronization among all the devices connected in the network.
Contribution to achieving the T4.1 objectives	The component fulfills the T4.1 objective of investigation for the low-powered wireless communication protocol, it helps in providing a method for time-based synchronization on low-powered devices. The synchronization is based on a PTP-like method with low-powered microcontrollers sending synchronization messages wirelessly. The implementation was achieved using ESP-32 microcontrollers and a communication protocol known as ESP-NOW.
Versal Isolation Design- Functional Safety - WP4T41-06	
Description	It aims to enhance the common Versal platform to strictly separate functional accesses and services from underlying HW access. In particular, in safety augmented designs on the Versal platform, the direct access to the infrastructure features of given hardware is limited to a non-mission mode compute core (RPU). This component exposes the actual internal state

	to the mission mode (FRACTAL node) services in a way that still allows to adhere to safety regulations. In addition, it provides monitor services.
Contribution to achieving the T4.1 objectives	This component defines the Versal FRACTAL node power control and monitoring accesses that are available to the time-triggered control instance. In safety-centric platforms, the power control and monitoring features defined in this component are only accessible through "Versal RPU access for Power Services".

Additionally, several activities have been carried out. In particular,

- In T4.1, the Data Compression for low-power services (WP4T41-01) has been designed and implemented. In addition, the effectiveness of this technique has been proven to evaluate this component.
- HATMA has been designed and implemented in hardware based on an NoC multi-core architecture. Runtime adaptation to slack events has been demonstrated while balancing a trade-off between adaptability overhead and energy saving.
- An agreement protocol was also developed in T4.1 for low-powered devices (WP4T41-05), here a wireless time-based synchronization was achieved with multiple microcontrollers. A PTP-like synchronization method was developed, which has one master and several slave devices. Additionally, a master selection routine was created for this task. This routine program is responsible for selecting a master device after the bootup process of all the peer devices. In case the master device loses its connection with the network, necessary fallback methods were also created to accommodate the selection of a new master device.
- Verification and validation of the LEDEL library have been carried out. They show that the LEDEL library can be used to implement neural network algorithms, compiled or exported to ONNX files, and then transfer to the FRACTAL node. Even more, programs created using PyTorch that define a neural network that can be exported to ONNX file can be as well transferred into the FRACTAL node, loaded using LEDEL, and perfectly executed.
- ETH investigated hardware specialization and microarchitectural modifications to increase the energy of PULP-based energy efficiency. Furthermore, ETH explored fine-grained power gating to minimize the power consumption powering off unused hardware units and allowing tuning the hardware configuration for specific application scenarios.
- For the Versal based platform the specific controls and monitoring of infrastructure features have been validated and checked against requirements to demonstrate scalability of a FRACTAL node. This informed the derived Versal power service components that have been developed with safety and isolation as a particular design constraint.

3.1 Document Organization

The document is organized as follows: in Section 4 we present a high-level picture of the Fractal solutions developed in WP4. Then, we report the detailed description, design and implementation, and evaluation and testing of data Compression for Low-Power Services, HATMA, low Power services for PULP systems, Versal RPU access for low Power Services, agreement Protocol for Low-Power Services, and Versal Isolation Design- Functional Safety, in Sections 5, 6, 7, 8, 9, 10 respectively. Next, we describe how the LEDEL library is validated for low-power services in Section 11. Finally, in Section 12, we draw the conclusions.

4 High-Level Picture

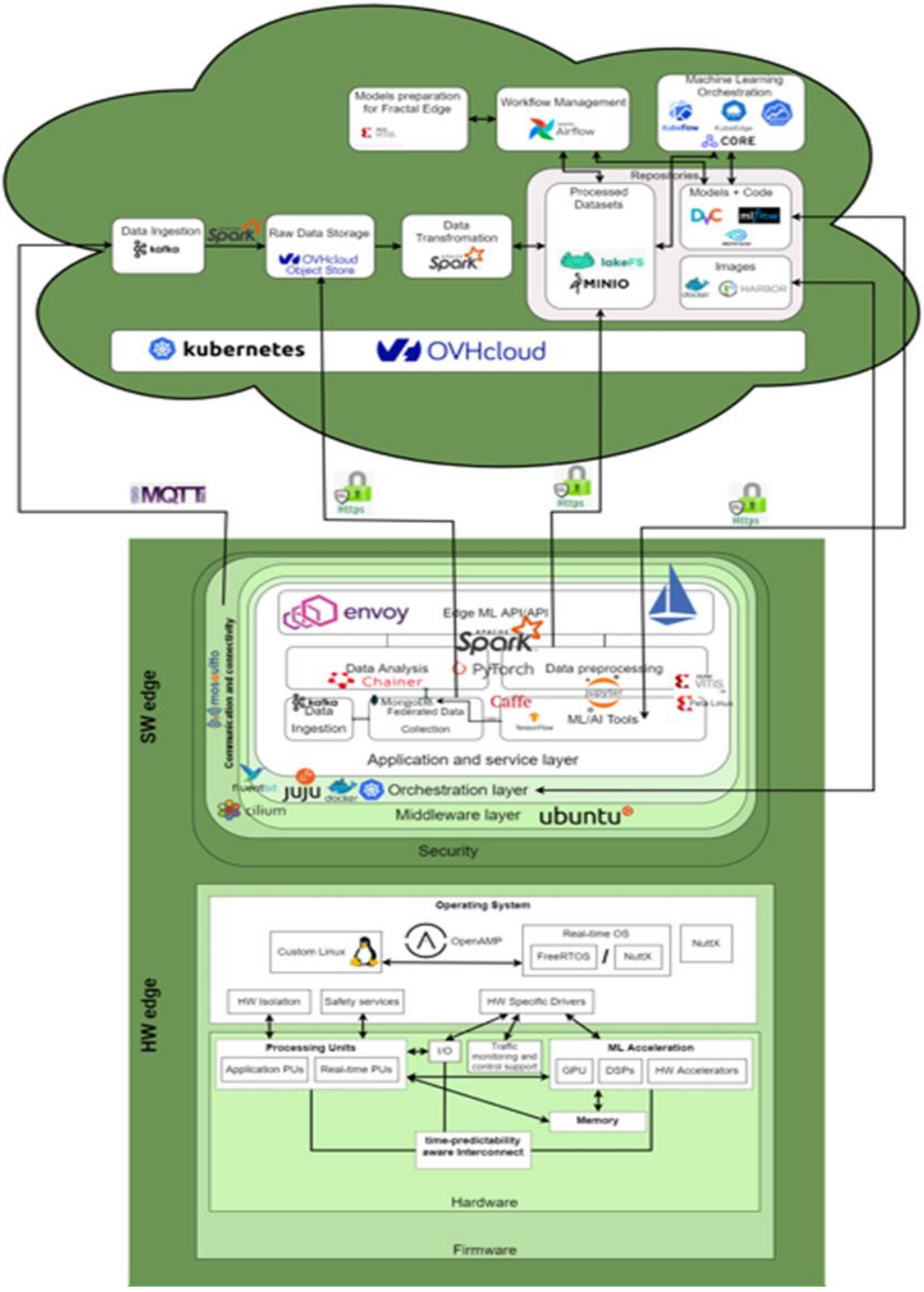

Figure 2 The big picture of the FRACTAL project

The big picture of the project, illustrated in Figure 2, is a holistic representation of the FRACTAL solution which aims to illustrate the assembly of the components to build the FRACTAL node. It provides an answer to the use cases' requirements, which are the functional and non-functional needs captured by FRACTAL use cases at the beginning of the project. Starting from these requirements, a set of features could be established to give a technical notion to the requirements.

Beginning with the platforms' hardware and low-level software layers, one may interpret the picture from the bottom up (OS, services, drivers...). The various edge application software layers are integrated on top of them. Finally, this node communicates with its cloud counterpart that includes for instance learning and orchestration.

In the following subsections we report how each of the T4.1 components mentioned in Chapter 3 is integrated into the big picture.

4.1 Data Compression for low power services

The Data Compression component has been developed at the software level thus, it could be regarded as a part of the software edge of the big picture. Specifically, it might be considered as a component of the data preprocessing block highlighted in Figure 3 and can be integrated into any component needing data compression/decompression functionalities.

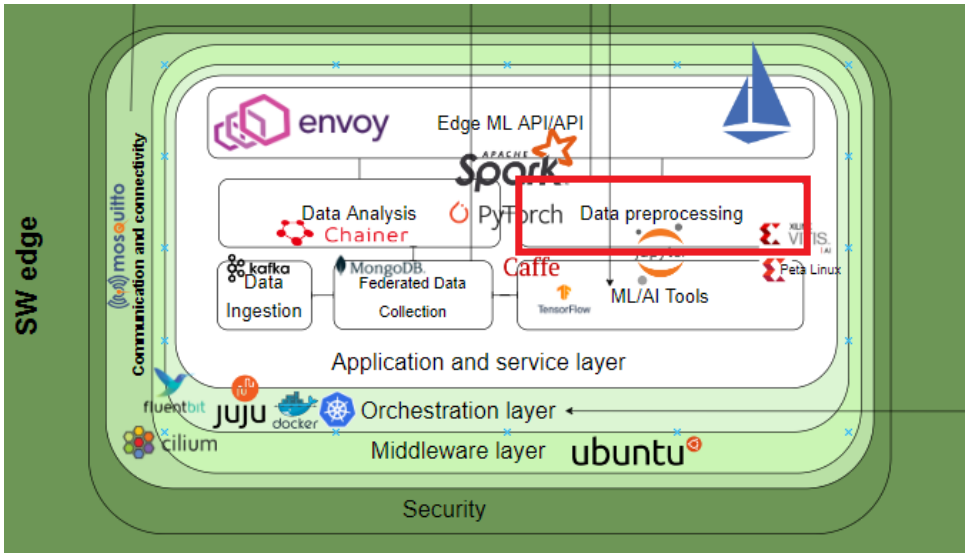

Figure 3 The integration of Data Compression component in the big picture

4.2 HATMA

HATMA employs a time-predictable/aware interconnect to provide communication services between processing units, I/O, shared memory, and ML accelerators. Traffic monitoring and control are supported through time-triggered scheduling of system resources. HATMA integrates vertically with the operating system to deliver system services and horizontally to other FRACTAL nodes to deliver hierarchical services.

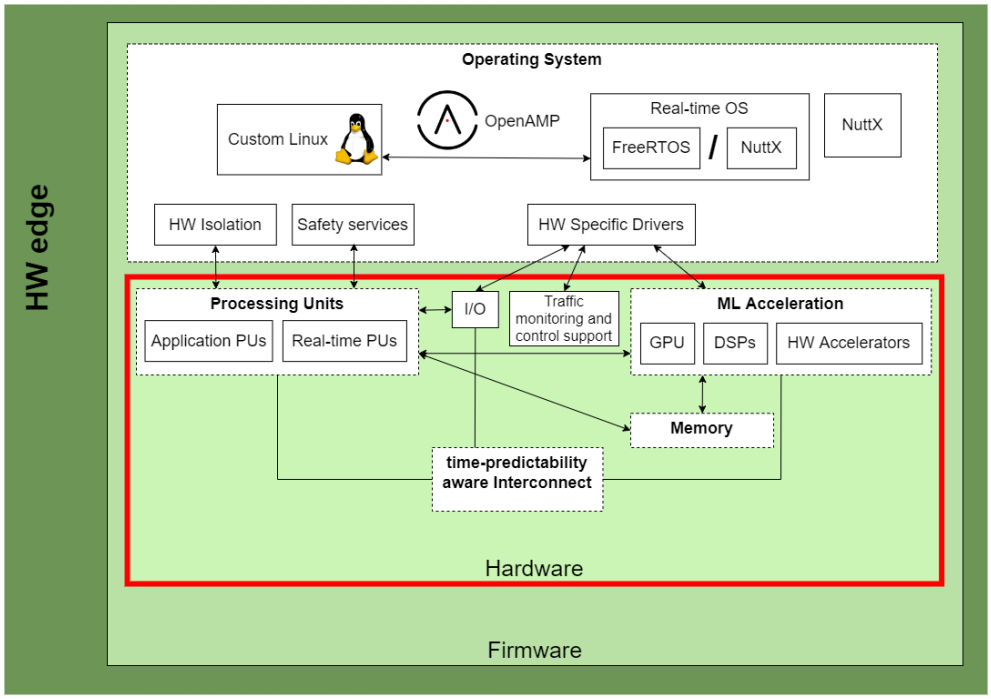

Figure 4 HATMA integration in the FRACTAL big picture

4.3 Low Power services for PULP systems

The component aims at enhancing the low-power services of PULP systems. Fine-grained power gating has been explored to minimize the power consumption powering off unused hardware units and allowing tuning the hardware configuration for specific application scenarios. Use cases built upon PULP-based end nodes will take advantage of the additional features to build low-power and efficient IoT applications, reaching higher battery life.

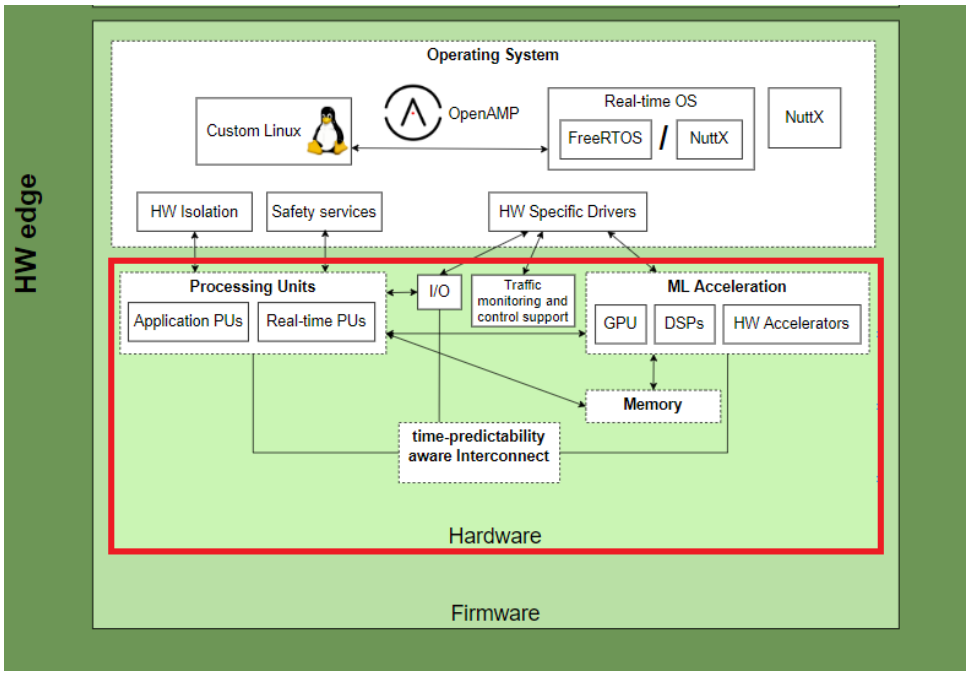

Figure 5 The integration of low power services for PULP systems in the big picture

4.4 Versal RPU access for Power Services

This component provides means to access dynamic power and frequency scaling features on Versal in safety-oriented platform designs. Thus, FRACTAL node-level services and software applications from use-cases can query or control power state through this access layer. Low power features must utilize this safety access channel introduced by this component in order to adhere to isolation defined for a safety-centric design as in Section 4.6.

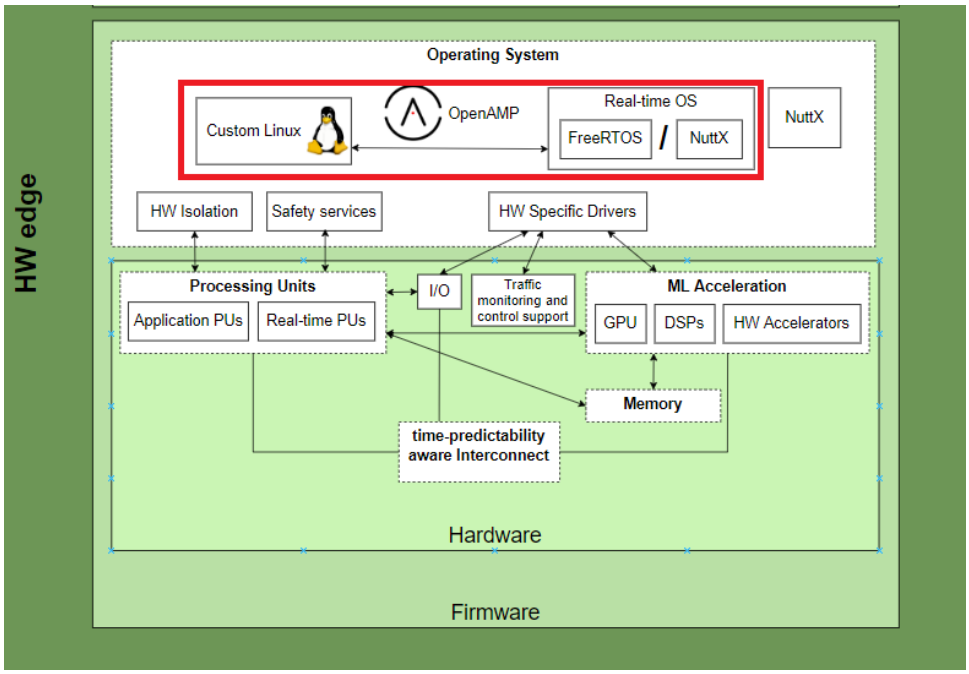

Figure 6 The integration of Versal RPU access for low power services in the big picture

4.5 Agreement Protocol for Low-Power Services

For the agreement protocol, we are using FreeRTOS as a component to perform tasks for synchronization of the peers. It integrates into the Hardware platform of the big picture as shown in Figure 7**.** The FreeRTOS tasks handle the whole synchronization protocol starting from the selection of master among a group of nodes, sending SYNC messages, and handling of cases when the peer connection is lost. The nodes are low powered microcontrollers (ESP-32).

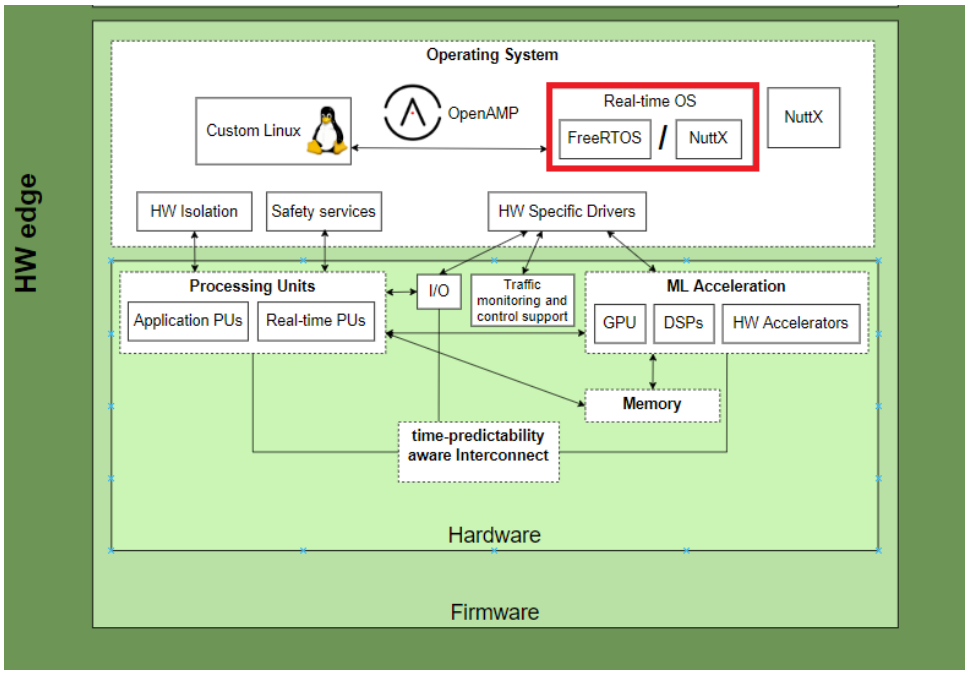

Figure 7 The integration of agreement protocol in the big picture

4.6 Versal Isolation Design- Functional Safety

To fulfil the requirements for a safety-focused Versal design the basic accessibility of the generic platform is restricted as a base for proper functional safety capabilities. This component defines a particular policy to support a safety channel that encapsulates the device and board infrastructure. The definition of the safety channel has been provided in Deliverable *D2.2 - 6.3.2.3 Versal Safety Channel Architecture* and encapsulates the power functionality with a specific RPU core. Resources within this safety channel are accessible from the application layer through this RPU power control software component as described in Section 4.4.

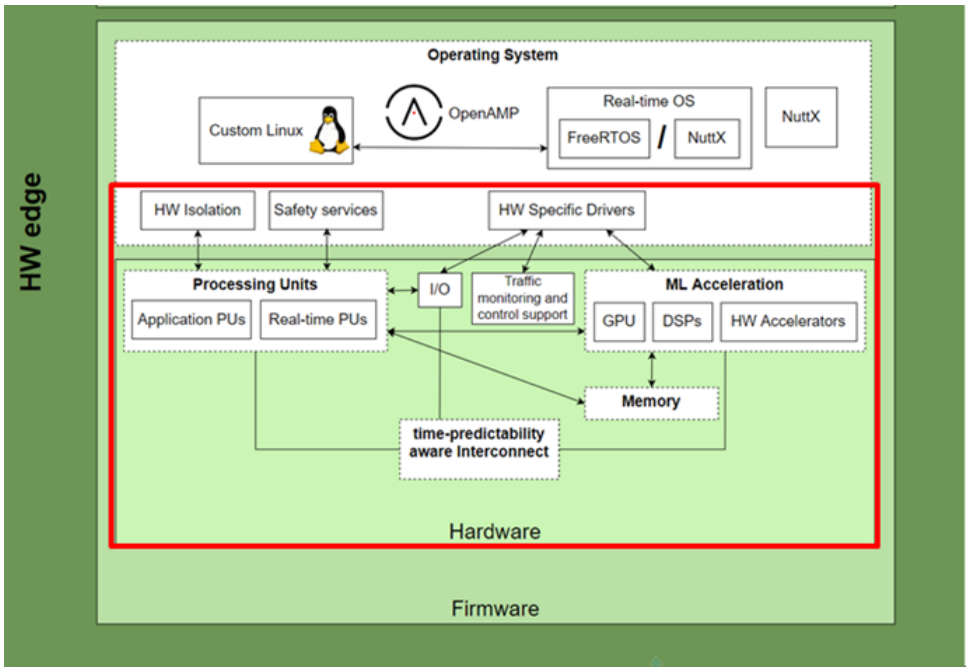

Figure 8 Location of the Versal isolation features in the big picture

5 Data Compression for Low-Power Services - WP4T41-01 - ROT

5.1 LZW compression technique

As it was stated in the deliverable D4.1, a variety of compression techniques could be used in energy-constrained systems in order to minimize the data exchanged among nodes and thus reduce the network energy consumption. These techniques could be classified based on some parameters such as application type, data quality, coding schemes, and data type. In addition, we have provided the state of the art of possible data compression algorithms and data aggregation techniques that could be used in such networks. Moreover, according to the Use Cases requirements, we have set out some specific features which should be taken into careful consideration in order to choose a suitable data compression technique. Satisfying these features, it was decided to choose the LZW algorithm. The LZW algorithm is a very common compression technique that is typically used in text file, TIFF, GIF and optionally in PDF. The main features of the algorithm:

- Lossless algorithm, no information loss during the processing;
- Excellent compression/decompression speed;
- Good compression ratio;
- Dictionary-based operating principle (the algorithm is an evolution of the first known dictionary algorithms, LZ77 and LZ78);
- Ability to compress streaming data without knowing the data before compression;
- Simple implementation, with a potential for very high throughput in hardware implementations;

LZW is the foremost technique for general purpose data compression due to its simplicity and versatility. It is the basis of many PC utilities that claim to double the capacity of your hard drive. LZW is mainly used in the compression of text and images in the most well-known existing formats. Although its diffusion is still rather limited compared to algorithms based on LZ77, it is widely used Unix file compression utility compress and is used in the GIF image format.

Its main advantage over other algorithms is that it uses a dictionary that is built up as it reads a file or stream of data.

The working principle of a dictionary algorithm consists in replacing a set of recurring symbols with abbreviated symbols in the output data. The idea relies on reoccurring patterns to save data space.

To give an example of an algorithm with a dictionary, we can imagine compressing an SMS by replacing the most used words with numbers. For example, using 16bit we could have 65535 words in our new compressed dictionary, more than enough to compose an SMS with a simple language. The saving of this imaginary algorithm in

terms of data would be considerable; in fact, each word could occupy even only 2 symbols (in computer science each character of text occupies 8bit), with such a compression we could get to compose SMS with a much higher number of words. compared to a classic SMS.

As we often have to work with very varied and non-recursive data, finding a good algorithm for many cases is no small feat, but it is precisely what LZW guarantees.

The working of the LZW algorithm for compression is as follows:

```
STRING = INPUT CHARACTER
WHILE there are characters in DO input
CHARACTER = read unput
IF STRING + CHARACTER is present in the then dictionary
STRING = STRING + CHARACTER
ELSE
write the dictionary symbol of STRING
add STRING + CHARACTER in the dictionary
STRING = CHARACTER
END of IF
END of WHILE
write the dictionary symbol of STRING
```

Figure 9 LZW compression algorithm

To better understand the process, let's take for example a word to be compressed consisting of 11 characters: ABCABEFGHIL.

The dictionary is initialized with 256 symbols, an array of strings of single character, which also contain each of the single characters of the word mentioned above: A B C E F G H I L.

After the first reading, the input buffer will consist of: A.

In the second step, the algorithm reads B, and the buffer becomes: AB.

AB is a symbol not present in the dictionary, at this point the algorithm inserts the new symbol AB in the first position available in the dictionary, i.e., position 256 which will therefore contain the new symbol.

Once this symbol has been added, the algorithm will produce an output corresponding to the position of the last symbol found, i.e., the value of the position associated with character A, precisely: 65.

The algorithm proceeds by discarding the symbol sent out from the buffer and appending the new byte to be read at the last read, the buffer thus becomes BC. The new symbol BC is not present in the dictionary thus the algorithm inserts the new symbol BC in position 257, and always returns as output the value of the position of the last symbol which is in this stage 66 associated with the symbol B.

The process continues and the buffer becomes CA which is not present in the dictionary too, thus the algorithm outputs the position of the symbol C that is: 67, and inserts the new symbol CA in position 258.

In the next step the buffer becomes AB, AB is a symbol in the dictionary, the algorithm does not produce any output and continues with the reading (the compression begins).

The buffer then becomes: ABE which is not present in the dictionary. Therefore, the output is 256 (the value of the symbol AB), and the new symbol inserted in the dictionary is: ABE with the value 259 and the buffer becomes EF.

The algorithm proceeds in this way until the incoming data ends.

Decompression proceeds in a very similar way to compression, so the algorithm of decompression can be expressed as in Figure 10:

```
Read OLD_CODE
Write OLD_CODE
CHARACTER = OLD_CODE
WHILE there are characters in DO input
Read NEW_CODE
IF NEW_CODE is not in the THEN dictionary
STRING = string in the dictionary with code OLD_CODE
STRING = STRING + CHARACTER
ELSE
STRING = string in the dictionary with NEW_CODE code
END IF
Write STRING
CHARACTER = first character in STRING
add OLD_CODE + CHARACTER in the dictionary
OLD_CODE = NEW_CODE
END WHILE
```

Figure 10 LZW decompression algorithm

Although the LZW algorithm is quite simple to understand from logical point of view, its implementation brings up considerable technical difficulties from practical point of view:

1) The use of memory must be optimized within the code where the system must store a very large dictionary in a small space.
2) The search for symbols in the dictionary must be optimized to reduce data processing time.
3) Symbols must be written in a format with variable bit-size depending on the size of the dictionary during the compression phase to maximize the benefits of the algorithm.
4) Since the size of memory is limited, it is necessary to limit the size of the dictionary.
5) The dictionary must be reset when it becomes full or the compression ratio becomes inconvenient since during compression with this method the compression ratio tends to degrade very easily depending on the portion of data to be compressed.

With an eye toward achieving the main objective of the T4.1 to ensure the energy efficiency of the FRACTAL system, we proposed some possible solutions of the aforementioned challenges such as:

1) To limit the space occupied by the dictionary, you can keep in memory only a list of pairs of variables for each symbol of the dictionary. In other words, a prefix/last character pair, while the code in the dictionary will be nothing more than the index of the array used. In particular, it can be seen that each new symbol to be inserted in the dictionary is nothing more than an old symbol in the dictionary with one last extra character. It is, therefore, sufficient to memorize the value of the old symbol and the character that forms the new symbol instead of memorizing the whole new symbol which could be as long as the dictionary itself. For instance, by applying this technique on the first memorization step of the previous example, the insertion of the symbol AB in the dictionary becomes prefix 65-character B, array index 256.
2) With the aim of solving the problems of the search times of the symbols in the dictionary, a Hash algorithm is used in many implementations of LZW. Although this algorithm is more efficient than other data structures and it provides constant time for searching, it has some disadvantages. Specifically, using the Hash algorithm can have a considerable cost. It demands a necessary increase in the memory to be dedicated to saving the dictionary. Therefore, a large portion of the memory will be wasted to the detriment of the maximum compression capabilities.

   An optimal solution has been analyzed which consisted in making the most of the hypothesized structure to store the dictionary. In particular, we considered that each prefix-character element of our structure is used to form other symbols of the prefix/character + second character type, and the maximum number of derived symbols will never be higher than 255 due to

the operating principles of the algorithm. Then, we can add an array of 255 pointers for each prefix-character pair which point to the pairs derived from the source ones. The index of these pointers will be used with the second character. Despite the significant improvement this solution will lead to, where the cost of research with this structure would always be equal to 1, it involves excessive use of memory. One way to solve this drawback is to reduce the pointer arrays in order to create linked lists. Faster processing times are going to be required in UC6, where the analysis reported here is going to be implemented and integrated.

3) With the purpose of maximizing the effectiveness of this compression algorithm, the output values should be stored using only the needed space. This means that, for example, when a symbol with a position 256 is added to the dictionary which has a maximum size of 16 bits, the written value must occupy only 9 bits of the memory rather than 16 bits, as long as it does not need more bits for storage. In addition, the bit size has to grow dynamically according to the momentary size of the dictionary. This one will be solved and developed in another European project.

4) In order to limit the size of the memory, we defined the size of the dictionary. The best method used in this case is to give a limit to the dictionary based on a bit-size of the code-word. By doing so, there will be no wasted information in the compressed output. This means that if, for example, we use code-word of 14 bits, the maximum number of symbols in the dictionary will be

$$2^{14} - 1 = 16383\ symbols$$

5) Since the compression ratio using the LZW algorithm tends to decline easily depending on the percentage of data to be compressed, the dictionary must be reset to prevent compression from degrading once the dictionary is full. In order to do so, we propose two methods which might be implemented in the next period. We are going to report this improvement on the related UC deliverable. These methods are:
   a) The first one is based on resetting the dictionary to return to the initial situation in which we have, for example, 255 symbols. In addition, one way to indicate to the decompression process that the dictionary starting from the special symbol has been reset is to reverse that symbol in the dictionary.
   b) The second technique is more refined. It focuses on the continuous analysis of the compression rate and resetting it every time the dictionary degrades.

In the following subsections, we report the implementation of LZW data compression, perform a set of practices and processes in order to validate our component, and determine if our component meets the project KPI metrics. Additionally, we address the use case which utilizes our component and how it is integrated into that use case.

5.2 Design and implementation

The data compression component was designed to be split into two subcomponents: *Compression* and *Decompression* which both have a single entry-point for their respective action to be performed on the file taken as input.

The following flowchart, Figure 11, explains the data flow of the Compression subcomponent library which takes as input path, name, and extension of the file to be compressed. It will read the original file and perform LZW compression to give as output a file on the same path, with the same name, and extension ".lzw".

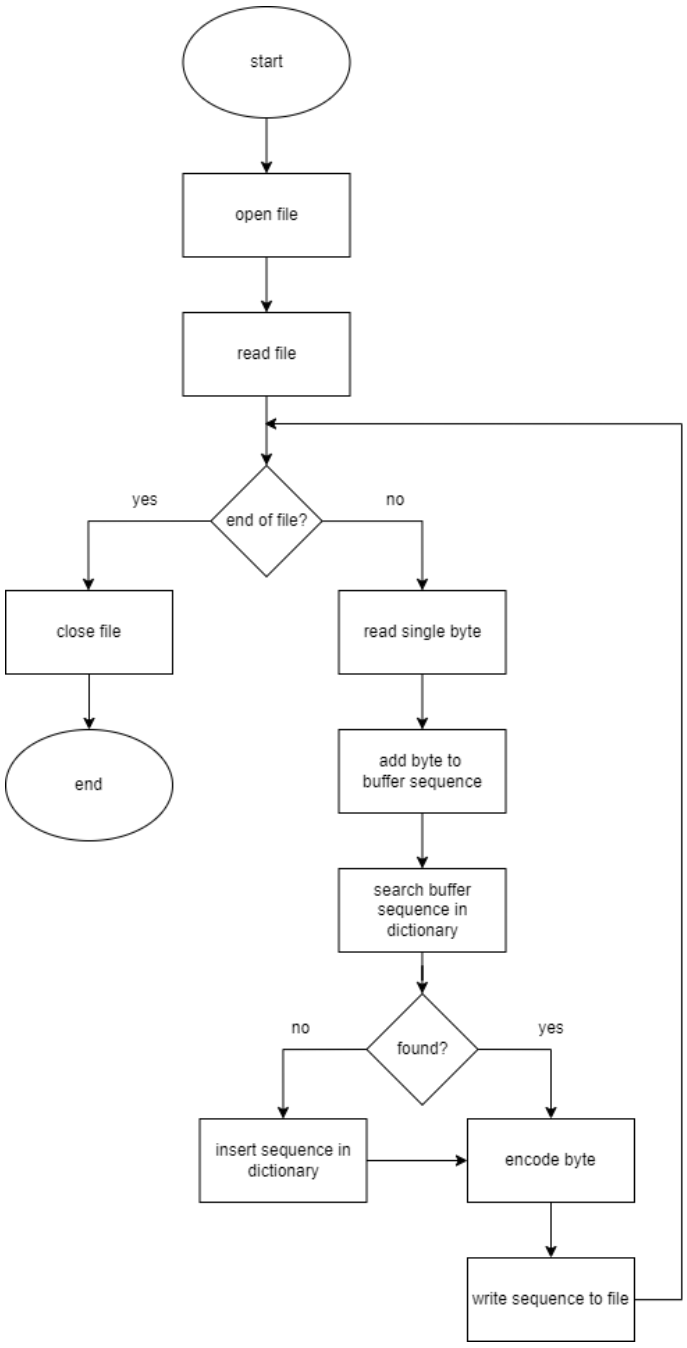

Figure 11 The flow chart of the compression process

The flowchart in Figure 12 explains the data flow of the Decompression subcomponent library, which takes the same input as the Compression subcomponent (the file extension must be the same as the output file, since the input file is supposed to always be a previously compressed file, hence with the extension ".lzw".)

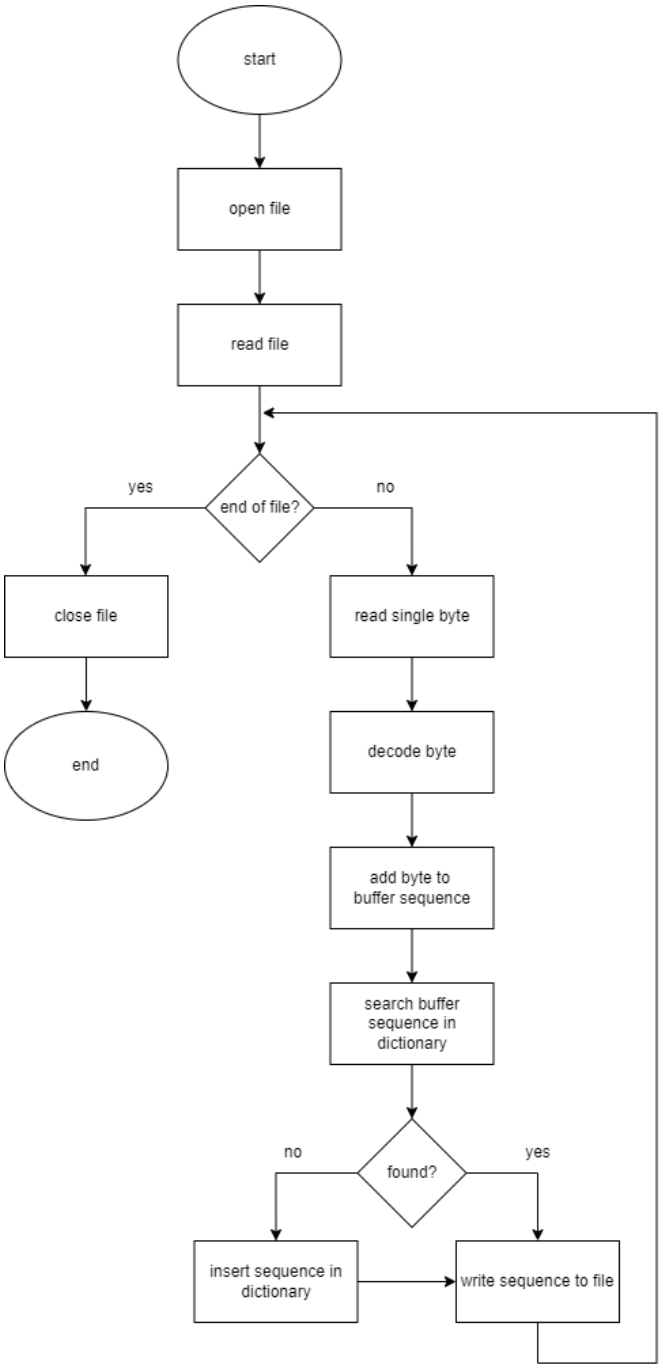

Figure 12 The flow chart of the decompression process

The component was implemented as a software library, developed in C++, as a wrapper and extension of an open-source LZW C++ implementation. [9]

5.3 Testing and evaluation

The Data Compression component can be evaluated in several ways. Considering the different types of parameters, it is possible to define a series of tests that measure or calculate these parameters, through which it is, therefore, possible to provide evidence of the effectiveness of the component.

Data compression ratio, also known as **compression power**, is a measurement of the relative reduction in the size of data representation produced by a data compression algorithm. It is typically expressed as the division of uncompressed size by compressed size.

$$CR = \frac{US}{CS}$$

where CR is the *Compression Ratio*, US the *Uncompressed Size* of input file and CS the *Compressed Size* of output file. This formulation applies equally for compression, where the uncompressed size is that of the original; and for decompression, where the uncompressed size is that of the reproduction.

Another measurement of size reduction is **Space Saving**, which is defined as the reduction in size relative to the uncompressed size:

$$SS = 1 - \frac{CS}{US}$$

where SS is the *Space Saving*, CS is the *Compressed Size* the of output file, and US is the *Uncompressed Size* of the input file. This measurement provides a percentage value of the space saved with the compression process.

The third parameter to evaluate our component is the **time** for the compression and decompression processes, which is an important indicator of the efficiency of the algorithm.

The Lab tests were conducted on a Windows 10 computer with both compression and decompression algorithms compiled as a library.

During the lab tests, we focused on the **saved space** and the **time** needed for both compression and decompression.

Due to the fact that Data Compression component is going to be integrated in Use Case 6, tests have been conducted using sample files provided by the use case. This set of sample files is composed of images in JPEG format and audio in WAV format, and the tests have been performed using 14 different audio files. For further details about integration refer to section 5.3.1.

All the files that were used during the tests were successfully compressed apart from 3 cases. In the first one, the file size was the same as the original, and in the other two the size was greater than the original. However, it was expected that some files could not be compressed by one compression algorithm because no lossless compression technique can efficiently compress all types of data/messages. [10]

Given the limits of lossless data compression algorithms, we have reported the results of the successful tests performed to estimate the value of *Space Saving* in Table 2. The average compression and decompression times was, on the other hand, estimated among all the results of the tests.

As we notice from table below, the algorithm meets the KPIs identified during the requirements identification phase.

Table 2 KPI and metric of component WP4T41-01

KPI description	Means of assurance	Expected	Measured/ Achieved
ACCURACY Saved space	Lab test	> 10%	12%
RESPONSE TIME Compression	Lab test	< 3s	2.3 s
RESPONSE TIME Decompression	Lab test	< 3s	2.6 s

The next steps will be to perform improvement to meet the specific requirements of UC6 (Intelligent Totem) and for integration and field tests.

5.3.1 Integration into UC6

The Data Compression component is going to be integrated into UC6, which is expected to use to reduce the amount of the data transferred between nodes. In particular, the Data Compression is used when the Load Balancer component is triggered, meaning the current node is overloaded and needs to distribute workloads.

As we mentioned, the component could be utilized to compress/decompress various media types. Referring to *D8.1 - Specification of Industrial validation Use Cases*, the first proposed way of integration was to use our component in the compression/decompression of the images taken by the node camera. However, the images provided by the node have JPEG format, which is not suitable for the LZW algorithm. Furthermore, JPEG is a method of compression for digital images, so using the LZW compression on a .jpeg image does not produce an admissible result. These images could be sent to other nodes while all the UC6 requirements are fulfilled among which is the time constraint.

However, using it with the type of image captured by the node's camera in UC6, will do a reverse job and expand the output file. Therefore, we decided to use the compression technique to compress the audio files captured by the node's microphone. In fact, the tests reported in section 5.3 have been conducted using sample audio files.

6 Hierarchical Adaptive Time-triggered Multi-core Architecture (HATMA) - WP4T41-02 - SIEG

The Hierarchical Adaptive Time-triggered Multi-core Architecture (HATMA), as introduced in D4.1, delivers services through adaptation on two levels, the system and node levels. The system-level encompasses nodes interconnected through a time-triggered off-chip communication network. Access to the network is provided via a gateway, which encrypts and decodes messages before injecting them into the network to their intended destination. On the other hand, the node level encompasses processing elements and a time-triggered network-on-chip to facilitate message exchanges within the node. Each processing element within a node is interfaced with a network interface which injects messages from the processing element into the respective network.

As illustrated in Figure 13, each processing element includes computational cores for application services, an adaptation logic, and a network interface for accessing the adaptive communication network and the network-on-chip. In the case of the gateway core, the network interface is used to access the off-chip communication network.

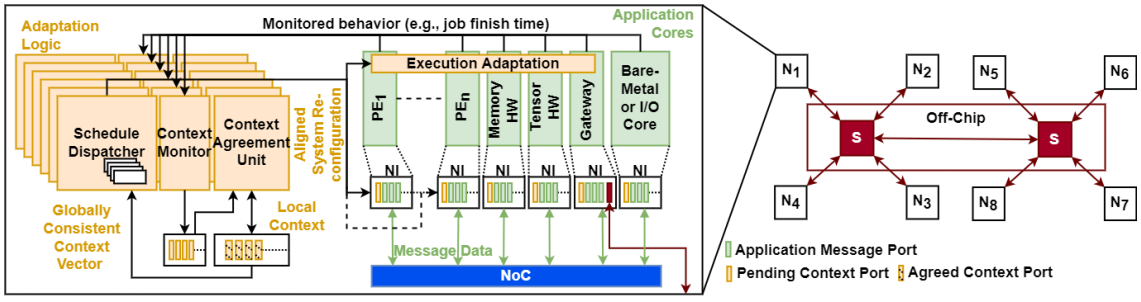

Figure 13 Hierarchical Adaptive Time-triggered Multi-core Architecture (HATMA)

A consistent state of all resources facilitates hierarchical adaptation in HATMA at periodic points in a scheduled period. The time-triggered schedules maintain such consistency. Adaptation is then achieved at the different hierarchies by an aligned switching of schedules in response to context events such as dynamic slack for low power and energy management services.

6.1 HATMA Subcomponents

The adaptation logic of HATMA manages and coordinates adaptation services in a distributed manner and comprises the context monitor, context agreement unit and schedule dispatcher. Therefore, hierarchical adaptation is synchronized across all adaptation logics at the respective hierarchy. The establishment of a consistent system state is taken in a distributed manner by all context agreement units. The

adaptation logic establishes a global view of the system state. A global system state is realized through observation, reporting, broadcast, distribution and agreement of local contexts. Therefore, the selection and dispatch of the following schedules for hierarchical adaptation are possible based on the agreed contexts as described in Figure 14:

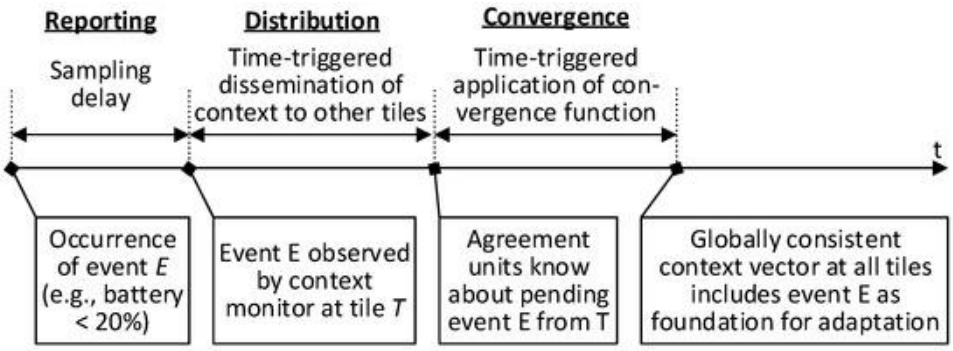

Figure 14 HATMA Adaptation Logic Process

Context monitor (CM): The CM observes the state of its local system resource and generates local context information. Such local context can be the status of an application being executed on a core. The local resource is periodically observed for each scenario based on the precomputed adaptation schedule. The precomputed adaptation schedule for the periodic sampling of the local resource is synchronized to the system schedule to enable a potential schedule change without detriment to the system.

Context Agreement Unit (CAU): Through the Hierarchical Interactive Consistency Protocol (HICP), the CAUs establish a globally consistent context vector (system state) by collecting and agreeing on the local contexts reported by all CMs at the respective hierarchy.

a. All CAUs initiate a synchronized context distribution phase based on a precomputed adaptation schedule. Through the HICP, the local context is distributed in a double-ring topology through the adaptation communication network. Local context is sent to and collected from the next neighbors in the network and concatenated to produce a global context vector.
b. At the end of the distribution phase, all resources possess identical system information. At this point, HICP converges, and the CAUs agree on the system state. The agreed context vector is the globally consistent context vector representing the system status at all resources. The schedule dispatcher uses the globally consistent context vector to determine the next schedule and perform an aligned switching of schedules at runtime.

Schedule Dispatcher: The precomputed multi-schedule graph is stored in this unit. The next dispatched schedule is chosen based on the globally consistent context vector from the CAU mapped to the edges in the multi-schedule graph. Based on the system state, adaptation is achieved by choosing the next schedule and dispatching it for an aligned reconfiguration of the respective hierarchy of the system. At the start

of the adaptation window, the CMs are triggered, and an instance of the HICP is initiated. At the end of the window, the schedule dispatcher performs an aligned schedule change in adaptation to the agreed observed contexts.

6.1.1 Hierarchical Interactive Consistency Protocol (HICP)

The context event to be monitored influences the timing of instances of the HICP at the various hierarchies. Context events are differentiated based on their urgency. The urgency of a context event indicates a period when adaptation to the event can yield a benefit before it loses its value for adaptation. For example, urgent slack events require fast switching at the core and node levels to trigger low power and energy management services. On the other hand, adaptation at the system level is much slower given the number of resources and the communication cost of the off-chip communication.

Furthermore, adaptation generally introduces overhead. Thus, the system level's adaptation frequency has a coarser granularity to balance the increased overhead. Slowly changing events such as the available power in battery-operated devices or events critical to overall system reliability are prioritized at the system level given the adaptation window and the constraints of adaptation overhead.

6.2 Design and implementation

HATMA is instantiated in hardware using the Xilinx Vivado design suite and Vitis toolchain for synthetic slack scenarios to validate the HATMA low power service. A multi-core architecture consisting of a Zynq processing system and 3 Microblaze cores, the adaptation logic and the Adaptive Time-Triggered Network-on-Chip (ATTNoC) described in deliverable D4.1 was instantiated on a Xilinx Zynq UltraScale+ MPSoC ZCU102 FPGA board.

The adaptation logic for HATMA is key to the timely and consistent reporting, agreement and adaptation of HATMA to context events. During each adaptation period, local resources are observed, and local context is reported. Then, through the HICP, observed local contexts are broadcast to neighboring resources and an agreed system state at the respective hierarchy is established. The agreed system state allows for an aligned hierarchical adaptation to the observed events. Therefore, adaptation at the node level is consistent with the global system state.

6.2.1 Context Monitor (CM)

The CM is triggered to monitor and report the status of its local resource, for example, dynamic slack of the jobs of an application service running on a core. The CM can be realized in hardware and software and is designed to poll and report local context information.

Synchronous events are predictable, and CMs can be scheduled to observe such events. On the other hand, asynchronous events are random and are therefore observed through periodic monitoring of system resources. In either case, there is a polling delay between the occurrence of a context and its reporting. Therefore, a

time-triggered adaptation of system resources is scheduled periodically, where the granularity is a trade-off between a reduced polling delay and adaptation overhead.

A CM implemented in hardware using a finite state machine is described in Figure 15. The time-triggered machine reacts to an external trigger and a reset. It starts in the *initial* state where no context is observed, and its reported context is initialized to null. When triggered, it transitions to the *encode* state. It monitors and records the local resource state based on observable information (dynamic slack) and the context time (timestamp). When the observed local context is encoded into a 32-bit bitstring, it transitions to the *output* state when the encoded context bitstring is driven at the output. At the output state, the machine transitions back to the *initial* state when reset or to the *encode* state when the CM is triggered. The machine remains in the *output* state when no trigger or reset is present.

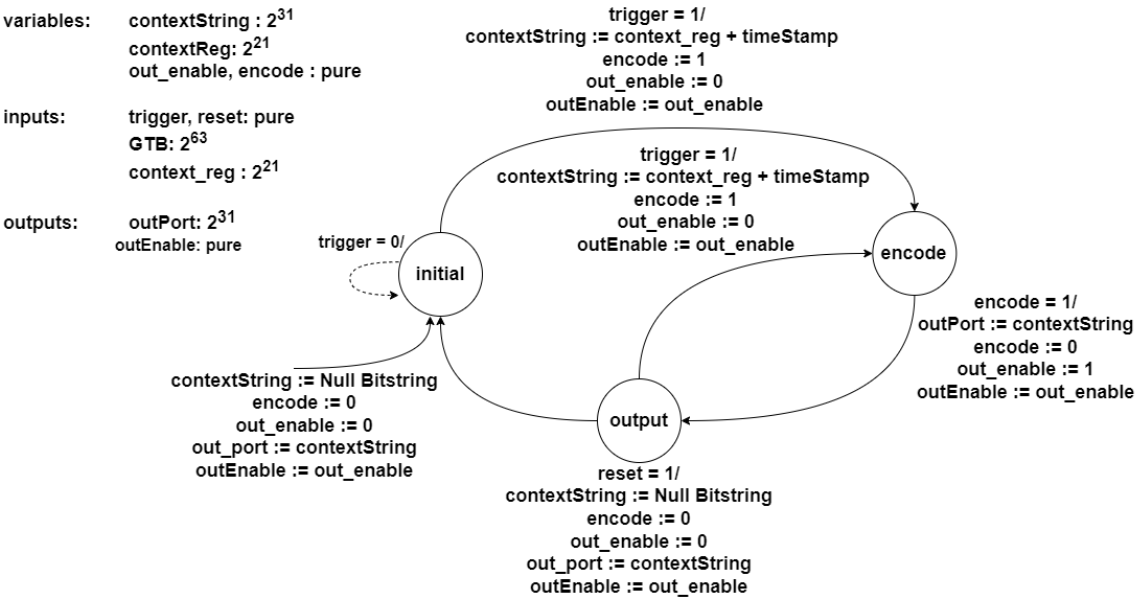

Figure 15 Context Monitor Implemented in Hardware using a Finite State Machine

In each case, triggers are generated based on the precomputed adaptation schedule. Such operation of the machine allows for a periodic observation and reporting of context events in multiple instances. The outputs produced by this machine are the encoded local context and *outEnable* to indicate the availability of the observed context.

6.2.2 **Context Agreement Unit (CAU)**

The CAU manages instances of the HICP when reported local contexts are broadcast to neighbors in a dedicated double-ring half-duplex network topology. Each instance of the HICP across multiple CAUs is synchronized to realize an aligned convergence of the HICP. On the convergence of HICP, each CAU possesses an identical global view of the system state, which is the basis for hierarchical adaptation.

The CAUs are triggered periodically within adaptation windows in the system schedule. For example, in one instance of the HICP, the local context provided by

CMs is frozen in time and agreed upon by all CAUs. The following instance of the HICP is used to agree on the next set of reported events, as illustrated in Figure 16. Each received local context is saved to the local register and relayed to the next neighbour until the local context gets to the initial broadcaster when the instance of the HICP is converged.

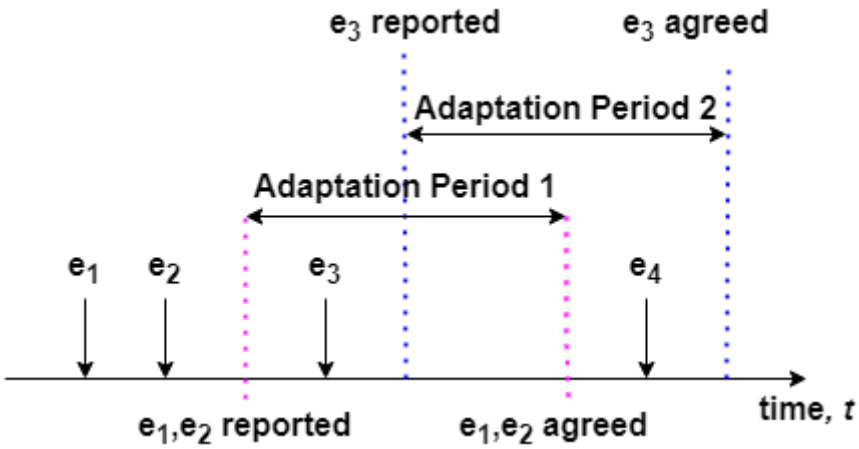

Figure 16 Periodic Adaptation to Context Events

A CAU implemented in hardware using a finite state machine is described in Figure 17. In the figure, the machine reacts to a trigger and a reset. It starts in the *initial* state where its local registers are initialized and its output disabled. Each received context bitstring is assigned a unique slot in the local registers based on its CAU ID. When a trigger is present, it transitions to the *poll* state, where context events reported by CMs are saved in two registers and held as input context. In this state, the CAU's local ID is also the CAU ID of origin for the input context.

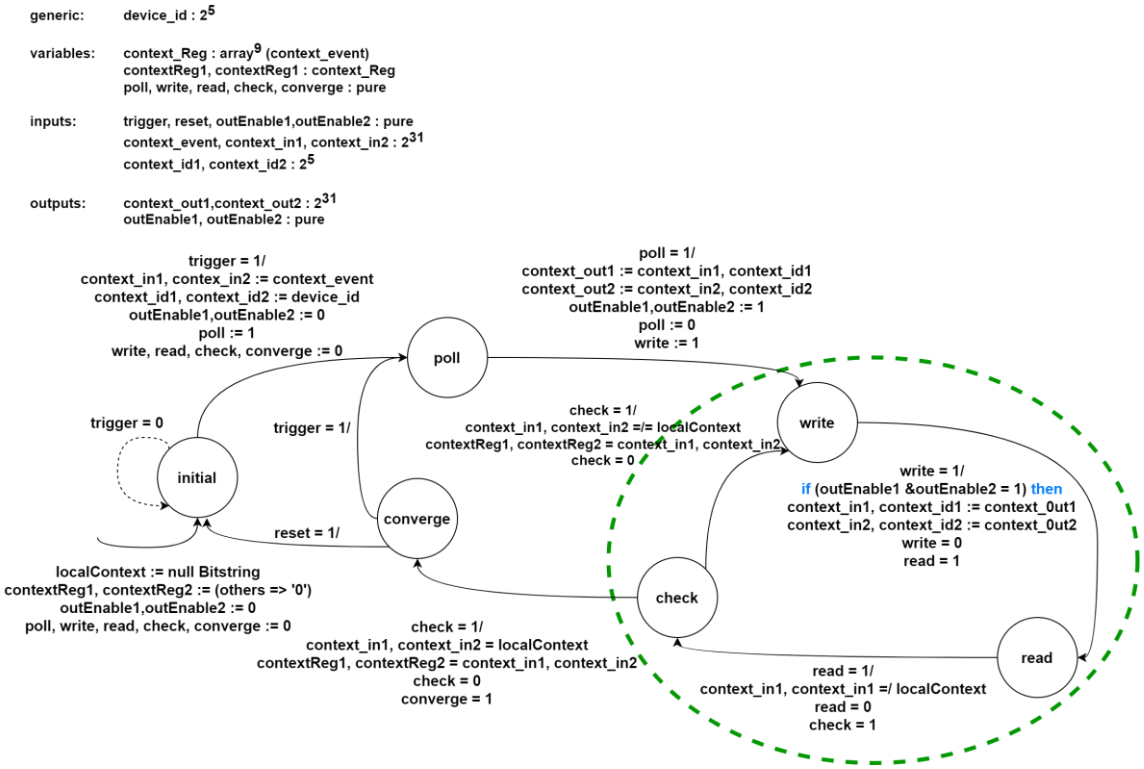

Figure 17 Context Agreement Unit Implemented in Hardware using a Finite State Machine

When the context event is collected and saved, the machine transitions to the *write* state, where the input contexts are written to the two output ports. The CAU ID of origin and an indication of the context availability are also driven as outputs. At the end of the writing process, the machine transitions to the *read* state, where inputs from neighbors in the double-ring are read synchronously from the input ports. These contexts from neighbors are saved to the local register and held as input contexts. The CAU ID of the context origin of the received contexts is also stored in this state. The machine then transitions to the *check* state, where the received CAU IDs are checked against the local CAU ID. If any of the received CAU IDs match the local CAU ID, the machine transitions to the *converge* state and terminates the HICP. The machine returns to the *write* state and drives the respective inputs to the output ports if the CAU IDs do not match the local CAU ID. The machine maintains the *write-read-check* loop synchronously across all CAUs in the network until it transitions to the *converge* state, as shown in Figure 17 (Green circle). When in the *converge* state, the machine transitions back to the *initial* state when reset input is present or to the *poll* state when a trigger is present. The machine remains in the *converge* state when no trigger or reset is present.

Triggers for the CAUs are based on the current schedule and are synchronized across all CAUs. The aligned and periodic triggers for the CMs and CAUs allow for parallel instances or periods of adaptation based on Figure 16. For example, an instance of adaptation can be in the distribution phase, and the next instance of adaptation can begin with the reporting phase. In each instance, the outputs produced by the CAUs are the agreed context bitstrings and the trigger signal for schedule change.

6.3 Testing and evaluation

The timely adaptation to context events at runtime is desired to increase the efficiency of HATMA's adaptation. In addition, adaptation through HATMA should also result in energy savings for low-power applications. Therefore, the evaluation of HATMA's adaptation for low-power is based on four key metrics: polling delay, time to convergence, overhead and energy saving.

Polling Delay: HATMA's adaptation is time-triggered and executed based on a precomputed adaptation schedule. This periodic execution introduces a delay between the occurrence of a context event and when the adaptation logic reports such context event, referred to as the polling delay δ. Runtime events are generally asynchronous due to the unpredictability of their occurrence. To minimize the polling delay, we introduce a timing granularity τ of adaptation where the system is sampled periodically between 100 and 500µs. This range of granularity ensures that all scheduled tasks are sampled at least once for the occurrence of a slack event relevant for adaptation for energy saving. This range also ensures that sufficient time is available to adapt the system schedule. We also simulate a schedule of 20 and 100 tasks with hard deadlines and worst-case execution times (WCET) in the range of 700 - 1000µs on the architecture.

On average, 70% of application tasks are completed in 50% of their WCET [1] where the maximum polling delay is such that:

$$\delta_{\max} = \tau,$$

where the timing granularity of adaptation is applied globally to the system schedule. A 100µs timing granularity represents a frequent sampling of the system every 100µs. The more frequent the system is sampled, the lower the polling delay and the faster HATMA adapts to observed events. For example, HATMA exploits the early completion of tasks (slack events) to apply Dynamic Voltage and Frequency Scaling (DVFS) and Power/Clock gating techniques to save energy. Therefore, the system is frequently sampled for highly volatile events such as slack, in which energy saving decreases as the polling delay increases.

Time to convergence: To evaluate the performance of the HICP, we set up a VHDL testbench in which four adaptation units are interconnected in a double-ring topology. A global time base is implemented to synchronize the adaptation units, ensuring each component has a common notion of the system time. A time-triggered (TT) scheduler is implemented to provide the triggers for monitoring and agreement every 30 clock cycles. The granularity of triggers is set sufficiently large to allow each instance of the HICP to converge. A finer timing granularity could lead to an incomplete execution of the HICP. In this case, all adaptation units do not possess an agreed context event necessary for an aligned schedule switch. We show the time of convergence of HICP, which is the time difference between the trigger for agreement and the convergence of the protocol.

Figure 18 shows a 100MHz clock, the triggers from the TT scheduler and context events reported by each context monitor in the adaptation units. The context events are simulated to evaluate the time to convergence of the HICP. We observe that all context agreement units report the same context event after convergence of the HICP instance. All CAUs start in the *initial_state* where the time to convergence and reporting of the detected context event is ≈230ns (23 clock cycles). The time to convergence is also observed in the second instance of the HICP, where all CAUs agree upon the context event reported by Context Monitor 2.

Figure 18 Convergence Time of HICP

The time to convergence of the HICP represents a minimum inter-adaptation time for a given hierarchical architecture. When computing a schedule for adaptation, the minimum inter-adaptation time is considered. It is the finest granularity for HATMA adaptation.

Overhead: The system is periodically sampled to minimize the polling delay. This frequent sampling of the system results in a communication overhead for adaptation due to the broadcasts of reported context events to all adaptation units in the system. We evaluate the communication overhead for adaptation due to multiple instances of the HICP with timing granularities in the range of 100 - 500μs. Each broadcast of a context event is represented as a message from one CAU to the next. The total number of messages broadcasted for multiple instances of HICP is such that:

$$C_{ov} = 2n^2 * nSamp$$

where *n* is the number of monitored hardware resources and *nSamp* is the number of instances of the HICP for a time-triggered application. C_{ov} represents the total communication overhead of adaptation for scheduled applications. We further observe the polling delay as a trade-off between the communication overhead and the timing granularity for adaptation.

Energy Saving: the goal of hierarchical adaptation in HATMA for low-power is to dynamically reduce the system's energy consumption when executing a given application without detriment to system performance. We evaluate the energy-saving capability of HATMA as the runtime percentage decrease *idle time* in applications execution time compared with the base schedule computed offline. Furthermore, HATMA exploits the system idle times to facilitate power/clock gating, reducing energy consumption. We show the reduction in energy consumption of a synthetic application with 20 tasks and messages and the communication overhead of adaptation due to the timing granularity of HICP in the range of 100 - 500µs.

In Figure 19, a key point ≈180µs highlights an optimal point for energy consumption and communication overhead. A shift to the right of this point, although resulting in lower communication overhead, leads to reduced energy saving and vice versa for a change to the left. In scheduling HATMA's adaptation, a trade-off is made between the adaptation overhead and energy-saving constraints of the application.

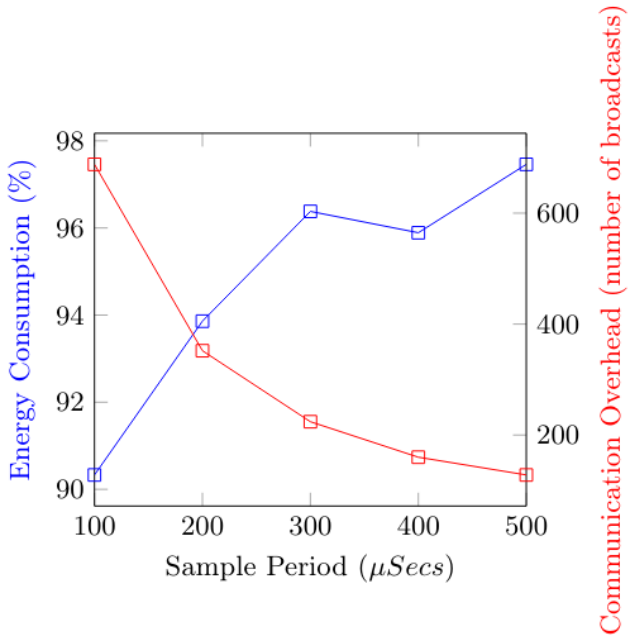

Figure 19 Energy Saving and Communication Overhead for Synthetic Application

Table 3 KPI and metric of component WP4T41-02

KPI description	Means of assurance	Expected	Measured/ Achieved
Adaptation to predefined scenarios	Simulation	<1ms	900µS
Adaptation for energy saving	Simulation	>5%	6%

6.3.1 Integration in UC8

A hierarchical automated warehouse shuttle system, UC8, is implemented as a SWARM intelligent system, utilizing the adaptability service of HATMA to improve dependability. HATMA ensures the FRACTAL node-based shuttle system adapts to new tasks and failures in the system, ensuring tasks are completed even in crash scenarios (e.g., failure of a shuttle, failure of a lift, failure of a track). HATMA also facilitates adaptation to avoid obstacles by adjusting routing paths and enabling strategies to clear them if possible.

7 Low-Power Services for PULP Systems - WP4T41-03 - ETH

7.1 Component description

PULP (Parallel Ultra-Low-Power) is an open-source computing platform targeting IoT applications. To cope with the tight power constraints that small battery-powered IoT devices need to meet, the PULP architecture is designed around low-power operations. Further low-power services have been developed at ETH to meet the requirements as defined by the UCs.

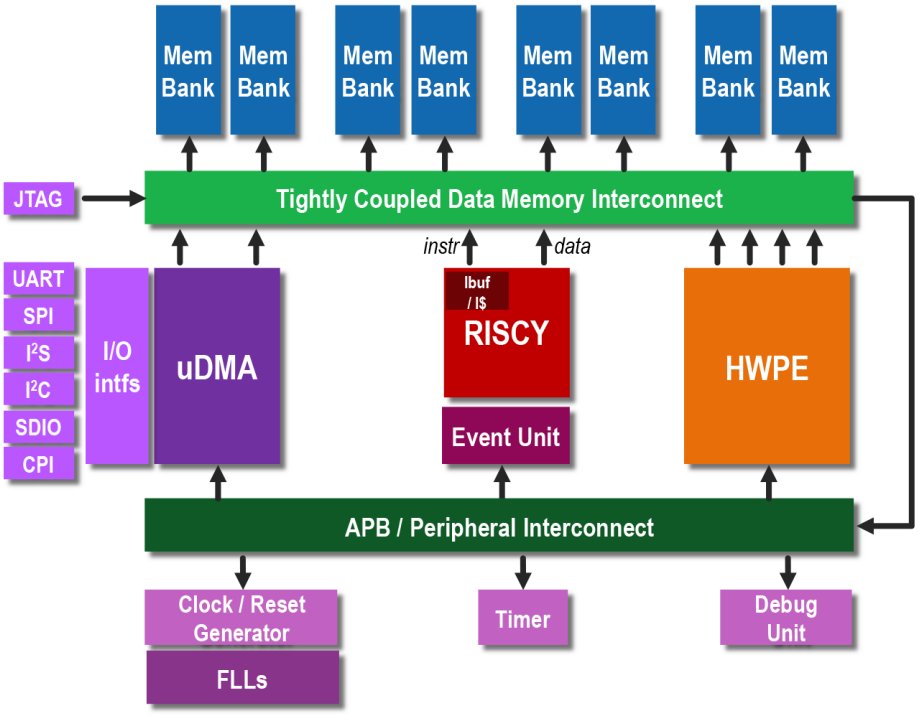

Figure 20 PULPissimo SoC Schematic

7.2 Design and implementation

PULPissimo, introduced in deliverable D2.1, provides a full microcontroller architecture containing a single RISC-V core, a low-latency multi-bank scratchpad memory, a set of peripherals, and a direct memory access (DMA) engine taking care of autonomous I/O, advanced data pre-processing, and external interrupts.

The main three phases of the workloads faced by an IoT device are sensing, processing, and transmission. As the transmission phase requires more power than processing data on such a low-power microcontroller, the goal of PULP-based systems is to maximize the processing performed on the edge to minimize the time spent in transmission. To increase the energy efficiency of the overall system and achieve

higher performance, various paths have been investigated: (i) coupling PULPissimo with a multicore compute cluster, (ii) extending PULPissimo with domain-specific hardware accelerators, or (iii) enhancing PULPissimo's RISC-V core with instruction set architecture extensions.

To minimize the power consumption of the IoT device, additional components introduced to increase the energy efficiency of the system can be placed in different power domains, which can be powered down during sleep mode. We targeted power budgets typical of microcontroller systems (<100mW), and explored fine-grain clock and power-gating techniques that can be employed to fine-tune the architecture configuration to specific application phases following a sub-10ns power-up sequence.

7.3 Testing and evaluation

We prototyped PULPissimo-based platforms on FPGA to evaluate the benefits of the extensions and customization. Deployment scripts for various FPGA platforms (Digilent Genesys2, Xilinx VCU108, Xilinx ZCU102, ZedBoard, ...) are open-source on the PULPissimo GitHub page (https://github.com/pulp-platform/pulpissimo). Partners interested in prototyping their PULPissimo-based use case can use such scripts to speed up the testing process. Furthermore, we taped out and tested Echoes, a low-power PULPissimo-based chip enhanced with a domain-specific hardware accelerator.

Figure 21 Echoes Chip - A Low-Power PULPissimo-based Chip

7.3.1 Integration into UC3

A PULP-based architecture will be used in UC3 (Smart meters for everyone), where a smart meter prototype will be designed. A PULP-based IoT system will be connected to a camera to take a picture of the display of a mechanical meter, process it to extract the information displayed by the meter, and finally send the data over the

cellular network. The low-power features provided by PULP will allow achieving a long battery life.

Table 4 KPI and metric of component WP4T41-03

KPI description	Means of assurance	Expected	Measured/ Achieved
Power reduction through clock and power gating	Lab results	> 35%	42% reduction in the idle power
Additional power reduction introduced by fine-grain power gating with respect to clock-gating-only	Lab results	> 10%	12.7% additional reduction in the idle power

8 Versal RPU Access for Power Services - WP4T41-04 – PLC2

8.1 Component description

As described in Section 10.1 the isolated Versal ACAP platform as designed would block access to the device infrastructure to protect core functionality from any access fault or attack. To still enable system state changes on purpose some access mechanism has to be provided.

To accomplish this, this component creates an RPU project that exposes a defined subset of Versal / VCK190 power and observability features through an interface. The communication interface between APU and RPU subsystems is selected as OpenAMP.

This component shall provide means that the application running on APU can request support for power settings, local power and temperature monitoring data from RPU node via an OpenAMP channel. RPU shall handle the requests in a safety focused manner as this RPU core is the target of a safety zone in the certifiable platform version. RPU image shall be included in the main boot image and shall be loaded via PMC to comply with safety regulations.

8.2 Design and implementation

This work package will provide a system with APU cores running SMP Linux OS (PetaLinux) and RPU running FreeRTOS.

RPU application has two main roles which are handling OpenAMP communication and interfacing with hardware through PMC. In OpenAMP communication, RPU plays the slave role which is receiving messages from APU and reacts by checking the command embedded in the message protocol. In RPU application development, Vitis Unified Software Platform 2021.2 tool has been used as the software development kit.

APU core is running PetaLinux which is supporting OpenAMP. The device-tree and Linux Kernel configuration of the PetaLinux project has been adapted properly to support OpenAMP and the RPMsg channel between RPU and APU. This channel between APU and RPU supports maximum 512 bytes message including header and payload. Therefore, the messages defined in the protocol cannot be larger than this limit.

In Figure 22, the message protocol is described. The message protocol between APU and RPU shall have three components which are data, the size of the data and the command. The command is a definition of the request from APU to RPU. Data is the result of the action performed on the RPU for a specific command.

```
struct payload {
    uint32_t cmd;
    uint32_t size;
    char data[];
};
```

Figure 22 PLC2 OpenAMP payload struct

In Figure 23, the command types for the message protocol are defined. This enumerator can be further extended with the new commands required.

```
typedef enum {
    READ_TEMP_MIN = 0,
    READ_TEMP_MAX,
    READ_VCC_INTPL,
    READ_VCC_RAM,
    READ_VCC_SOC,
    READ_VCCBATT,
    READ_VCCPMC,
    READ_VCCPSFP,
    READ_VCCPSLP
} CommandType;
```

Figure 23 PLC2 OpenAMP command list

8.3 Testing and evaluation

System level functional testing can be performed in this platform related component. It shall contain following steps to proof the functionality. For testing purposes, the device-tree supports RemoteProc which enables developers to update RPU images at run-time to enhance the development cycle. RemoteProc support will be disabled in the final design and RPU image shall be inside the main boot image as it is described in the component description.

- APU shall load executable on RPU core via RemoteProc,
- RPU application shall initialize OpenAMP and wait for message,
- APU shall initialize OpenAMP and send a message for reading local voltage or temperature,
- RPU shall receive and parse the message, read the corresponding sensor data and send back as payload via RPMsg channel,
- APU shall display the message on the standard output.

Table 5 KPI and metric of component WP4T41-04

KPI description	Means of assurance	Expected	Measured/ Achieved
Versal based node infrastructure monitoring	Development Kit testcases	Retrieve local temperatures and current and voltage values	Achieved
Versal based node infrastructure control	Development Kit testcases	Control and scale power consumption	Achieved, with restrictions due to dev kit.

9 Agreement Protocol for Low-Power Services – WP4T41-05 - QUA

9.1 Component description

In the deliverable D4.1 the principles of the Agreement Protocol were explained. The main objective for this protocol is to have a clock synchronization among all the devices connected in the network achieved wirelessly. This clock synchronization would lead to all the devices in the network, having the same clock reading/value. Which would be essential in case of measurement of sensor data (e.g. acceleration, gyroscope) or keeping devices in a configuration in sync with the other devices in the network.

This synchronization is performed by selecting a master which will share its clock with all the devices in the network. The master device in this case is a low power microcontroller. The process of selecting a master starts when all the devices share a specific value, in this case their clock value. Once all devices have agreed on a value, the device with a specified value will be selected as master, and start the synchronization from the slaves in the network.

The Agreement Protocol must detect and correct when slaves or even a master with a faulty clock is in the network. If a faulty clock of a slave is detected, the master will restart the procedure to synchronize the faulty slave clock. In case the master is lost or present a faulty clock, the selection of a new master will be initiated.

One of the biggest challenges to implement an Agreement Protocol is the latency. The greater the latency a greater difference between the master and the slave will be present. In the case for wireless communications, latency is affected to even more with factors such as interference distance and signal strength. Even though with these challenges, a wireless communication provides such a greater benefit which is the mobility of a device or setting a system where cabling might be difficult to route. To this advantage, add a low power device and the capability to obtain data or control systems, provides a versatility that can simplify multiple engineering fields.

9.2 Design and implementation

For the implementation of the Agreement Protocol on a wireless network the ESP32-WROOM microcontroller from Espressif was selected. This microcontroller can be programmed in C or C++, it also counts with a two-core processor that can be individually controlled, enabling this microcontroller to run FreeRTOS. It can also provide Wi-Fi, Bluetooth and Bluetooth Low Energy (BLE) for wireless connectivity. These two key features allow us to use Espressif wireless communication ESP-NOW. This protocol is based on the Data Link layer providing a low latency connection with the devices connected to this network.

The availability of having a multicore processor and FreeRTOS implies that it is possible to control the wireless communications while performing additional processes without interference between tasks. To communicate the devices among them, they require only the MAC address of the ESP32 peers that be in the network. This enables a direct communication between devices.

9.2.1 Master selection

To select the master of the network each device will start their boot-up sequence and will record that boot-time using their internal clock. Once the boot-up sequence is finalized, each device on the network will broadcast their boot-up time with their MAC address. For this a structure is defined as shown in Figure 24.

```
typedef struct payload
{
  int64_t time;
  STATE state;
}payload_t;

typedef struct payload_ext{
  payload_t payload;
  uint8_t mac_add[6];
}payload_ext_t;
```

Figure 24 QUA Data structure for synchronization

To address the Y2K38 problem and to have an accurate time value 64-bit Integer is used. We can get the system time from the ESP32 microcontroller using the function *gettimeofday()*. We can also set the system time using the function *settimeofday()* as shown in Figure 25**.** These functions were used to get and set time in the peer ESP32.

```
struct timeval tv_now;
gettimeofday(&tv_now, NULL);

int64_t time_us = (int64_t)tv_now.tv_sec * 1000000L
+ (int64_t)tv_now.tv_usec;
```

Figure 25 QUA gettimeofday() command to obtain the clock value of an ESP32 in values of microseconds

While receiving the boot-times and MAC addresses of the peers in the network, each device will create a list for all the devices that will be present in the network. Once compared and verified the lowest boot time from all the devices the master is selected. With a master defined, the process of synchronizing the clocks of the slaves is started.

9.2.2 Synchronization of slaves

Once the master is selected from the group of nodes (ESP-32 microcontrollers), it will start sending SNYC messages to the peers. The synchronization protocol is based

on a PTP-like synchronization method. The master would send a SYNC message requesting the peer/slave to start syncing the internal clock. This is followed up by a FOLLLOW_UP message from the master. Then the peer would send a DELAY_REQUEST message, the master would reply to this message by DELAY_RESPONCE message. This interaction is visualized in Figure 26. The peer/slave would now adjust its own clock by calculating the offset in time and then adjusting its own clock with the offset. This whole synchronization method is repeated regularly to course correct the slave if the clock has moved more than the maximum offset (MAX_OFFSET). The payload used for the message is shown in Figure 24.

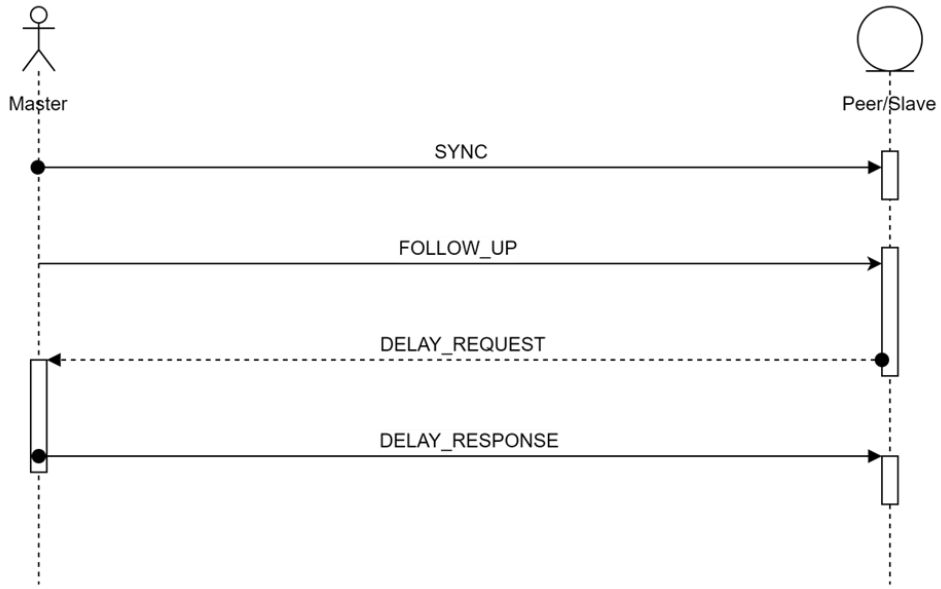

Figure 26 QUA message interaction between master and slaves

9.3 Testing and evaluation

Agreement protocol for low-powered devices provides a method to synchronize the clocks of all the peers with a master device in a network. Since the system clock (software timer) is updated internally with each SYNC message, not the RTC (Real Time Clock), it becomes hard to test the time offset externally. This led us to create a test bench to observe the offset time using two different approaches, one using a GPIO (General Purpose Input Output) pin and the other using messages to calculate the offset time in a salve.

GPIO Triggering

In this method we wanted to create an external response which is cyclic in nature. The GPIO trigger would happen every second (or millisecond) on the master and peer device. If the clocks are synchronized among the peers, the trigger event would occur

at the same time for each of the devices. Later with the help of an oscilloscope we can measure the offset time externally among the peers and the master.

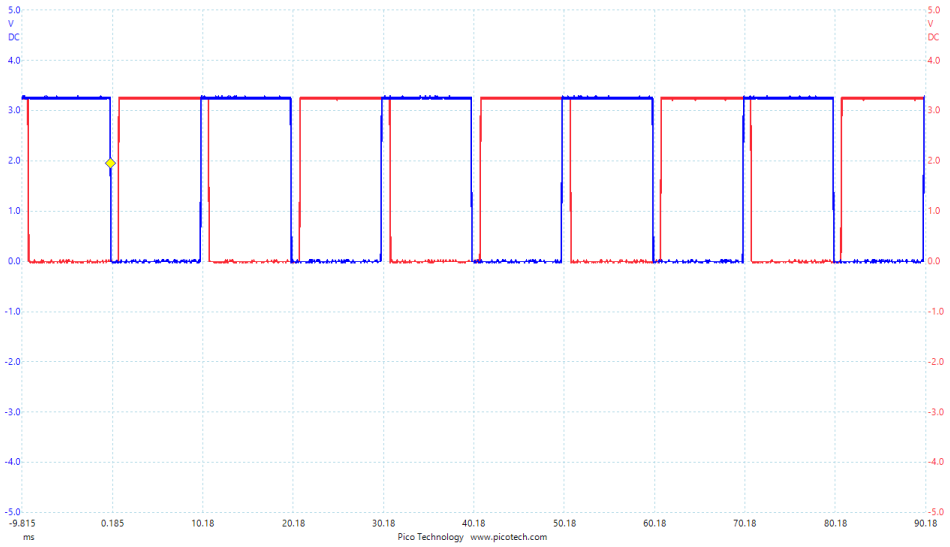

Figure 27 An example of clock synchronization on GPIO

Since the POSIX function *settimeofday()* was used to update the clock value, it was only updated on the software level instead of the RTC clock. Due to this we cannot observe the changes in clock values in peer and master, as the GPIO trigger works with the clock value (delay) from FreeRTOS, which in turn depends on the RTC clock of the hardware. Due to this, the offset time were not observed for the software timers.

Figure 27 shows a demonstration of expected behavior with GPIO signals from a master (red) and a slave (blue). If the clocks were synchronized, both the signal would either overlap or be close to each other. Since it is difficult to observe the software timing externally, Figure 27 shows the internal clock behavior instead.

Offset estimation with messages

Another approach which was used to evaluate the offset time was calculation of offset value in a peer device. Since we are sending our clock value with each message, it is easy to save the time values locally and use them later to calculate the offset value for a peer. The formula is used to calculate the offset time of a peer with the master's clock. The lower the value for the offset time the closer it is with the master's clock.

$$delay = \frac{\big((syncTime - followupTime) - (delayResponseTime - delayRequestTime)\big)}{2}$$

$$offset = (syncTime - followupTime) - delay$$

We can monitor this value to observe for changes, if the offset time has increased for the peer, we would update the clock of the peer to accommodate it for the new offset

value. Our goal was to have the clock offset value of the peer be under 100 µs. With this method we can track the timings of peers easily.

Table 6 KPI and metric of component WP4T41-05

KPI description	Means of assurance	Expected	Measured/ Achieved
Synchronization Achieved: The value of synchronization between the clocks of peers and master	Lab test	Having the clock of the peer be in sync all the time with a value lower than 100 µs.	The clocks drifted in value after the subsequent message. It adjusts itself when the clock drift increases more than 100 µs
Number of peers: The maximum number of peers that could be connected to the network	Lab test	The network allows a total number of 20 peers for one master	Performance of the synchronization drops after 4 peers with one master
Clock drift per hour: Number of times the clock drifted in one hour, when the SYNC message is being sent every minute	Lab test	Drift should be 15 times in one hour	It was observed that our method produced a clock drift of more than 100µs 20 times in an hour

10 Versal Isolation Design- Functional Safety - WP4T41-06 - PLC2

10.1 Component description

The basic description of the platform reference design for Versal based FRACTAL nodes as described in deliverable D4.1 shall be used in a commercial / industrial environment. In some scenarios with safety concerns access separation needs to be put in place to comply to the definitions of deliverable D2.2. This component details the design choices in the hardware setup of the Versal ACAP device in the VCK190 development kit to achieve such basic isolation design. With subsequent maturity of the Versal development eco system the supplier AMD-Xilinx will publish formally tested and thus certifiable design setups. At the time of this report such official releases are not available. The current state of this component reflects the understanding of the approach that will become available in due time but some detail changes may apply.

To allow controlled access to the data and controls available in the isolation-based platform, a proxy setup is required to accompany this component. This is established by the WP4T41-04 RPU component, that is exposing the internals of the safety channel features as defined in this component.

10.2 Design and implementation

As commonly used for various AMD-Xilinx device technologies, the basic hardware setup of Versal Designs is created in Vivado. Specific separation settings are configured in the specifications of the hard- IP block CIPS and are given here.

10.2.1 Hardware Block Design and Settings

The overall setup of the hardened IP of the Versal ACAP devices allows for protection units to control and filter the transactions hitting a specified address slice. Depending on the target wrapped by these units, there is a difference in control granularity supported by these units. A more block granular set of units typically connects to memory Xilinx Memory Protection Units (XPMU) whereas a finer control is exerted for peripherals with the Xilinx Peripheral Protection Units (XPPU). For this component the following protection units are applied and setup to split between APU and RPU transactions.

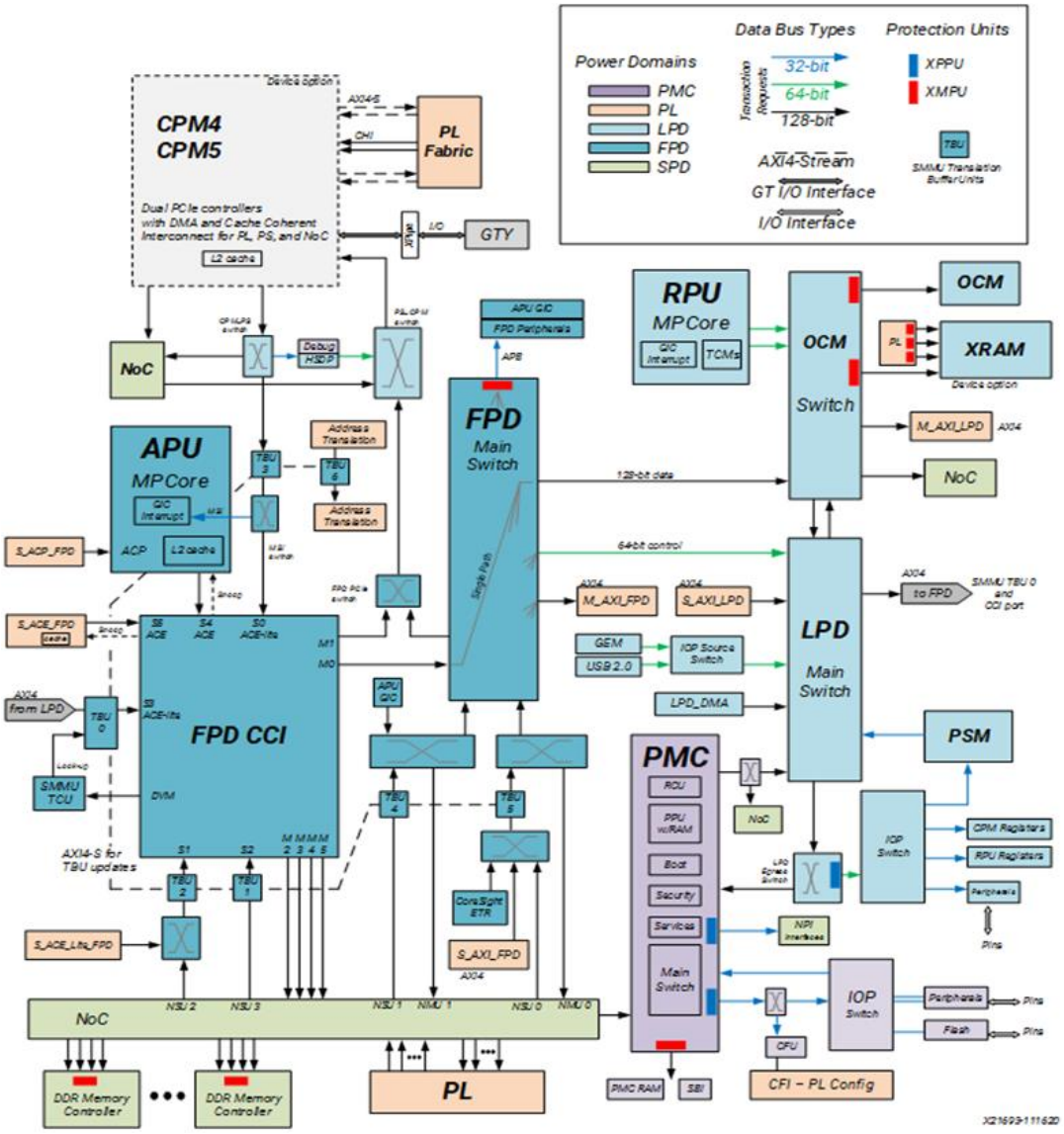

Figure 28 Location of protection units in Versal CIPS

The protection units are set to block transactions from APU side for the respective address blocks while allowing RPU side accesses to propagate. To allow this there are multiple registers that need to be configured appropriately and finally arrive at a separated address map. This configuration is defined as a base scenario but minor adaption to the specific requirements of the use cases will be visited.

On top for Versal ACAP based safety centric designs the isolation of address blocks through the network on chip (NoC) typically need to be considered. Within the generation of this component this has been visited to create a suitable mapping of NoC paths as exclusive or defined by a grouping of lanes. Final choice was to skip this from the isolation setup as for the use cases under consideration the NoC based components are not safety related.

Using these units with configurations according to the desired isolation properties yields an address map that reflects the separation as seen here.

Table 7 CIPS isolation address map and peripherals based on XPPU and XPM

APU Subsystem	**Configured Base Address**	**Size**	SW = Secure world NSW = non-secure world	**Access**
A72			**NSW**	
OCM	0xFFFF_0000	64 KB	NSW	R/W
DDR_LOW	0x0000_0000	32 MB	NSW	R/W
DDR_LOW	0x6000_0000	1 MB	NSW	R/W
UART0			NSW	R/W
GPIO			NSW	R/W
SWDT0			NSW	R/W
TTC0			NW	R/W
RPU Subsystem	**Configured Base Address**	**Size**	SW = Secure world NSW = non-secure world	**Access**
RPU			**SW**	
OCM	0xFFFC_0000	192 KB	SW	R/W
OCM	0xFFFF_0000	64 KB	SW	R/W
DDR_LOW	0x4000_0000	16 MB	SW	R/W
DDR_LOW	0x6000_0000	1 MB	SW	R/W
GPIO			SW	R/W
I2C1			SW	R/W
UART1			SW	R/W

10.2.2 **Software Platform**

To allow the binding of the hardware setup into the application layer on APUs is carried out by deploying ARM Trustzone technology, so that the APU and its memory and peripherals are TZ non-secure while the RPU and PMU along with their dedicated memory and peripherals are TZ secure. This component is supported by correct setup of the software platform that is set up to drive this hardware setup.

With this platforming place, a minimum executable can be delivered on the bare metal as well as on the Linux domain to run and prove the effectiveness of this hardware definition.

10.3 Testing and evaluation

The implementation with the low-profile applications provides a bootable PDI file to ensure the ramp of the multiprocessor system in the defined isolated setup. The core access to the low-level infrastructure is blocked from APU / application level and the functionality can be reached only through the indirection through the PMC. A proper testcase would follow through:

- Show access is possible to the low-level services and peripherals from APU and RPU in a non-isolation-based platform, i.e., before the platform availability.
- Show access is only available for the respective component that is allowed access through the separation setup, prove blocked access otherwise
- Identify that in the final setup the delivery version will not allow APU access to the PMC level.

Table 8 KPI and metric of component WP4T41-06

KPI description	Means of assurance	Expected	Measured/ Achieved
Consistently block access to predefined components	Development board tests	PMC functions cannot be reached from APU	Achieved

10.3.1 **Integration into UC8**

Current plan is to ramp UC8 application-level software on the overall Fractal Versal Reference Design for safety related systems, specifically on the APUs. All environmental awareness is provided through the RPU proxy into the isolated domain.

The dynamic scheduling of the UC8 Fractal nodes is also deploying this communication path to even control the local power level if the schedule planning requires.

11 Validation of LEDEL library for low-Power Services - SML

11.1 Introduction

This is the documentation of the demonstration on the use of the EDDL library and the execution of code which use said library in RISC-V systems, more precisely, in an emulation of a NOEL-V hardware system. Following the instructions here, one should be able to compile C++ code using the functions of the EDDL library, with the only dependency being the installation in your local machine of Docker and a Docker Image.

We have explained how we have approached the creation, infrastructure and first simple example of the LEDEL in deliverable D3.6 that belongs to WP3. A very similar procedure will be followed in this document with different examples. First, In the first docker container the EDDL library and all the tools needed to compile and run C and C++ code have already been installed. The idea of this container is that it can be perceived as the FRACTAL node since we will work with it in a similar manner.

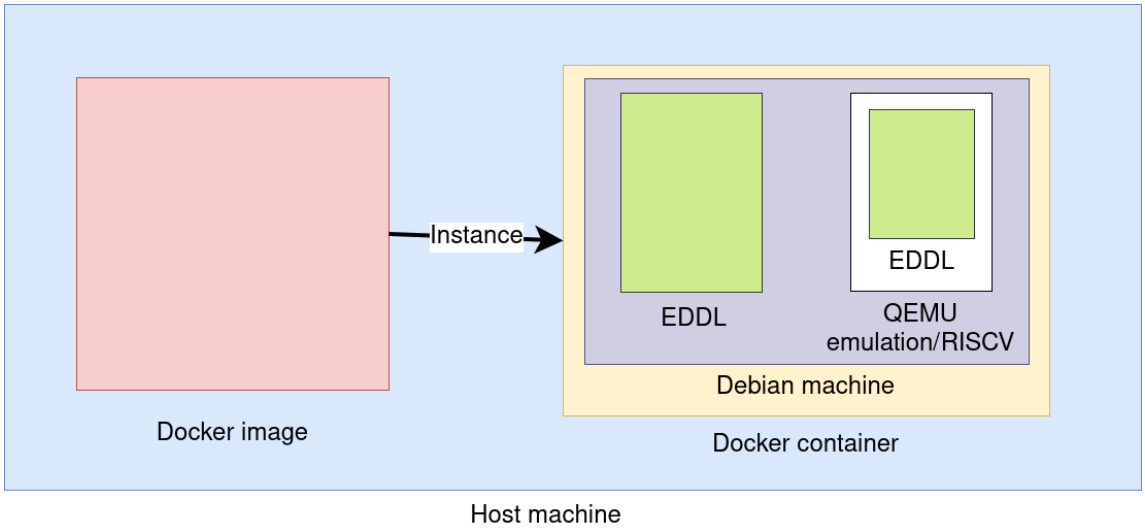

Figure 29 Docker diagram

In Figure 29 diagram of the docker container is presented. The docker containers enclose a Debian Linux Machine where the EDDL is installed. Next to it, we can find the QEMU emulation RISC-V SIEMENS/ISAR [2]76, which emulates a Linux SO running in an emulated NOELV RISC-V based machine HW.

All the documentation, files, containers and examples are uploaded to the repository of the FRACTAL project [3].

One entry of the repository is the manual of the use of the LEDEL and, as well, two demonstration videos.

The tests and examples used to validate the use of LEDEL as a low power service in a FRACTAL node are organized as follows. First, we explain how to compile and generate an ONNX file with the description of the machine learning algorithm topology, inside and outside of the FRACTAL node. Second, the ONNX generated in the previous step is imported inside the FRACTAL node and the process of inference is shown. Third, a couple of neural network training algorithm implemented using PyTorch and TensorFlow that create other ONNX files that are loaded inside the node for the inference process. Fourth example is the cross-compilation process of the first program. Finally, a use case from the Deep Health project with a reduced dataset is executed.

All the examples are executed using the first docker container, while the cross-compiled example has an own docker container with its required system configuration.

11.2 EDDL code compilation process

To start with the testing process, we focus on the first case, a simple neural network that uses the CIFAR10 dataset [4].

To be complete and thorough with the tests, we have compiled the same program out of the FRACTAL node, to show that the mechanism used to load ONNX files works, and also inside the node, to prove that the LEDEL works properly.

For the demonstration of this process, we have included some code examples inside the docker container that shows the training of a convolutional network over the Cifar10 dataset. The same example will be used in the second part of this section to show the complete compilation process.

The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, so the trained neural network needs to classify each image as one of the 10 possible classes, these classes being ‘airplane’, ‘truck’, ‘cat’, ‘horse', 'automobile’, ‘dog’, ‘deer’, ‘bird’, ‘frog’ and ‘ship’.

11.2.1 Code compilation for training and ONNX file creation

We are going to start by compiling a simple program. To this aim, only two files are required and need to be placed in the same folder of choice: a program with a main function implemented in C++ and using the EDDL [Figure 30 and Figure 31, and a CMakeLists.txt file with instructions to link the EDDL library and configure the executable that will be generated [Figure 32]:

```cpp
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include "eddl/apis/eddl.h"
#include "eddl/serialization/onnx/eddl_onnx.h"


using namespace eddl;

///////////////////////////////////
// cifar_conv_da.cpp:
// A very basic Conv2D for cifar10
// Data augmentation
// Using fit for training
///////////////////////////////////

int main(int argc, char **argv){

    // download CIFAR data
    download_cifar10();

    // Settings
    int epochs = 5;
    int batch_size = 100;
    int num_classes = 10;

    // network
    layer in=Input({3,32,32});
    layer l=in;

    // Data augmentation
//    l = RandomShift(l, {-0.2f, +0.2f}, {-0.2f, +0.2f});
//    l = RandomRotation(l, {-30.0f, +30.0f});
//    l = RandomScale(l, {0.85f, 2.0f});
//    l = RandomFlip(l, 1);
//    l = RandomCrop(l, {28, 28});
//    l = RandomCropScale(l, {0.f, 1.0f});
//    l = RandomCutout(l, {0.0f, 0.3f}, {0.0f, 0.3f});

    // l=Select(l, {"1", "1:31", "1:31"});
    l=MaxPool2D(ReLu(Conv2D(l,32,{3,3},{1,1})),{2,2});
    l=MaxPool2D(ReLu(Conv2D(l,64,{3,3},{1,1})),{2,2});
    l=MaxPool2D(ReLu(Conv2D(l,128,{3,3},{1,1})),{2,2});
    l=MaxPool2D(ReLu(Conv2D(l,256,{3,3},{1,1})),{2,2});

    l=Reshape(l,{-1});
```

Figure 30 C++ program using EDDL, part 1

```
    l=Activation(Dense(l,128),"relu");

    layer out=Activation(Dense(l,num_classes),"softmax");

    // net define input and output layers list
    model net=Model({in},{out});


    // Build model
    build(net,
          sgd(0.01, 0.9), // Optimizer
          {"soft_cross_entropy"}, // Losses
          {"categorical_accuracy"}, // Metrics
          //CS_GPU({1}) // one GPU
          //CS_GPU({1,1},100) // two GPU with weight sync every 100 batches
          CS_CPU()
    );


    // plot the model
    plot(net,"model.pdf");

    // get some info from the network
    summary(net);

    // Load and preprocess training data
    Tensor* x_train = Tensor::load("cifar_trX.bin");
    Tensor* y_train = Tensor::load("cifar_trY.bin");
    x_train->div_(255.0f);

    // Load and preprocess test data
    Tensor* x_test = Tensor::load("cifar_tsX.bin");
    Tensor* y_test = Tensor::load("cifar_tsY.bin");
    x_test->div_(255.0f);

    // training, list of input and output tensors, batch, epochs
    fit(net,{x_train},{y_train},batch_size, epochs);

    // saving net architecture and weights in a ONNX format file
    save_net_to_onnx_file(net, "cifar10_eddl_net.onnx");
}
```

Figure 31 C++ program using EDDL, part 2

```
cmake_minimum_required(VERSION 3.9.2)
project(cifar10_eddl_train)

add_executable(cifar10_eddl_train main.cpp)

find_package(EDDL REQUIRED)
target_link_libraries(cifar10_eddl_train PUBLIC EDDL::eddl)
```

Figure 32 Configuration file for CMAKE

To compile the program, simply execute these commands within the same directory where the two files are placed. To make things neater, it is recommended to create a folder where the compiled files will be created and saved, and then, follow with the compilation command. For instance:

```
mkdir build
cd build/
cmake ..
make
```

Figure 33 Recommended instructions

In the directory named 'build' we find the compiled file, which can be directly executed and the training process of the network will automatically start, "./cifar10_eddl_train". In Figure 34 we can see the results of command for the program "./cifar10_eddl_train":

```
Downloading cifar
cifar_trX.bin ✓
cifar_trY.bin ✓
cifar_tsX.bin ✓
cifar_tsY.bin ✓
No axis for 'softmax' was specified. Using last one (-1) as default (LActivation::Softmax)
Generating Random Table
CS with full memory setup
Building model
sh: 1: dot: not found
[PLOT] Unable to run the following command:
        => dot -T pdf ./tmp.dot >./model.pdf
____________________________________________________________________________
model
____________________________________________________________________________
input1    |  (3, 32, 32)         =>   (3, 32, 32)         0
conv2d1   |  (3, 32, 32)         =>   (32, 32, 32)        896
relu1     |  (32, 32, 32)        =>   (32, 32, 32)        0
maxpool2d2|  (32, 32, 32)        =>   (32, 16, 16)        0
conv2d2   |  (32, 16, 16)        =>   (64, 16, 16)        18496
relu2     |  (64, 16, 16)        =>   (64, 16, 16)        0
maxpool2d4|  (64, 16, 16)        =>   (64, 8, 8)          0
conv2d3   |  (64, 8, 8)          =>   (128, 8, 8)         73856
relu3     |  (128, 8, 8)         =>   (128, 8, 8)         0
maxpool2d6|  (128, 8, 8)         =>   (128, 4, 4)         0
conv2d4   |  (128, 4, 4)         =>   (256, 4, 4)         295168
relu4     |  (256, 4, 4)         =>   (256, 4, 4)         0
maxpool2d8|  (256, 4, 4)         =>   (256, 2, 2)         0
reshape1  |  (256, 2, 2)         =>   (1024)              0
dense1    |  (1024)              =>   (128)               131200
relu5     |  (128)               =>   (128)               0
dense2    |  (128)               =>   (10)                1290
softmax6  |  (10)                =>   (10)                0
____________________________________________________________________________
Total params: 520906
Trainable params: 520906
Non-trainable params: 0

5 epochs of 500 batches of size 100
Epoch 1
[                                                  ] 500 softmax6[loss=1.836 metric=0.323] 0.2439 secs/batch
121.9730 secs/epoch
Epoch 2
[                                                  ] 500 softmax6[loss=1.344 metric=0.517] 0.2299 secs/batch
114.9400 secs/epoch
Epoch 3
[                                                  ] 500 softmax6[loss=1.081 metric=0.617] 0.2417 secs/batch
120.8744 secs/epoch
Epoch 4
[                                                  ] 500 softmax6[loss=0.899 metric=0.687] 0.2276 secs/batch
113.7806 secs/epoch
Epoch 5
[                                                  ] 500 softmax6[loss=0.772 metric=0.731] 0.2313 secs/batch
115.6503 secs/epoch
root@3ad629b5a243:/isar-riscv/eddl_examples/cifar10_eddl_train/build#
```

Figure 34 Training trace and results for CIFAR10 example.

Once the training process is finished, we can find the ONNX file in the same 'build' folder. This ONNX file contains the network architecture and the trained weights. From here, we can import the ONNX file to the FRACTAL node. This process will be depicted in the following section.

11.2.2 Example with RISC-V

In order to complete the demonstration of the full functionality of the LEDEL we have prepared another example that shows its behavior inside the FRACTAL node. For the purpose of this task, we have used the same code as in the previous example and its training execution can be seen in Figure 35.

```
riscv@localhost:~/cifar10_examples/cifar10_eddl_train/build$ ./cifar10_eddl_train
Downloading cifar
cifar_trX.bin x
cifar_trX.bin       100%[===================>] 585.94M  8.89MB/s    in 72s
cifar_trY.bin x
cifar_trY.bin       100%[===================>]   1.91M  5.39MB/s    in 0.4s
cifar_tsX.bin x
cifar_tsX.bin       100%[===================>] 117.19M  7.38MB/s    in 17s
cifar_tsY.bin x
cifar_tsY.bin       100%[===================>] 390.64K  --.-KB/s    in 0.08s
No axis for 'softmax' was specified. Using last one (-1) as default (LActivation::Softmax)
Generating Random Table
CS with full memory setup
Building model
sh: 1: dot: not found
[PLOT] Unable to run the following command:
        => dot -T pdf ./tmp.dot >./model.pdf
--------------------------------------------------------------------------------
model
--------------------------------------------------------------------------------
input1     |  (3, 32, 32)          =>   (3, 32, 32)          0
conv2d1    |  (3, 32, 32)          =>   (32, 32, 32)         896
relu1      |  (32, 32, 32)         =>   (32, 32, 32)         0
maxpool2d2|  (32, 32, 32)         =>   (32, 16, 16)         0
conv2d2    |  (32, 16, 16)         =>   (64, 16, 16)         18496
relu2      |  (64, 16, 16)         =>   (64, 16, 16)         0
maxpool2d4|  (64, 16, 16)         =>   (64, 8, 8)           0
conv2d3    |  (64, 8, 8)           =>   (128, 8, 8)          73856
relu3      |  (128, 8, 8)          =>   (128, 8, 8)          0
maxpool2d6|  (128, 8, 8)          =>   (128, 4, 4)          0
conv2d4    |  (128, 4, 4)          =>   (256, 4, 4)          295168
relu4      |  (256, 4, 4)          =>   (256, 4, 4)          0
maxpool2d8|  (256, 4, 4)          =>   (256, 2, 2)          0
reshape1   |  (256, 2, 2)          =>   (1024)               0
dense1     |  (1024)               =>   (128)                131200
relu5      |  (128)                =>   (128)                0
dense2     |  (128)                =>   (10)                 1290
softmax6   |  (10)                 =>   (10)                 0
--------------------------------------------------------------------------------
Total params: 520906
Trainable params: 520906
Non-trainable params: 0

1 epochs of 500 batches of size 100
Epoch 1
[                                                  ] 500 softmax6[loss=1.913 metric=0.292] 13.3136 secs/batch
6656.8113 secs/epoch
riscv@localhost:~/cifar10_examples/cifar10_eddl_train/build$
```

Figure 35 Cifar-10 training execution

11.2.3 Loading ONNX and inference process in the FRACTAL node

Up to this point, if the steps have been followed, we have created an ONNX file that includes not only the topology of the neural network, but the weights of the training process as well. We are ready for the next step.

Now is the moment to import the ONNX file into the RISC-V machine or FRACTAL node to proceed with the process of inference. We point out that the process this time is a lot slower due to the emulation process. A similar situation might happen with real NOEL-V/RISC-V hardware due to its low computing power. From this, it is easy to understand why the training process is smarter to be executed in the cloud or in a designated hardware more powerful, to create the ONNX file and then load it into the FRACTAL node. Thus, it is more efficient and interesting to perform only the inference part of the machine learning process by using the strategy of importing the ONNX format file to place the neural network and/or the weights. The node can send afterwards the data collected or created back to the cloud if needed.

Consequently, we can understand that the training phase is a lot heavier in terms of computational time and resources and, actually, it does not need to be executed in the FRACTAL node.

Coming back to the first example of the previous section, now we proceed to import the ONNX file, that includes the network architecture and its trained weights. It is indicated in the program with the function indicated by figure 36. Consider that other languages and frameworks use a similar function with the same objective.

```
save_net_to_onnx_file(net, "cifar10_eddl_net.onnx");
```

Figure 36 Function used to create ONNX file in a program that uses LEDEL

Besides the ONNX file, we need to compile a new code for the inference of the trained network. This code will simply load the test part of the dataset, load the net from the ONNX file and start the evaluation of the network, printing the results in the console. The code we have used can be observed in Figure 37:

```cpp
#include <cstdio>
#include <cstdlib>
#include <iostream>

#include "eddl/apis/eddl.h"
#include "eddl/serialization/onnx/eddl_onnx.h"

using namespace eddl;


int main(int argc, char **argv) {

    // download CIFAR data
    download_cifar10();

    // Load and preprocess test data
    Tensor* x_test = Tensor::load("cifar_tsX.bin");
    Tensor* y_test = Tensor::load("cifar_tsY.bin");
    x_test->div_(255.0f);

    // Load net
    Net* net = import_net_from_onnx_file("cifar10_eddl_net.onnx");


    // Build model
    build(net,
          sgd(0.01, 0.9), // Optimizer
          {"soft_cross_entropy"}, // Losses
          {"categorical_accuracy"}, // Metrics
          //CS_GPU({1}) // one GPU
          //CS_GPU({1,1},100) // two GPU with weight sync every 100 batches
          CS_CPU(),
          false //Disable weight initialization
    );

    // View model
    summary(net);

    // Evaluate
    evaluate(net, {x_test}, {y_test});

    std::remove("cifar_trX.bin");
    std::remove("cifar_trY.bin");
    std::remove("cifar_tsX.bin");
    std::remove("cifar_tsY.bin");
}
```

Figure 37 Inference code

In contrast to the training code shown in Figure 30 and Figure 31 we can see that in the inference code there is no need to define the network layers one by one, since they are already loaded from the ONNX. We still need to build the model with the EDDL function 'build', to assign an optimizer, a loss function and metrics. Another important difference with the training code is that in the 'build' function of the

inference code we need to specify explicitly that the weight initialization of the model is disable. Otherwise, the loaded weights from the ONNX file will be overwritten by random values and the result from the training code will be lost. To disable the weight initialization simply pass the value *False* to the parameter *init_weights* of the build function (Figure 38) [5].

```
build(net,
      sgd(0.01, 0.9), // Optimizer
      {"soft_cross_entropy"}, // Losses
      {"categorical_accuracy"}, // Metrics
      //CS_GPU({1}) // one GPU
      //CS_GPU({1,1},100) // two GPU with weight sync every 100 batches
      CS_CPU(),
      false //Disable weight initialization
);
```

Figure 38 Details of build function

The method to compile the code is identical to the one followed for the training code. Along with the code file shown in Figure 39 we will need a CMakeLists.txt file for the CMake compilation:

```
cmake_minimum_required(VERSION 3.9.2)
project(cifar10_eddl_inference)

add_executable(cifar10_eddl_inference main.cpp)

find_package(EDDL REQUIRED)
target_link_libraries(cifar10_eddl_inference PUBLIC EDDL::eddl
```

Figure 39 CMakeLists.txt file for compilation

Remember to place the inference code and the CMakeLists.txt files in the same folder of your choice and then execute the following commands within the folder created earlier for this task (like in Figure 33).

Like in the training compilation, this will generate an executable file named 'cifar10_eddl_inference' this time and inside the 'build' folder that can be found in the directory containing the code a CMakeLists.txt files we have just compiled. Executing right away the generated file will return an error, shown in Figure 40.

```
================================================================
⚠  The specified path does not exist: cifar10_eddl_net.onnx (ONNX::ImportNet) ⚠
================================================================

terminate called after throwing an instance of 'std::runtime_error'
  what():  RuntimeError: ONNX::ImportNet
Aborted (core dumped)
```

Figure 40 Error

This error is caused because the executable is trying to load the file "cifar10_eddl_net.onnx" but can't find it. To correct this simply transfer the ONNX file generated by the training code to the same folder where the executable for the inference code has been generated. Executing the program again will show the result of Figure 41, where we can see that the same model defined during the training code has been loaded and the metrics reach in training are kept.

```
root@1f6706e9a533:/home/eddl_examples/cifar10_inference_eddl/build# ./cifar10_eddl_inference
Downloading cifar
cifar_trX.bin ✓
cifar_trY.bin ✓
cifar_tsX.bin ✓
cifar_tsY.bin ✓
Generating Random Table
CS with full memory setup
Building model without initialization
--------------------------------------------------------------------------------
model
--------------------------------------------------------------------------------
input1    |  (3, 32, 32)         =>   (3, 32, 32)         0
conv2d1   |  (3, 32, 32)         =>   (32, 32, 32)        896
relu1     |  (32, 32, 32)        =>   (32, 32, 32)        0
maxpool2d2|  (32, 32, 32)        =>   (32, 16, 16)        0
conv2d2   |  (32, 16, 16)        =>   (64, 16, 16)        18496
relu2     |  (64, 16, 16)        =>   (64, 16, 16)        0
maxpool2d4|  (64, 16, 16)        =>   (64, 8, 8)          0
conv2d3   |  (64, 8, 8)          =>   (128, 8, 8)         73856
relu3     |  (128, 8, 8)         =>   (128, 8, 8)         0
maxpool2d6|  (128, 8, 8)         =>   (128, 4, 4)         0
conv2d4   |  (128, 4, 4)         =>   (256, 4, 4)         295168
relu4     |  (256, 4, 4)         =>   (256, 4, 4)         0
maxpool2d8|  (256, 4, 4)         =>   (256, 2, 2)         0
reshape1  |  (256, 2, 2)         =>   (1024)              0
dense1    |  (1024)              =>   (128)               131200
relu5     |  (128)               =>   (128)               0
dense2    |  (128)               =>   (10)                1290
softmax6  |  (10)                =>   (10)                0
--------------------------------------------------------------------------------
Total params: 520906
Trainable params: 520906
Non-trainable params: 0

Evaluate with batch size 10
[                                                  ] 1000 softmax6[loss=0.958 metric=0.670]
```

Figure 41 Inference process output

11.2.4 Importing ONNX file generated by PyTorch and TensorFlow

Not only ONNX files generated by EDDL code can be imported by the EDDL functions. Since ONNX is a standard format, any ONNX file, regarding of its method of generation, can be loaded as network by the EDDL/LEDEL. As an example of this

behavior, we have implemented a counter-part to the training code of Figure 30 and Figure 31, using PyTorch [6] [Figure 42 and Figure 43].

```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
        self.conv4 = nn.Conv2d(128, 256, 3, padding=1)
        self.fc1 = nn.Linear(256 * 2 * 2, 1024)
        self.fc2 = nn.Linear(1024, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.pool(F.relu(self.conv4(x)))
        x = torch.flatten(x, 1)  # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

if __name__ == '__main__':
    transform = transforms.Compose(
        [transforms.ToTensor(),
         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

    batch_size = 16

    trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                            download=True, transform=transform)
    trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                              shuffle=True, num_workers=2)

    testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                           download=True, transform=transform)
    testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                             shuffle=False, num_workers=2)
```

Figure 42 Cifar-10 training code using PyTorch, part 1

```
    classes = ('plane', 'car', 'bird', 'cat',
               'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

    net = Net()
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

    for epoch in range(10):  # loop over the dataset multiple times

        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            # get the inputs; data is a list of [inputs, labels]
            inputs, labels = data

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # print statistics
            running_loss += loss.item()
            if i % 2000 == 1999:  # print every 2000 mini-batches
                print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3
f}')
                running_loss = 0.0

    print('Finished Training')

    PATH = './cifar_net.pth'
    torch.save(net.state_dict(), PATH)
    dummy_input = torch.randn(16, 3, 32, 32)
    torch.onnx.export(net, dummy_input, 'cifar10_pytorch_model.onnx')
```

Figure 43 Cifar-10 training code using PyTorch, part 2

Using this code, we build a model with the same layers as in the first example, but using PyTorch framework. The model is then train with the CIFAR10 dataset, and the architecture and weights of the network are saved to ONNX. The output of this code while training is the following shown in Figure 44.

```
(venv) rgarcia@nlp:~/Desktop/Proyectos/Fractal/ejemplo_eddl/CIFAR10/cifar10_pytorch_train$ python3 cifar10_pytorch_train.py
Files already downloaded and verified
Files already downloaded and verified
[1,  2000] loss: 2.299
[2,  2000] loss: 1.811
[3,  2000] loss: 1.454
[4,  2000] loss: 1.238
[5,  2000] loss: 1.056
[6,  2000] loss: 0.911
[7,  2000] loss: 0.794
[8,  2000] loss: 0.687
[9,  2000] loss: 0.592
[10,  2000] loss: 0.520
Finished Training
```

Figure 44 Training trace using PyTorch

We can execute this code in any machine using Python. Once the training has finished, a ONNX file named 'cifar10_pytorch_model.onnx' will be generated. Then we can transfer the ONNX to a RISC-V or any other system where we have already installed the EDDL library and reuse the code from Figure 37 to test the inference of this network.

To reuse the inference code from Figure 37 we will simply need to change the line where the ONNX file is loaded so the path reaches the new ONNX from PyTorch [Figure 45]:

```
Net* net = import_net_from_onnx_file("cifar10_pytorch_model.onnx");
```

Figure 45 Function used to create ONNX file in a program that uses LEDEL

Once the change is done, we just need to re-compile the code following the same instructions as in section 11.2.1 of this document to generate the executable, transfer the ONNX to the folder where the executable has been generated and executed. If done correctly the Figure 46 should be the output shown on console:

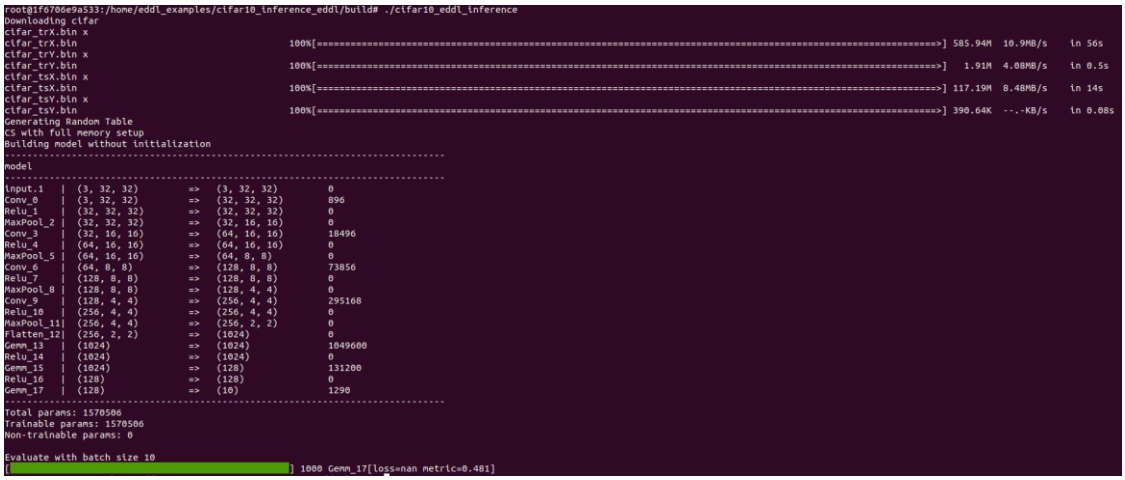

Figure 46 Output for inference process

In addition to the ONNX files generated with PyTorch, we can use other technologies like TensorFlow code to generate a model and then import it to the FRACTAL node. Unfortunately, in the case of TensorFlow ONNX files, only the structure of the model can be imported to EDDL code and not the trained weights, since EDDL still doesn't fully support this type of ONNX files.

To transfer a model from TensorFlow first we need to save a model after training and then transform that saved model to ONNX using the Python library 'tf2onnx':

```
import tensorflow as tf

from keras import datasets, layers, models

if __name__ == '__main__':

    (train_images, train_labels), (test_images, test_labels) = datasets.cifar10.
load_data()

    # Normalize pixel values to be between 0 and 1
    train_images, test_images = train_images / 255.0, test_images / 255.0

    model = models.Sequential()
    model.add(layers.Conv2D(32, (3, 3), activation='relu', padding='same', input
_shape=(32, 32, 3)))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(128, (3, 3), activation='relu', padding='same'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(256, (3, 3), activation='relu', padding='same'))
    model.add(layers.Flatten())
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(10))

    model.compile(optimizer='adam',
                  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits
=True),
                  metrics=['accuracy'])

    history = model.fit(train_images, train_labels, epochs=25, batch_size=100,
                        validation_data=(test_images, test_labels))
    tf.saved_model.save(model, "tmp_model_tf")
```

Figure 47 To test the importation of ONNX files from TensorFlow we have created, trained and saved a model using the following code

Code depicted in Figure 47 saves the resulting model in the "saved_model" format from TensorFlow. Then, to transform the "saved_model" to ONNX we execute the command from Figure 48:

```
python3 -m tf2onnx.convert --saved-model tmp_model_tf --output "cifar10_tf_mode
l.onnx"
```

Figure 48 Converting TensorFlow format model to ONNX

Indicating the path to the "saved_model" generated by the code and the path where the ONNX file will be saved to. This ONNX file can then be transferred to a system

with the LEDEL library installed, and use the inference code from the first example changing the line of code where the ONNX file is loaded.

Doing that and recompiling the code will allow us to execute the inference of the loaded ONNX from TensorFlow using EDDL functions, showing the output of Figure 49:

```
root@1f6706e9a533:/home/eddl_examples/cifar10_inference_eddl/build# ./cifar10_eddl_inference
Downloading cifar
cifar_trX.bin ✓
cifar_trY.bin ✓
cifar_tsX.bin ✓
cifar_tsY.bin ✓
Generating Random Table
CS with full memory setup
Building model without initialization
--------------------------------------------------------------------------------
model
--------------------------------------------------------------------------------
conv2d_input                                    |  (32, 32, 3)         =>   (32, 32, 3)         0
StatefulPartitionedCall_sequential_conv2d_BiasAdd__6      |  (32, 32, 3)         =>   (3, 32, 32)         0
StatefulPartitionedCall_sequential_conv2d_BiasAdd         |  (3, 32, 32)         =>   (32, 32, 32)        896
StatefulPartitionedCall_sequential_conv2d_Relu            |  (32, 32, 32)        =>   (32, 32, 32)        0
StatefulPartitionedCall_sequential_max_pooling2d_MaxPool  |  (32, 32, 32)        =>   (32, 16, 16)        0
StatefulPartitionedCall_sequential_conv2d_1_BiasAdd       |  (32, 16, 16)        =>   (64, 16, 16)        18496
StatefulPartitionedCall_sequential_conv2d_1_Relu          |  (64, 16, 16)        =>   (64, 16, 16)        0
StatefulPartitionedCall_sequential_max_pooling2d_1_MaxPool|  (64, 16, 16)        =>   (64, 8, 8)          0
StatefulPartitionedCall_sequential_conv2d_2_BiasAdd       |  (64, 8, 8)          =>   (64, 8, 8)          36928
StatefulPartitionedCall_sequential_conv2d_2_Relu          |  (64, 8, 8)          =>   (64, 8, 8)          0
StatefulPartitionedCall_sequential_max_pooling2d_2_MaxPool|  (64, 8, 8)          =>   (64, 4, 4)          0
StatefulPartitionedCall_sequential_conv2d_3_BiasAdd       |  (64, 4, 4)          =>   (128, 4, 4)         73856
StatefulPartitionedCall_sequential_conv2d_3_Relu          |  (128, 4, 4)         =>   (128, 4, 4)         0
StatefulPartitionedCall_sequential_max_pooling2d_3_MaxPool|  (128, 4, 4)         =>   (128, 2, 2)         0
StatefulPartitionedCall_sequential_conv2d_4_BiasAdd       |  (128, 2, 2)         =>   (256, 2, 2)         295168
StatefulPartitionedCall_sequential_conv2d_4_Relu          |  (256, 2, 2)         =>   (256, 2, 2)         0
StatefulPartitionedCall_sequential_conv2d_4_BiasAdd__40   |  (256, 2, 2)         =>   (2, 2, 256)         0
StatefulPartitionedCall_sequential_flatten_Reshape        |  (2, 2, 256)         =>   (1024)              0
StatefulPartitionedCall_sequential_dense_MatMul           |  (1024)              =>   (64)                65600
StatefulPartitionedCall_sequential_dense_Relu             |  (64)                =>   (64)                0
StatefulPartitionedCall_sequential_dense_1_MatMul         |  (64)                =>   (10)                650
--------------------------------------------------------------------------------
Total params: 491594
Trainable params: 491594
Non-trainable params: 0

Evaluate with batch size 10
[                                                  ] 1000 StatefulPartitionedCall_sequential_dense_1_MatMul[loss=nan metric=0.171]
```

Figure 49 Inference output for TensorFlow ONNX

In the image we can see that the metric for this network is almost identical to the one obtained randomly classifying the images of the dataset. That is because, as we have said at the beginning of this section, LEDEL does not support the transferring of the trained weights through ONNX files, it only transfers the architecture of the model.

11.2.5 Cross-Compilation

Due to the limitations of the available software, to compile a program on system build on RISC-V may be more difficult than simply to cross-compile on another common machine, and then to transfer the compiled program generated and ready to work on the RISC-V architecture (in our case, in the FRACTAL node)

To this aim, we have assembled a second Docker image with the tools needed to cross-compile a program which uses the LEDEL library. This docker works identically than the previous one and it already has the same code examples as presented in the previous sections, with the difference that anytime we compile a program, we will need to specify the cross-compiler for the RISC-V architecture.

For example, in the console of the Debian system of this docker container we can navigate to the directory "cd /home/eddl_examples/cifar10_eddl_cross_train".

Inside, there are files identical to those describe in section 11.2.1. To cross-compile them, the following commands need to be executed:

```
mkdir build
cd build/
cmake .. -DCMAKE_CXX_COMPILER="/home/isar-riscv/sdk/sdk-debian-sid-ports-riscv6
4/usr/bin/riscv64-linux-gnu-g++"
make
```

Figure 50 Commands needed to follow cross-compilation example

These are the same commands used during a usual compilation, with the addition of the specification of the cross-compiler installed inside the Docker container. After the execution, we can find the executable file in the '/build' directory, but trying to execute it will return an error message. This happens because this executable is compiled to work on a RISC-V architecture and we are trying to run it in a different architecture. If instead, we transfer this file to the QEMU emulation and execute it there, we will get the correct functioning of the program [Figure 51]:

```
riscv@localhost:~/cifar10_examples/cifar10_eddl_cross_train$ ./cifar10_eddl_train
Downloading cifar
cifar_trX.bin x
cifar_trX.bin       100%[===================>] 585.94M  9.76MB/s    in 63s
cifar_trY.bin x
cifar_trY.bin       100%[===================>]   1.91M  8.55MB/s    in 0.2s
cifar_tsX.bin x
cifar_tsX.bin       100%[===================>] 117.19M  9.57MB/s    in 12s
cifar_tsY.bin x
cifar_tsY.bin       100%[===================>] 390.64K  --.-KB/s    in 0.1s
No axis for 'softmax' was specified. Using last one (-1) as default (LActivation::Softmax)
Generating Random Table
CS with full memory setup
Building model
sh: 1: dot: not found
[PLOT] Unable to run the following command:
        => dot -T pdf ./tmp.dot >./model.pdf
----------------------------------------------------------------------------
model
----------------------------------------------------------------------------
input1    |  (3, 32, 32)         =>   (3, 32, 32)         0
conv2d1   |  (3, 32, 32)         =>   (32, 32, 32)        896
relu1     |  (32, 32, 32)        =>   (32, 32, 32)        0
maxpool2d2|  (32, 32, 32)        =>   (32, 16, 16)        0
conv2d2   |  (32, 16, 16)        =>   (64, 16, 16)        18496
relu2     |  (64, 16, 16)        =>   (64, 16, 16)        0
maxpool2d4|  (64, 16, 16)        =>   (64, 8, 8)          0
conv2d3   |  (64, 8, 8)          =>   (128, 8, 8)         73856
relu3     |  (128, 8, 8)         =>   (128, 8, 8)         0
maxpool2d6|  (128, 8, 8)         =>   (128, 4, 4)         0
conv2d4   |  (128, 4, 4)         =>   (256, 4, 4)         295168
relu4     |  (256, 4, 4)         =>   (256, 4, 4)         0
maxpool2d8|  (256, 4, 4)         =>   (256, 2, 2)         0
reshape1  |  (256, 2, 2)         =>   (1024)              0
dense1    |  (1024)              =>   (128)               131200
relu5     |  (128)               =>   (128)               0
dense2    |  (128)               =>   (10)                1290
softmax6  |  (10)                =>   (10)                0
----------------------------------------------------------------------------
Total params: 520906
Trainable params: 520906
Non-trainable params: 0

5 epochs of 500 batches of size 100
Epoch 1
[----------------------------------------------------] 3 softmax6[loss=2.356 metric=0.117] 14.6486 secs/batch
```

Figure 51 Cross-compiled program output

In the QEMU emulation of the docker dedicated to the cross compilation we can find both examples, training and inference, of the executables ready to be launched, in the paths '~/cifar10_examples/cifar10_eddl_cross_train' and ~/cifar10_examples/cifar10_eddl_cross_infenrece'.

11.2.6 Use Case 15 from DeepHealth project

As an example of the capabilities of the LEDEL library executed over a RISC-V architecture, we have decided to implement the process followed in the Use Case 15 from DeepHealth project [11]. The objective of this UC15 is to classify x-ray scans of human lungs between two classes, if those scans come from patients with or without Covid.

To implement the original code of the UC we have selected two different approaches. On one hand we will use a reduced version of the original dataset where the images have been downsized and reformed to binary files. For this case, we will simply re-use the code from the MNIST and CIFAR10 examples to load the images into the neural network and execute the training and test of it.

On the other hand, we are going to simplify the original pipeline of the UC15, removing the data augmentation from the images and skipping the use of the ECLV and OpenCV libraries.

For this demonstration we have used the data available in the following link under the name "uc15_data_for_cpu_mpi_evaluation.tgz" [7]. Firstly, we are going to start the example with a reduced dataset [8].

The data in the .tgz file has been created applying some processing to the original images with the purpose of creating a dataset suitable for Support Vector Machine training, but we think that can also be useful to demonstrate the behavior of the EDDL over RISC-V. When decompressing the file in the link we will find multiple folders, each named using pixel windows size.

Each folder contains the data resulting from measuring the mean and the standard deviation over the image using moving windows of the sizes in their names. For our demonstration we are going to work only with files in the folder '5x5_and_7x7_and_9x9_and_11x11'.

The data is already divided into train, validation and test partition; and we can make use of the EDDL functions to directly load these files as tensors into the C++ code. Once the tensors with the training and test data are loaded, we will start the training and evaluation of a simple neural network (almost identical to the one used in the

MNIST examples). We have used the code implemented in C++ from Figure 52 and Figure 53 for the entire process.

```
int main(int argc, char **argv) {
    // Settings
    int epochs = 5;
    int batch_size = 16;
    int num_classes = 1;

    // Define network
    layer in = Input({2000});
    layer l = in;  // Aux var

    l = LeakyReLu(Dense(l, 1024));
    l = LeakyReLu(Dense(l, 1024));
    l = LeakyReLu(Dense(l, 1024));

    layer out = Sigmoid(Dense(l, num_classes));
    model net = Model({in}, {out});
    net->verbosity_level = 0;

    // Build model
    build(net,
          adam(0.01), // Optimizer
          {"binary_cross_entropy"}, // Losses
          {"binary_accuracy"}, // Metrics
          //CS_GPU({1}) // one GPU
          //CS_GPU({1,1},100) // two GPU with weight sync every 100 batches
          CS_CPU()
          //CS_FPGA({1})
    );
    //toGPU(net,{1},100,"low_mem"); // In two gpus, syncronize every 100 batches, low_me
m setup
```

Figure 52 Use case 15 code, part 1

```
    // View model
    summary(net);

    // Load dataset
    Tensor* x_train = Tensor::load("uc15_dataset/X_train.bin");
    Tensor* y_train = Tensor::load("uc15_dataset/y_train.bin");
    Tensor* x_test = Tensor::load("uc15_dataset/X_test.bin");
    Tensor* y_test = Tensor::load("uc15_dataset/y_test.bin");

    // Preprocessing
    float x_max = x_train->max();
    x_train->div_(x_max);
    x_test->div_(x_max);

    for(int i=0;i<epochs;i++)
        std:cout<<"Epoch "<< i <<std::endl;
        // Train model
        fit(net, {x_train}, {y_train}, batch_size, 1);
        evaluate(net, {x_test}, {y_test}, batch_size);

    save_net_to_onnx_file(net, "uc15_eddl_net.onnx");
    save(net, "uc15_eddl_weights.bin");
}
```

Figure 53 Use case 15 code, part 2

The most important aspects of this code are:

- The input layer of the network receives a Tensor of size 2000, this is the size of every image after applying the preprocessing with moving windows.
- "num_classes" has been defined as 1 since this a binary classification problem. In reality there are 2 classes and it is defined this way because we only require one output from the network.
- The last layer of the network is a Sigmoid layer connected to dense layer with only one output value. This makes the output of the entire network a value between 0 and 1. If the final output is closer to 0 the input image is classified in the first class and if is close to 1 in the second class.
- Instead of dividing the training and test tensors loaded by 255, as we should do when loading a PNG image, in this case we normalize each Tensor dividing by the maximum value in it.
- The training and evaluation are done one to one. In each epoch the network is trained using the whole train dataset and evaluated with the test dataset.
- The resulting network is stored in ONNX format.

Now that the code and use case is presented, we start the training process as described in 11.2.1, and we get the output shown in figure below:

```
riscv@localhost:~/uc15/uc15_reducido$ make
Scanning dependencies of target uc15_eddl_train
[ 50%] Building CXX object CMakeFiles/uc15_eddl_train.dir/main.cpp.o
[100%] Linking CXX executable uc15_eddl_train
[100%] Built target uc15_eddl_train
riscv@localhost:~/uc15/uc15_reducido$ ./uc15_eddl_train
Generating Random Table
CS with full memory setup
Building model
--------------------------------------------------------------------------------
model
--------------------------------------------------------------------------------
input1     |  (2000)             =>   (2000)             0
dense1     |  (2000)             =>   (1024)             2049024
leaky_relu1|  (1024)             =>   (1024)             0
dense2     |  (1024)             =>   (1024)             1049600
leaky_relu2|  (1024)             =>   (1024)             0
dense3     |  (1024)             =>   (1024)             1049600
leaky_relu3|  (1024)             =>   (1024)             0
dense4     |  (1024)             =>   (1)                1025
sigmoid4   |  (1)                =>   (1)                0
--------------------------------------------------------------------------------
Total params: 4149249
Trainable params: 4149249
Non-trainable params: 0

Epoch 0
1 epochs of 811 batches of size 16
Epoch 1
[                                        ] 147 sigmoid4[loss=nan metric=0.598] 1.0006 secs/batchch
```

Figure 54 Output

Here we encounter an unexpected error. During the first training epoch, when executing the code over a RISC-V architecture, the function loss suddenly skyrockets and our metric starts to descend until reaches an almost zero value by the end of the epoch. On the other hand, the same code runs without problem during training epochs when executed on a x86 architecture, achieving acceptable metric values.

The strategies followed to address this bug have been:

1. In the code of this use case different layers were used. Thus, we swapped them for the ones we had check worked in the previous examples that we certainly know have proper behavior. The bug appeared again.
2. We thought the problem was in the initiation of the data during the preprocessing step. Therefore, we normalized the data out of the program to be sure it was correct. This was not the problem.
3. We decided to use the other RISC-V image that we talked about in WP3 (see deliverable D3.6 for more details). With this approach, the LEDEL in this use case worked.

Currently, we are planning what steps should be taken in order to have the LEDEL completely working on the RISC-V SIEMENS/ISAR, since the use of the LEDEL is a requirement for Use Case 7.

12 Conclusions

In this document, the preliminary implementations for low-power services described in the deliverable D4.1 have been elaborated. Several building blocks and components that contribute to meeting the T4.1 objectives and could be demodulated/exploitable by any Use Case have been developed. Specifically, they are data Compression for Low-Power Services, HATMA, Low Power services for PULP systems, Versal RPU access for low Power Services, agreement Protocol for Low-Power Services, and Versal Isolation Design- Functional Safety. In particular, a thorough explanation, design, implementation, testing, and assessment of these components have been reported. Furthermore, T4.1 validated the LEDEL Library's adaptability to low-power services.

13 Bibliography

Ref.	Title
[1]	Axer, P., Ernst, R., et al.: Building timing predictable embedded systems. In: ACM Transactions on Embedded Computing Systems (TECS) **13**(4). (2014)
[2]	https://github.com/siemens/isar-riscv
[3]	https://github.com/project-fractal/WP3/tree/main/Components/WP3T35-03%20LEDEL
[4]	https://www.cs.toronto.edu/~kriz/cifar.html
[5]	https://deephealthproject.github.io/eddl/model/model.html#_CPPv4N4eddl5buildE5model9optimizerRK6vectorI6stringERK6vectorI6stringEP8CompServb
[6]	https://pytorch.org
[7]	https://clocalprog.dsic.upv.es/winter-school/data/uc15_data_for_cpu_mpi_evaluation.tgz
[8]	https://clocalprog.dsic.upv.es/winter-school/data
[9]	https://github.com/asad82/LZW-Compression
[10]	Joshi, M. (2015). Lossless Compression, *Proof Patterns* (pp. 21–22). Springer International Publishing Switzerland.
[11]	https://deephealth-project.eu

14 List of figures

15 List of tables

16 List of Abbreviations

Acronym	Title
AI	Artificial Intelligence
API	Application Programming Interface
ATTNoC	Adaptive Time-triggered Network-on-Chip
CAU	Context Agreement Unit
CM	Context Monitor
DMA	Direct Memory Access
DVFS	Dynamic Voltage and Frequency Scaling
ECVL	European Computer Vision Library
EDDL	European Distributed Deep Learning Library
EDP	Energy-Delay Product
GCC	GNU Compiler Collection
GNU	GNU is Not Unix
HATMA	Hierarchical Adaptive Time-triggered Multi-core Architecture
HICP	Hierarchical Interactive Consistency Protocol
HW	Hardware
IoT	Internet of Things
ISA	Instruction Set Architecture
LEDEL	Low Energy DEep Learning Library
MMU	Memory Management Unit
NI	Network Interface
NOC	Network on Chip
ONNX	Open Neural Network Exchange
PE	Processing Element
PULP	Parallel Ultra Low Power
QEMU	Quick EMUlator
QoS	Quality of Service
RPU	Realtime Processing Unit (Versal)
SoC	System on Chip
SW	Software
TT	Time-triggered
UC	Use case
VHDL	Very high-speed integrated circuit Hardware Description Language
WCET	Worst Case Execution Time
XMPU	Xilinx Memory Protection Unit