http://www.everest-h2020.eu

dEsign enVironmEnt foR Extreme-Scale big data analyTics on heterogeneous platforms

D5.1 — Intermediate runtime environment report

The EVEREST project has received funding from the European Union's Horizon 2020 Research & Innovation programme under grant agreement No 957269

Project Summary Information

Project Title	dEsign enVironmEnt foR Extreme-Scale big data analyTics on heterogeneous platforms
Project Acronym	EVEREST
Project No.	957269
Start Date	01/10/2020
Project Duration	36 months
Project Website	http://www.everest-h2020.eu

Copyright

Num.	Partner Name	Short Name	Country
1 (Coord.)	IBM RESEARCH GMBH	IBM	CH
2	POLITECNICO DI MILANO	PDM	IT
3	UNIVERSITÀ DELLA SVIZZERA ITALIANA	USI	CH
4	TECHNISCHE UNIVERSITAET DRESDEN	TUD	DE
5	Centro Internazionale in Monitoraggio Ambientale - Fondazione CIMA	CIMA	IT
6	IT4Innovations, VSB – Technical University of Ostrava	IT4I	CZ
7	VIRTUAL OPEN SYSTEMS SAS	VOS	FR
8	DUFERCO ENERGIA SPA	DUF	IT
9	NUMTECH	NUM	FR
10	SYGIC AS	SYG	SK

Project Coordinator: Christoph Hagleitner – IBM Research – Zurich Research Laboratory

Scientific Coordinator: Christian Pilato – Politecnico di Milano

Disclaimer

Deliverable Information

Work-package	WP5
Deliverable No.	D5.1
Deliverable Title	Intermediate runtime environment report
Lead Beneficiary	PDM
Type of Deliverable	Report
Dissemination Level	Public
Due Date	31/03/2022

Document Information

Delivery Date	31/03/2022
No. pages	22
Version | Status	0.5 | Final
Responsible Person	Gianluca Palermo (PDM)
Authors	Gianluca Palermo (PDM), Roberto Rocco (PDM), Michele Paolino (VOS), Stanislav Bohm (IT4I)
Internal Reviewer	Jeronimo Castrillon (TUD)

The list of authors reflects the major contributors to the activity described in the document. All EVEREST partners have agreed to the full publication of this document. The list of authors does not imply any claim of ownership on the Intellectual Properties described in this document.

Revision History

Date	Ver.	Author(s)	Summary of main changes
23.08.2021	0.0	M. Paolino (VOS)	Skeleton
01.10.2021	0.1	M. Paolino (VOS)	Initial RunTime View
01.11.2021	0.2	G. Palermo & R. Rocco (PDM)	First contributions on Dynamic Autotuning
18.01.2022	0.3	M. Paolino (VOS)	Initial contribution on the Virtualization
25.01.2022	0.31	S. Bohm (IT4I)	Initial contribution on multinode-runtime
01.02.2022	0.32	G. Palermo & R. Rocco (PDM)	First contributions on Dynamic Autotuning
10.03.2021	0.4	R. Rocco & C. Pilato (PDM)	Review of the contributions
29.03.2021	0.5	R. Rocco & G. Palermo (PDM)	Review of the contributions

Quality Control

Approved by Internal Reviewer	March 30, 2022
Approved by WP Leader	March 31, 2022
Approved by Scientific Coordinator	March 31, 2022
Approved by Project Coordinator	March 31, 2022

Table of Contents

1 Executive Summary

The EVEREST project proposes a platform for implementing big data applications following a data-driven model. This document provides an intermediate description of the run-time environment by defining its main components. In EVEREST, the run-time environment provides adaptivity features to the application, integrating and dynamically selecting kernel variants generated by the EVEREST compilation framework (cf. Deliverable D4.2) while reacting to execution environment changes, with support for task distribution across a multi-node architectures and virtualization features needed to make the FPGA visible into virtual machines under different scenarios.

Deliverable highlights

No.	Highlight	Section(s)
1	Overview of the runtime environment with main EVEREST developed components	Section 2
2	Dynamic autotuning framework and integration with the variants generated by the compiler framework	Section 3.1
3	Lightweight runtime layer for the multi-node application deploymen	Section 3.2
4	Overview of the virtualization extensions targeting FPGAs	Section 4

1.1 Structure of the Document

This document starts with an overview of the run-time environment in Section 2 to provide the context of the main components that are part of the remaining sections. In particular, Section 3 describes the main components under development for the run-time environment that are at the application and scheduling level. The section includes the current development status and simple usage scenarios. Section 4 describes the virtualization extensions under development within the project to make the host-attached FPGA accessible with the same capability within the guest OS. Section 5 concludes the deliverable and provides hints on the next steps.

1.2 Related Documents

The content of the document should be considered in a wider view of the project as described in the project paper [11]. Moreover, this document has a tight link with the following deliverables as they represent the actual software release of the run-time components:

- *D5.2 - Alpha release of the runtime support* [5]
- *D5.3 - Alpha release of the dynamic adaptation framework* [6]
- *D5.4 - Alpha release of the virtualization environment* [7]

Finally, a link with the deliverable *D4.2 - Intermediate report of the compilation framework* [4] is also present in the document when referring to the automatic variant generation part taken as input by the autotuning and dynamic adaptation framework.

2 Runtime and Virtualization Environment Architecture Overview

Executing an application in a heterogeneous environment is a non-trivial task. It requires adaptation in both directions: the application must be optimized for the resources provided and the system must decide how to split the resources among the different requests. The main goals of the EVEREST Work Package 5 (Virtualized Runtime Environment) activities are to deal with this task by enabling optimized execution of the application code, and efficient distribution and execution of the available resources. These activities develop into three main dimensions that represent also the three main tasks of the work package:

- A lightweight runtime layer to implement the multi-node application deployment in order to hide it from the application developer (Task 5.1);
- A dynamic autotuning framework capable to configure the target application in terms of available and suitable software variants generated by the EVEREST compilation framework, depending on the run-time scenario and the underlying hardware (Task 5.2);
- Virtualization extensions on top of the hardware accelerators, so that the virtual machines can fully exploit the underlying hardware capabilities (FPGA as the main target) (Task 5.3).

It is possible to represent the interactions of the three Tasks of the work package as shown in Figure 1. The figure shows also the interaction with other work packages, in particular the compilation flow (work package 4) and the development of the use-case applications (Work Package 6). The runtime layer operates using the application built by the compiler: it takes the application tasks generated by the compilation flow and schedules them on the available resources allocated, according to the users' request. Managed resources can be distinguished between nodes exploiting FPGA-as-a-Service functionalities (i.e. CloudFPGA) and others featuring the accelerator locally. In particular, regarding this second case, a task can be configured (version selection) by the autotuning library that interacts with the Knowledge Extractor modules to choose the best configuration given the allocated resources. The virtualization extensions enable the isolation of different executions sharing resources.

The actual execution is preceded by a learning phase since the autotuning components need to collect data about the application to optimize it properly. During the learning phase, the tasks are executed many times, checking the most relevant configurations for all the possible resource allocations. The learning phase terminates when the application knowledge is produced, and it will be used for the optimization of the actual tasks.

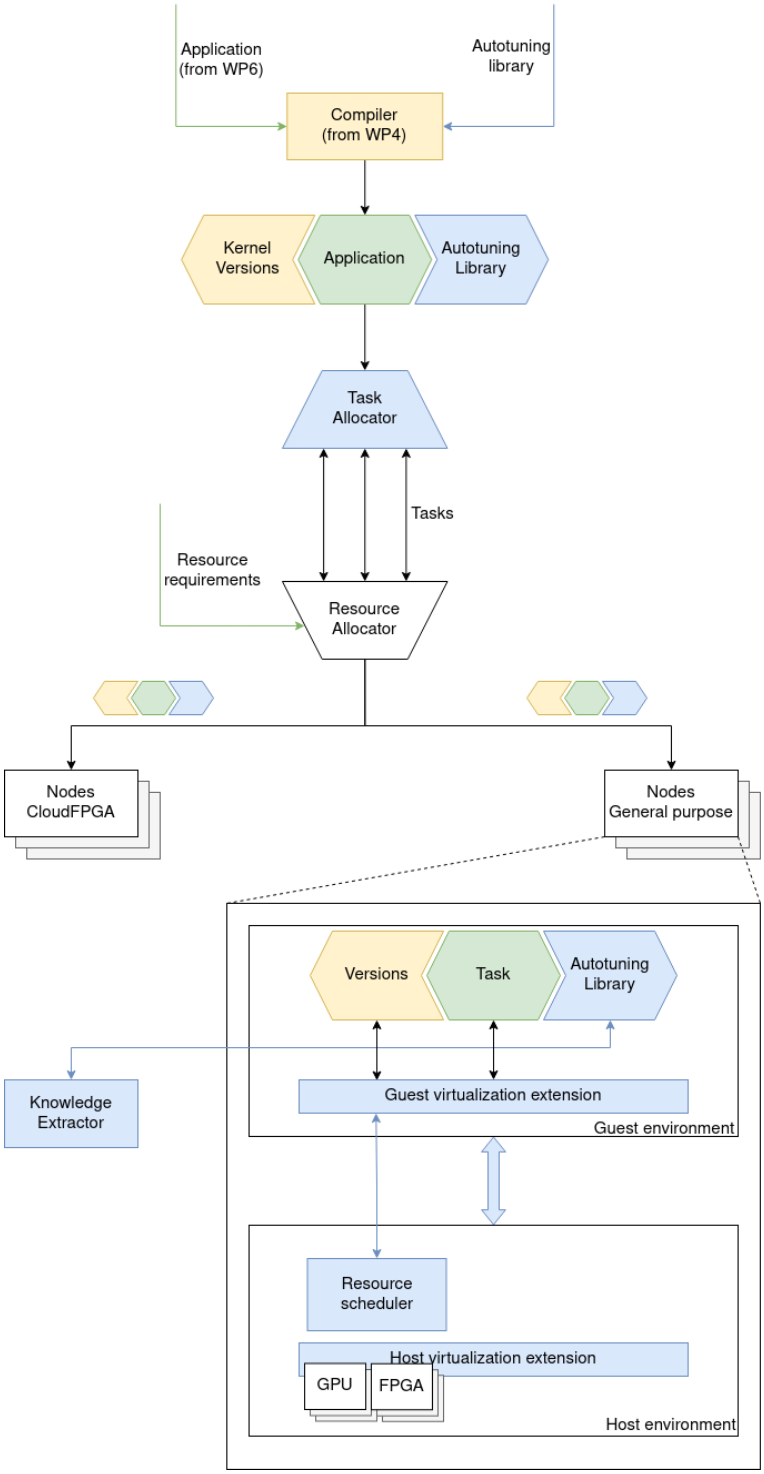

Figure 1 – Overview of the runtime view. In yellow components coming from work package 4, in green components from work package 6, in blue developed in work package 5.

3 EVEREST Components for the Runtime Environment

As mentioned in Section 2, the main components under development within EVEREST for the runtime environment are at the application and scheduling level. In Section 3.1, we describe the current development status for the application level framework including a local library for configuration and version selection and remote engines for learning and monitoring. In Section 3.2 we describe the scheduler and multi-node support that enables distributing computation across a cluster of computing nodes.

3.1 Application Autotuning and Dynamic Adaptation Framework

The heterogeneity of the execution environment and the radical differences between the use cases imply that the same configuration of runtime variables may not always be the best one. The choice of the optimal configuration becomes critical since it heavily impacts the performance levels reachable during the execution. Moreover, the optimal configuration may change during the execution due to sudden modifications to the environment or changes in the characteristics of the inputs. To aspire to the correct optimization of the code, we designed and implemented the mARGOt framework by extending the already existent library [9] to tackle all the difficulties shown above. mARGOt can make the best decision regarding the configuration variables based on the knowledge it has of the application. While it can be obtained by extracting it manually via offline profiling, we have now extended the framework with the capability to extract it at runtime and in an automatic and scalable way.

In the following, we describe the components of the mARGOt framework and how they interact internally (Section 3.1.1), the interactions between the framework and the other components of the runtime environment (Section 3.1.2), and how to use the mARGOt framework within an application (Section 3.1.3). Finally, in Section 3.1.4 we show an alternative and simplified way to integrate the mARGOt framework when considering a multi-versioned kernel.

3.1.1 mARGOt Framework Structure

The mARGOt framework consists of three main components:

- the mARGOt autotuning library, performing the actual application configuration;
- the AGORA application knowledge extractor, that creates the knowledge used by mARGOt for decision making;
- the Theo runtime observer, that monitors the execution and decides when there is a need for a model retraining.

The entire framework can be represented as in Figure 2, which also highlights the interactions between the components. The interaction between the components uses telemetry protocols (MQTT), using the JSON format for data exchange.

The *mARGOt autotuning library* performs runtime optimization by choosing the best configuration among the ones offered by the application. It consists of a C++ library that is linked with the application and requires a small modification to the application code to introduce its API. mARGOt works by using knobs and metrics:

- knobs are variables controllable by the library and are used to set the chosen configuration;
- metrics are observable variables whose value reflects the functional and extra-functional properties required for the execution.

The mARGOt library chooses the best knobs values based on the value of the metrics observed and other uncontrollable factors, like characteristics of the input or hardware environment. The latter are reflected by the input features, variables of the code observed by mARGOt. The best knob value selection is based on

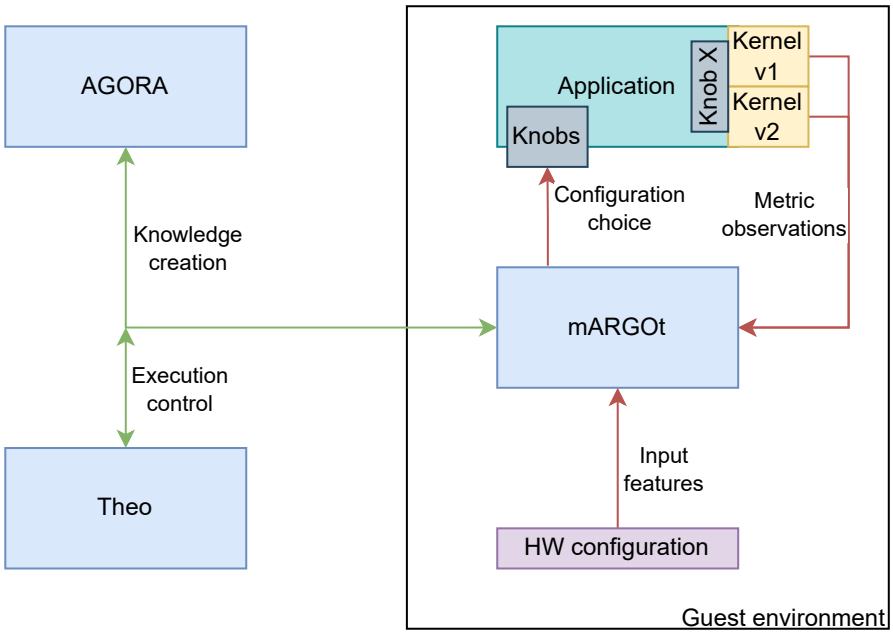

Figure 2 – Structure of the mARGOt autotuning framework.

the application knowledge. The knowledge consists of a collection of configurations with the effect they are expected to cause on the metrics of interest. Given an application, the production of its knowledge is a non-trivial task yet a mandatory one to perform autotuning. The mARGOt library by itself does not provide methods to extract it as its management is the duty of the other two components of the framework (AGORA and Theo).

The AGORA application knowledge extractor is a server application that interacts with the mARGOt library. Usually, AGORA does not run in the same node of the application and is remotely connected to the application via the MQTT telemetry protocol. Its focus is to enable the learning of the application knowledge at runtime, but can also be used to extract the knowledge during profiling. AGORA operates from application launch-time: it suggests mARGOt the configurations to run and it gathers the execution data. All this information is the basis for the production of a configuration space model, an analytical counterpart of the relation between the application knobs, the data features, and the target metrics. Using that model, AGORA creates the application knowledge and sends it to mARGOt, which proceeds to execute in an optimized way. The combination of AGORA and mARGOt is enough to provide dynamic adaptivity, but cannot ensure an effective optimization during the entire execution: the learning process done by AGORA is performed only once (when there is no prior knowledge). This means that a change in the execution environment that radically affects the behaviour of the application is not reflected in a modification of the application knowledge, resulting in a loss of optimality.

The Theo runtime observer deals with changes in the execution environment. It operates by checking the information exchanged between mARGOt and AGORA and keeps track of the monitored characteristics of the execution environment. Theo can run on a different node and monitors the interactions between mARGOT and AGORA by listening on the same MQTT channel. Whenever it notices that something has changed, it restarts the AGORA knowledge learning procedure and combines the results obtained with the previous ones.

The interactions between the components can be summarized as in the sequence diagram shown in Figure 3. The diagram describes the sequence of messages that are sent among the three components. When an application starts, it sends a welcome message to AGORA and Theo notifying its presence and asking to open a dedicated application channel for the specific mARGOt instance linked with the application, providing the configuration space. AGORA and Theo are unique instances able to serve multiple applications at the same time. Once the welcome message has been received, AGORA starts suggesting application configurations to be executed (and monitored) to build the application knowledge. Once the application knowledge is correctly built, AGORA broadcasts this info to the local instance of mARGOt that starts selecting the best configuration on its own (autonomously). Theo receives the application knowledge too and stores it for future reference. The application, during its autonomous execution, sends monitoring data to both AGORA and Theo.

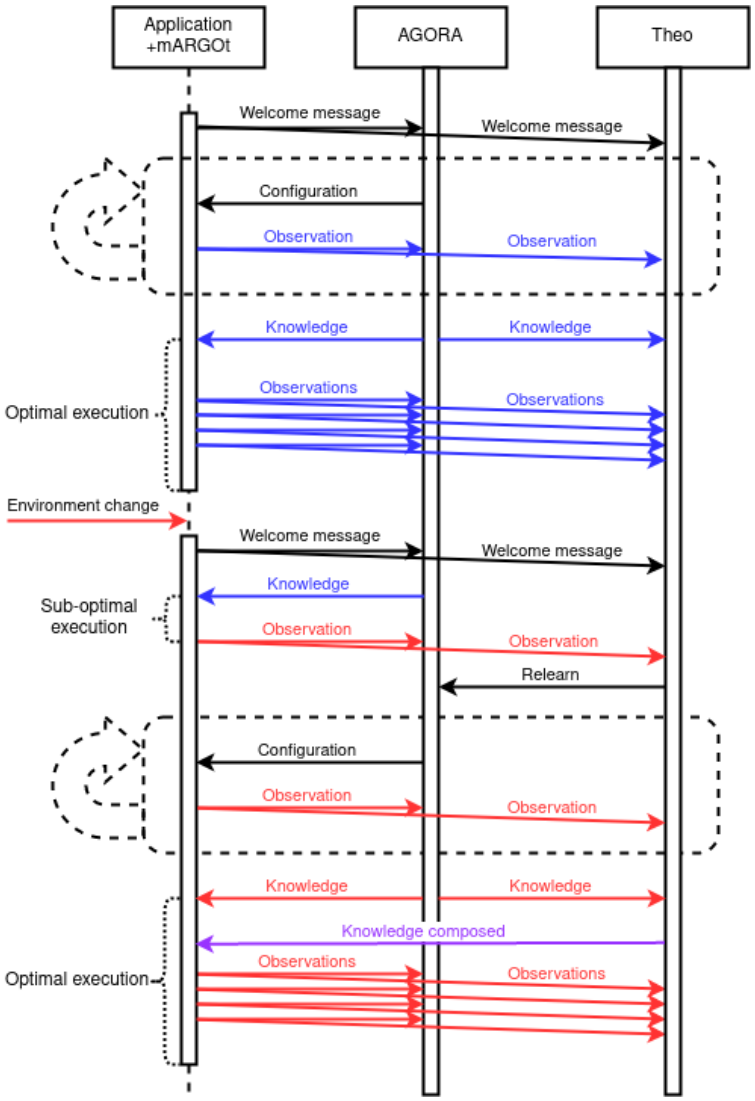

Figure 3 – Typical flow of execution of the optimization process. Red and blue arrows identify results obtained in different execution environments. The purple arrow indicates that the knowledge contains data about the blue and red configurations.

Let us suppose now that a new instance of the application is started. This new instance does not need to perform the learning phase since it can rely on the knowledge previously collected and stored in AGORA. If nothing changed in the execution environment, mARGOt configures the application for optimal execution. If something changes in the execution environment (e.g. a different HW configuration or different characteristics of the data), Theo detects it by analyzing the incoming observations and forces AGORA to restart the learning phase. At the end of the re-learning phase, the new knowledge is sent out by AGORA. Theo gathers the new knowledge and combines it with the previous one so that the local mARGOt can use both of them according to the actual condition.

3.1.2 Interaction with other Runtime Components

The mARGOt autotuning framework does not interact much with the other components of the runtime environment, but it is heavily influenced by their decision. Ideally, the mARGOt library should be able to extract all the needed data about the execution environment from the `/proc` filesystem, but this functionality is not embedded into the library and must be defined in the application code, so it is possible to communicate using different mediums. The detection of the execution environments enables the reactive approach introduced with the Theo runtime observer and it is key for accurate and fast optimization. Examples of execution environment characteristics include the number of cores of the CPU used, its frequency, its utilization and the presence of other accelerators (GPUs, FPGAs) compatible with the application code (or with some variants).

While not directly required by the mARGOt framework, interaction with the resource allocation components of the runtime environment can produce benefits for the overall process: the application knowledge produced by AGORA contains data that can be useful for the resource allocation decisions. Since the mARGOt framework does not benefit directly from this interaction, it does not need to be aware of it: the application knowledge can be directly fetched from the communication medium, without any impact on the autotuning framework.

The main interaction of the mARGOt autotuning framework is the one with the application. This topic, together with all the configuration settings needed to make the framework behave correctly, is discussed in the next section.

3.1.3 Using the mARGOt Framework

The user can interact with the mARGOt framework mainly through the mARGOt library, both with code modifications and configuration files. Code modifications are a required step to integrate the library: mARGOt provides an automatically generated C++ interface that can handle the change of the knob variables and the notifications of the metrics observed. Figure 4 contains a C++ code snippet of an application using the mARGOt library interface to optimize its execution. The application firstly calls an initialization function that prepares all the needed structures (line 2), then defines knobs (line 6) and input features (lines 7-8). It then computes the values of the input features (lines 12-17) and checks whether the current configuration is optimal (line 19, the update function tunes the knob values and returns true if they were changed). After those steps, it starts the monitors (line 24), executes the code to be optimized (line 25), stops the monitors that are computing all the needed metrics (lines 26-27), and prints information about the execution (line 29).

```
1  int main() {
2      margot::init();
3      function_ptr_t do_work_array[2];
4      do_work_array[1] = do_work_hw_boosted;
5      do_work_array[0] = do_work_sw;
6      int version = 0;
7      int hw_available = 0;
8      int data_feature = 0;
9      srand(time(NULL));
10     int repetitions = 1000;
11     for (int i = 0; i<repetitions; ++i) {
12         data_feature = extract_data_feature();
13         const char* presence = std::getenv("HW_MODULE_AVAILABLE");
14         if(presence)
15             hw_available = 1;
16         else
17             hw_available = 0;
18         //check if void configuration is different wrt the previous one
19         if (margot::block1::update(data_feature, hw_available, version)) {
20             margot::block1::context().manager.configuration_applied();
21             std::cout << "< < < CHANGE APPLIED > > >" << std::endl;
22         }
23         //monitors wrap the autotuned function
24         margot::block1::start_monitors();
25         int error = do_work_array[version](data_feature, hw_available);
26         margot::block1::stop_monitors();
27         margot::block1::push_custom_monitor_values(error);
28         //print values to file
29         margot::block1::log();
30     }
31 }
```

Figure 4 – Code snippet from an application using the mARGOt interface.

The code snippet shown in Figure 4 can work only if a compatible configuration file is provided during the compilation. The configuration file includes information about the different variables involved in the optimization process and settings for the other parts of the framework. The configuration file specifies knobs variables, metrics observed (together with the monitors which will observe them), features of the inputs and the optimization problem. It includes the configuration to communicate with the AGORA server application (and Theo) and all the needed parameters to configure the knowledge learning procedure. Additional details about the

configuration files can be found in [6].

Being a C++ library, it is easy to integrate mARGOt into C++ applications but care is needed when working with applications written in other languages. While the requirements defined in [11] limit the target of the project to C++ applications, the use case developments [8] added the need for to support other languages, in particular Rust and Python. To simplify the integration with such languages, we are developing the Adam component, able to generate a C++ application from the mARGOt configuration files: the generated application will interact with mARGOt for the optimization, and will then send the optimized values to the original application using a telemetry protocol (MQTT). This process eases the integration between mARGOt and applications written in any programming language: the application developer must just develop an additional interface to extract data from the communication medium and then it is possible to use the optimized values provided by mARGOt. Figure 5 shows how the integration changes when using the Adam component: the application must interact with the Adam interface, written in the application language, which will communicate via MQTT with the Adam component, written in C++ and automatically generated starting from the mARGOt configuration file. mARGOt will optimize the Adam component, whose values will be the same as the original application. The result of the optimization will be forwarded back to the application, which can execute with the values selected by the mARGOt library. While this way of integrating the application with the mARGOt framework is compatible with any programming language (only the Adam interface must be changed), it introduces overheads due to the communication over the MQTT medium, so it is better to use it only when strictly necessary.

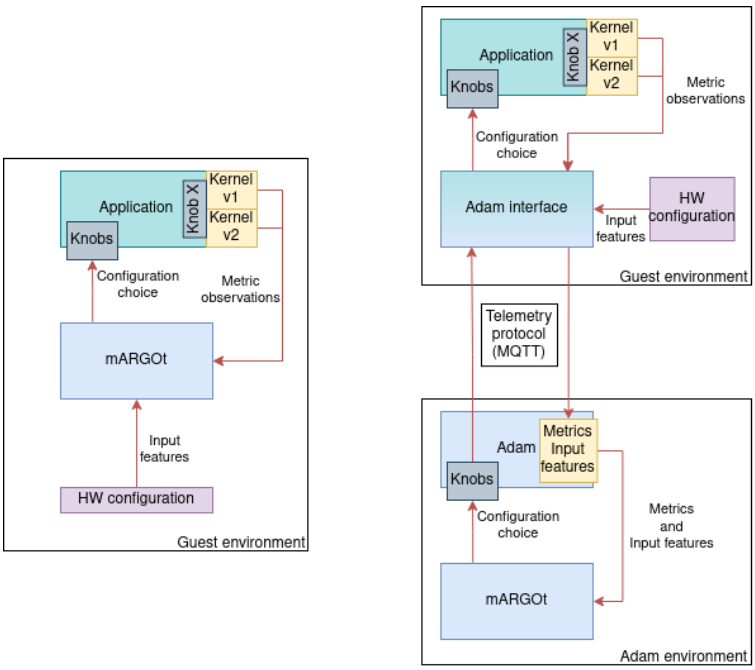

Figure 5 – Comparison between normal integration (on the left) and Adam integration (on the right).

3.1.4 Multiple-version Kernel Integration.

One of the tuning parameters we are interested in exploring within the EVEREST project is the selection of the right code variant when different alternatives are available. Having more than one implementation for a given kernel can be justified by several needs envisioned by the project, for example:

- Target server architectures can be composed of different hardware components, from the CPU-HW point of view but also the accelerator point of view (availability of FPGAs or GPUs). This makes it possible to have versions that are prepared considering the exploitation of the peculiar characteristics of the architecture used for the execution;
- In the case of FPGA-enabled nodes, the modules/versions deployable on the FPGA can be optimized considering different objectives, e.g. the versions may be taken from a Pareto curve for performance-

resource usage. This also accounts for versions that deploy multiple accelerators on the same FPGA card. From the code perspective, the knowledge of the type and number of accelerator available or deployable is known only at run-time;

- The implemented versions can be thought of considering different utilization scenarios considering the data to be processed and how to transfer them to the accelerators, e.g. single data processing vs batch execution. The different requirements and characteristics of the data can be known only at run-time;
- The different versions require different values of compile-time parameters, allowing to optimize those values (and the related code) without the need for a dynamic recompilation;

The mARGOt autotuning framework can be used on code featuring alternative implementations to choose the most promising version for each execution. An example implementation can be seen also in Figure 4: in lines 4 and 5 two alternative versions of the same function `do_work` are put into an array, and the one to be executed is chosen depending on a knob variable (version, see line 25). While this pattern is quite general, it requires that the functions must provide the same signature, which depends on the problem solved by the function itself.

To ease the integration of mARGOt into a multi-versioned kernel, we developed a set of scripts that works at compilation time and assists the generation of the versions, their collection and the integration with the mARGOt library. The set of scripts generate the interface to be implemented with the versions and the C++ files that eventually call the most suitable version using mARGOt. It also generates the files needed to compile the produced code, making the process from version generation to their integration inside an application a fully automated task. The flow is shown in Figure 6.

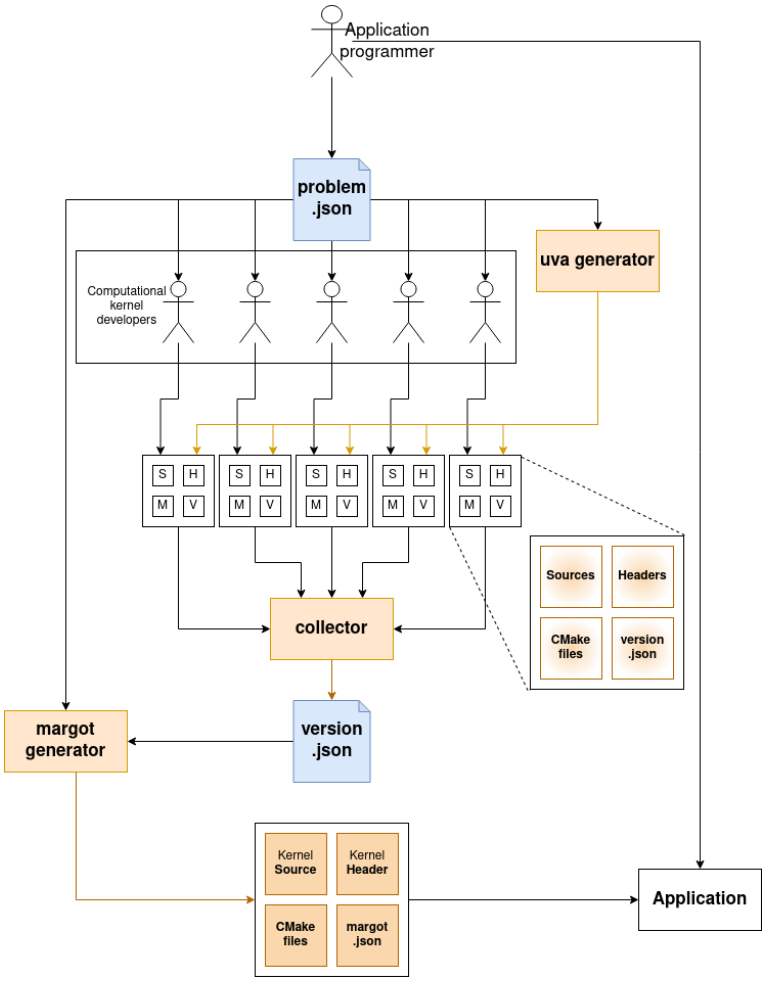

Figure 6 – mARGOt generator flow for multi-versioned kernels.

The mARGOt generator flow starts with a rigorous definition of the problem in a `problem.json` file, which must contain all the inputs and outputs of the multi-versioned function (see Figure 7). The file must also contain

information about eventual data features present in the input values. This file is used by version developers to produce an alternative implementation of the problem and is fed to a generator script (uva generator) to pre-generate the interface of the versions. Each version produced must feature a `version.json` file, containing information about the function that it implements. All these configuration files are collected into a single one (see Figure 8) including fields related to the implementation for each version and the condition needed for its execution (e.g. support functions querying for specific hardware on the platform). This collective file is fed to a generator script that produces the C++ sources with mARGOt that can choose the best version. This last script will also produce the CMake files needed for the correct compilation and usage of the generated sources. Additional details on the flow and the scripts can be found in Deliverable D5.1 [6].

```
{
  "name": "helmholtz",
  "input": [
    {
      "name": "S",
      "type": "matrix_t"
    },
    {
      "name": "D_inv",
      "type": "tensor4D_t"
    },
    {
      "name": "u",
      "type": "tensor4D_t"
    }
  ],
  "feature": [],
  "output": [{
    "name": "r",
    "type": "tensor4D_t"
  }],
  header": "#include <path_to_header>/inverse_helmholtz.hpp"
}
```

Figure 7 – Example of a `problem.json` file to manage different versions of the `inverse_helmholtz` kernel. It describes the interface that is used by the application to call the multi-versioned kernel.

```
{"versions": [
    {
        "name": "naive",
        "header": "#include \"naive.hpp\"",
        "packer": "naive::packer",
        "executor": "naive::executor",
        "unpacker": "naive::unpacker",
        "folder": "naive",
    },
    {
        "name": "fpga",
        "header": "#include \"fpga.hpp\"",
        "packer": "fpga::packer",
        "executor": "fpga::executor",
        "unpacker": "fpga::unpacker",
        "folder": "fpga",
        "condition": {
            "function": "fpga::condition()",
            "type": "int",
            "constraint": "[](int condition) -> bool {return condition != 0;}"
        }
    }
]}
```

Figure 8 – Example of a `version.json` file including 2 versions for the `inverse_helmholtz` kernel. The file is obtained by the union of the files of the various versions.

While the flow in Figure 6 supposes human-generated versions of a kernel, the flow is of course general and can receive versions that are automatically generated from higher-level tools. As discussed in Deliverable D4.2 [4], for instance, the kernel compiler can be instructed to generate multiple different variants from the high-level DSL for tensor expressions. At the moment, the variants differ in their internal implementation (e.g.,

different tensor factorizations, different loop nestings or different loop optimizations). Selecting a particular variant is non-trivial [12] for which mARGOt autotuning approach is very useful. This is especially true when generating not only pure software variants but also code variants that have to be processed using High-Level Synthesis tools to generate hardware accelerators. The compiler framework described in Deliverable D4.2 can generate the files and the interfaces as specified in Figure 6. The mechanisms and interfaces are now in place and can be used to implement more intelligent variant selection in the remaining of the project.

3.2 Multi-node Support

The goal of multi-node support is to provide a way to define a computation across multiple nodes on the EVEREST platform.

We provide an API to describe a computation that may be distributed across multiple nodes. We also provide an implementation of that API that can deploy and execute computations on the cluster nodes. This includes:

- translating the high-level description into single-node tasks;
- scheduling tasks (choosing on which node should a task run, maintaining dependency invariant);
- performing data transfers between nodes according to dependencies;
- monitoring nodes and reacting to the situation when a node is lost or a new node appears.

We decided to base the interface on the Dask API since it is compatible with EVEREST needs and well established in the big data domain.

3.2.1 Dask Bag API

Dask (https://dask.org/) is a well-established Python package for data science and scientific computing. It provides a low-level API and several high-level APIs for building task graphs. One of the high-level APIs is Dask Bag API, which supports working with unordered collections (bags). It provides operations like "map" or "reduce" to perform operations over a bag. An example of this API is shown in Figure 9. Dask can execute such pipelines in a distributed way across the cluster.

```
import dask.bag as db
import json

data = db.read_text('data/*.json').map(json.loads)
result = data.filter(lambda record: record["age"] >= 18).compute()
```

Figure 9 – A simple example of Dask Bag API usage; it loads json files and filter out some of entries.

3.2.2 EVEREST Python API

We are providing a reimplementation and extension of the Dask Bag API as the main interface for multi-node computation. Our goal is to provide sufficient compatibility to be able to run an existing application using Dask Bag API with a minimal modification of the source code.

Our implementation is split into two parts:

- EvKit (https://code.it4i.cz/boh126/evkit) is a Python module providing a Dask Bag API implementation, developed within EVEREST. It is responsible to build an internal task graph representation and it translates tasks into HyperQueue jobs which is used as an execution framework. EvKit also manages data transfers between tasks and prepares the environment for task execution.

- HyperQueue (https://github.com/It4innovations/hyperqueue) is a framework for running tasks on HPC clusters. It is designed to run many small tasks on that does not necessarily utilize a whole node. It also works transparently over system schedulers like PBS and SLURM, without manual and static grouping of tasks. HyperQueue is developed by IT4I outside of EVEREST but it is extended in two directions to support the project: generic resource management and richer Python API.

We provide our implementation of the Dask Bag API over HyperQueue. To this end, we do not use any part of the original implementation of Dask Bag or Dask itself, despite Dask being an open-source project. Reasons for this are:

- In the Dask codebase, the scheduler is tightly integrated into the server, making it hard to extend it with features from HyperQueue (finer grain resource management and transparent coordination with PBS/SLURM).
- The original Dask runtime has some performance limitations and for some use cases, this has a large impact on the total computational time. A deep analysis on this issue has been done in a previous work [2]. As a result, we decided to implement RSDS (https://github.com/It4innovations/rsds), which provides an alternative implementation of Dask in Rust with higher performance. The core part of RSDS is used within the scheduler for HyperQueue.

3.2.3 Usage

For running an application that uses the Dask Bag API over the EVEREST platform, the only modification needed to the application is to change the imports from `import dask` to `import evkit.dask` and to setup HyperQueue over the cluster. These steps are described in more detail in Deliverable D5.2.

3.2.4 Architecture for Multi-node Support

The architecture for the multi-node support is shown in Figure 10. It shows how the user code is distributed across multiple nodes through the API exposed by EvKit and HyperQueue. The flow from the user perspective is the following:

1. The user application uses the Dask Dag API.
2. When Bag methods are called, EvKit composes a high-level task graph representing the computation.
3. When the computation is triggered (usually by calling `.compute()` or `.persist()` method), EvKit creates a low-level task graph for the HyperQueue server from the high-level graph. Tasks in this graph are an invocation of Python with serialized Python functions.
4. The HyperQueue server is a central component that drives the computation. It contains a scheduler that schedules tasks to EVEREST nodes and may also interact with PBS/SLURM to ask for more nodes.
5. The HyperQueue worker runs on a computational node and performs tasks. It connects to the server from which it receives tasks and other commands. The worker itself also contains a scheduler that locally decides in which order the tasks will be assigned to the executing worker.
6. When a task is spawned, it starts the EVEREST task runtime. Its main task is to setup the environment and deserialize the Python function and start it.

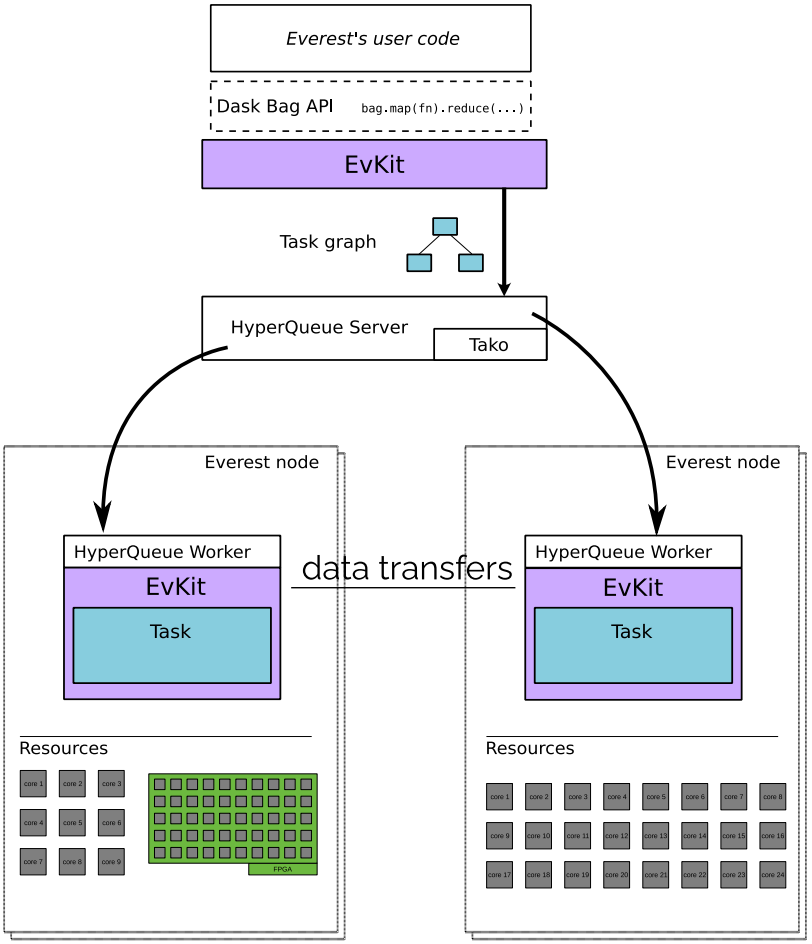

Figure 10 – EVEREST distributed runtime

3.2.5 Data Transfers

Dask supports direct peer-to-peer connections between workers like RSDS. For example, when a data object is produced on a worker 1 and a task that consumes these data is scheduled on worker 2, then worker 2 downloads the data object directly from worker 1. Such peer-to-peer communication is not yet implemented in HyperQueue. For the current version, we have written a thin layer that stores intermediate results into a file and uses a shared file system for data exchange. This is a temporary solution. Once HyperQueue will support direct data exchanges, this layer will be removed.

4 EVEREST Virtualization Extensions

The EVEREST system consists of multiple, potentially heterogeneous, nodes. To allocate and manage resources across these nodes, a virtualization environment needs to be deployed and utilized. In this context, EVEREST will be able to provide flexibility and isolation between various workloads running on the same physical node. Figure 11 shows the components running on each separate physical node, as well as the accelerators that are attached to it, i.e., GPU(s) and FPGA(s). The aim is to transparently provide the same accelerated functions to the applications inside the Virtual Machine (VM/Guest) that would be accessed when running on a physical machine. For this we use QEMU-KVM as the hypervisor running in the host and libvirtd as the agent, which exposes the relevant libvirt API to the external components, e.g. the resource allocator.

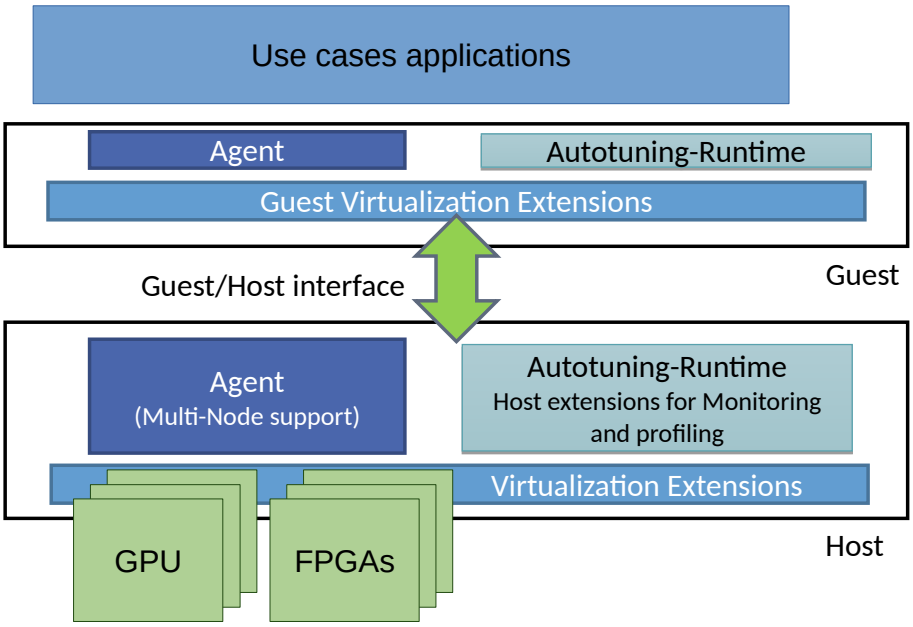

Figure 11 – General overview of the EVEREST Virtualization environment

4.1 The EVEREST Host attached FPGA Virtualization Layer

Host attached FPGAs are locally accessible to the main CPU and operating system, thus to the hypervisor. There are different ways of exposing the FPGA device to the CPU. This could be done via PCIe (typically on x86 systems) or by mapping them directly to the CPU memory (as in embedded/edge devices). For what concerns PCIe attached FPGAs, the SR-IOV technology is widely used nowadays and supports well virtualization since it is used mostly in cloud and HPC environments. Things are different for what concerns memory-mapped FPGA devices.

VOS aims to enable host connected memory-mapped FPGA virtualization extending the Linux kernel's FPGA manager subsystem [1], an architecture-independent FPGA abstraction layer that exposes a unified API to the higher layers/tools and provides a series of FPGA operations, (e.g., getting information, assigning/releasing FPGA ports, programming a bitstream to the FPGA exposed to userspace through `sysfs`, etc.). The Linux FPGA manager today does not support calls coming from a Virtual Machine (VM).

VOS is enabling FPGA manager support for VMs in a transparent way to enable tools/applications that are utilized in a native (non-virtualized) configuration to be used from inside the guest without modification. This enables operations that are available only to the host (e.g., bitstream programming) to be run in the guests. As of today, many engineers and research groups have published works on FPGA virtualization using various methods and techniques. Regarding the programming operation of the FPGA from inside a VM, most approaches partition the physical FPGA into programmable regions or virtual functions [13, 10] and then expose the programming operation to the respective area through a set of interfaces. In the EVEREST FPGA virtualization approach, we provide the guest with the programming feature of the FPGA by utilizing the mainline

Linux kernel's FPGA manager subsystem. In this way, the project framework will remain compatible with every existing or future component that uses the kernel's stack.

As a result, the EVEREST Virtualization layer design architecture is shown in Figure 12. The architecture mainly consists of two components: the guest kernel driver that receives a request and forward it, and the corresponding backend component that operates on behalf of the guest that initiated it. The solution is based on the virtio paravirtualization framework to expose a virtual device to the guest. This mechanism, with a special focus on VMs guests-hosts data transfers, has been described in more detail in [3, Section 3]. Besides forwarding the corresponding request from guest userspace eventually down to the host kernel driver, the EVEREST Virtualization layer needs to properly translate and map the guest user addresses to the respective host user ones. Consequently, userspace applications and tools can utilize the FPGA resources by mapping the corresponding memory region (using `mmap()`).

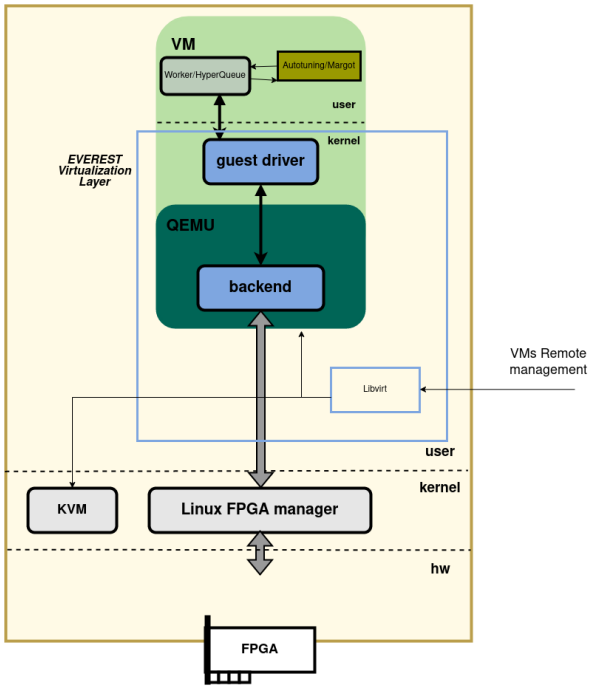

Figure 12 – EVEREST Host Connected FPGA Virtualization Layer

On top of this, the EVEREST specific libvirt virtualization API extensions enable remotely executed queries (FPGA acceleration availability, status, etc) to the hypervisor system. Libvirt is a well-known standardized and open-source interface for virtualized systems, widely used and available for many different hypervisors. Thanks to libvirt, the node where the hypervisor is installed can respond to queries about available FPGAs resources and the current status of the system. In this way, both the resource allocator, the runtime manager and the autotuner will then be able to communicate with the EVEREST Virtualization layer.

Today, VOS is working on the implementation of the guest and host extensions to support FPGA programming for the guest. This includes host and VM drivers, as well as a QEMU backend device. The first proof of concept will be available on a Xilinx MPSoC platform, based on the ARM CPU architecture.

4.2 Interactions with Autotuning and Multi-node

In general, EVEREST distributed and autotuned applications will run transparently inside VMs. The execution environment presented by the guests to the applications in a virtualized system is the same as it is when running directly on the machine. On top of this, the EVEREST virtualization layer will make sure that: 1) High-performance access to hardware accelerators (e.g. the FPGA) is granted. The Host attached FPGA Virtualization layer described above targets this objective 2) High-performance TCP/IP networking is available to the VMs. This will enable communication of guests' autotuning and multinode components with their remote counterparts. Additionally, this will make CloudFPGA available for virtualized applications.

More concretely, since the dynamic autotuning framework requires the description of the hardware to optimize the application, this description must contain all the aspects of the current execution environment that can influence the choice of the best configuration. Some examples of those aspects include the number of cores of the CPU used, its frequency, its utilization and the presence of other accelerators (GPUs, FPGAs) compatible with some kernel versions. Such information is available in the guest (each of the VMs present the hardware that has been effectively allocated to it) and can be passed to the autotuning framework with no specific interaction with the virtualization framework.

On the other hand, for what concerns the Multi-node support, libvirt is the component that will link the virtualization extensions with multinode support. The APIs exposed by libvirt will be used mainly to remotely start, stop and configure VMs.

5 Conclusions

In this deliverable we described the current status of the activities carried out in WP5 regarding the run-time and virtualization environment. We described the overall picture of the run-time and virtualization environment including how the different components (developed in EVEREST or not) interact.

Three main activities and tools have been described, while the software releases including how to use them have been included in the other parallel WP5 related Deliverables: *D5.2 - Alpha release of the run-time support* [5], *D5.3 - Alpha release of the dynamic adaptation framework* [6], and *D5.4 - Alpha release of the virtualization environment* [7].

This deliverable also shows the interaction with the compilation framework developed in WP4 and described in Deliverable *D4.2 - Intermediate report of the compilation framework* [4]. In particular, the activity links with the generation and management of multiple versions of the same kernel considering also HW modules.

Within the next period, the alpha version of the WP5 tools will be used to start integrating parts of the WP6 use cases, testing the deployment of the envisioned run-time environment on the target systems, and to increase the level of interaction with WP4.

References

[1] Fpga manager kernel documentation, 2022.

[2] Stanislav Böhm and Jakub Beránek. Runtime vs scheduler: Analyzing dask's overheads. In IEEE/ACM Workflows in Support of Large-Scale Science, WORKS@SC 2020, Atlanta, GA, USA, November 12, 2020, pages 1–8. IEEE, 2020.

[3] Everest Consortium. D3.1 - Data management techniques: initial version, 2022.

[4] Everest Consortium. D4.2 - Intermediate report of the compilation framework, 2022.

[5] Everest Consortium. D5.2 - Alpha release of the runtime support, 2022.

[6] Everest Consortium. D5.3 - Alpha release of the dynamic adaptation framework, 2022.

[7] Everest Consortium. D5.4 - Alpha release of the virtualization environment, 2022.

[8] Everest Consortium. D6.1 - Preliminary version of the EVEREST applications, 2022.

[9] Davide Gadioli, Emanuele Vitali, Gianluca Palermo, and Cristina Silvano. mARGOt: A Dynamic Autotuning Framework for Self-Aware Approximate Computing. IEEE Trans. Computers, 68(5):713–728, 2019.

[10] Joel Mandebi Mbongue, Festus Hategekimana, Danielle Tchuinkou Kwadjo, David Andrews, and Christophe Bobda. FPGAVirt: A Novel Virtualization Framework for FPGAs in the Cloud. pages 862–865, 07 2018.

[11] Christian Pilato, Stanislav Bohm, Fabien Brocheton, Jeronimo Castrillon, Riccardo Cevasco, Vojtech Cima, Radim Cmar, Dionysios Diamantopoulos, Fabrizio Ferrandi, Jan Martinovic, Gianluca Palermo, Michele Paolino, Antonio Parodi, Lorenzo Pittaluga, Daniel Raho, Francesco Regazzoni, Katerina Slaninova, and Christoph Hagleitner. Everest: A design environment for extreme-scale big data analytics on heterogeneous platforms. In 2021 Design, Automation Test in Europe Conference Exhibition (DATE), pages 1320–1325, 2021.

[12] Norman A. Rink, Immo Huismann, Adilla Susungi, Jeronimo Castrillon, Jörg Stiller, Jochen Fröhlich, and Claude Tadonki. Cfdlang: High-level code generation for high-order methods in fluid dynamics. In Proceedings of the 3rd International Workshop on Real World Domain Specific Languages (RWDSL 2018), RWDSL2018, pages 5:1–5:10, New York, NY, USA, February 2018. ACM.

[13] Tian Xia, Ye Tian, Jean-Christophe Prévotet, and Fabienne Nouvel. Ker-ONE: A new hypervisor managing FPGA reconfigurable accelerators. Journal of Systems Architecture, 98:453–467, 2019.