http://www.everest-h2020.eu

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

D5.5 — Final 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	42 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.	D5.5
Deliverable Title	Final runtime environment report
Lead Beneficiary	IT4I
Type of Deliverable	Report
Dissemination Level	Public
Due Date	31/01/2024

Document Information

Delivery Date	24/06/2024
No. pages	33
Version | Status	1.3 | Final
Responsible Person	Jakub Beránek (IT4I)
Authors	Jakub Beránek (IT4I), Roberto Rocco, Gianluca Palermo (PDM), Michele Paolino (VOS), Samuele Paone (VOS)
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
15.03.2024	1.3	Jakub Beránek (IT4I)	Final version
08.03.2024	1.2	Michele Paolino (VOS)	Added Requirements and Related works
16.02.2024	1.1	Jeronimo Castrillon (TUD)	Internal review
13.02.2024	1.0	Michele Paolino (VOS)	Complete review
12.02.2024	0.9	Roberto Rocco and Gianluca Palermo (PDM)	adaptivity part added
05.01.2024	0.7	Samuele Paone (VOS)	Virtualization extensions first draft
27.10.2023	0.0	Jakub Beránek (IT4I)	Initial draft

Quality Control

Approved by Internal Reviewer	March 15, 2024
Approved by WP Leader	March 15, 2024
Approved by Scientific Coordinator	March 15, 2024
Approved by Project Coordinator	March 16, 2024

Table of Contents

1 Executive Summary

The EVEREST project proposes a platform for implementing big data applications following a data-driven model. This document describes the final state of the EVEREST runtime environment, together with a detailed description of its individual components.

In EVEREST, the runtime environment provides adaptivity features to the application, integrating and dynamically selecting kernel variants generated by the EVEREST compilation framework (cf. Deliverable D4.5) that was created as a part of WP4, while reacting to execution environment changes, with support for task distribution across a multi-node architectures and virtualization features needed to make the Field Programmable Gate Array (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	Distributed runtime library for parallelizing and offloading kernels	Section 3.1
3	Dynamic autotuning framework and integration with the variants generated by the compiler framework	Section 3.2
4	Overview of the virtualization extensions targeting FPGAs	Section 3.3
5	Related Works	Section 4
6	Description of requirements and their fulfillment	Section 5

1.1 Structure of the Document

This document first describes the run-time environment from a high-level point of view in Section 2 to provide the context of the main components that are detailed in the remaining sections.

Section 3 describes the individual components of the runtime environment that were developed or enhanced within EVEREST. It details the architecture of the components and simple usage scenarios. Section 3.1 describes Evkit, a distributed runtime that is used to offload embarrassingly parallel computations to distributed nodes in the Traffic Simulator use cases. Section 3.2 contains an overview of mARGOt. This auto-tuning framework is integrated with Evkit and allows selecting the best FPGA kernel variants based on runtime profiling information. Finally, Section 3.3 describes the virtualization extensions developed within the project, which allow using host-attached FPGAs under a virtualized interface. Section 4 describes other work and techniques that are related to the tools used in the runtime environment. Section 5 then specifies how do the individual components and their integration within the runtime environment fulfill requirements imposed by the EVEREST project. Finally, Section 6 concludes the deliverable.

1.2 Related Documents

The content of the document should be considered in a wider view of the project as described in the project papers [19, 18]. It expands a previous deliverable *D5.1 - Intermediate runtime environment report* [6] and has a tight link with Deliverable *D5.6 - Beta release of the virtualized runtime environment* [12], which presents the actual software release of the run-time components. Moreover, we refer to Deliverable *D4.5 - Final report of the compilation framework* for the automatic variant generation part taken as input by the autotuning and dynamic adaptation framework.

2 Runtime environment overview

The EVEREST runtime environment enables kernels generated by WP4 technology to be executed in a distributed (Evkit, Section 3.1), virtualized (EVEREST SR-IOV FPGA Manager (ESFM), Section 3.3) and adaptive (mARGOt, Section 3.2) environment. When a user has a *variant* (e.g., an FPGA kernel bitstream) available, they can use the runtime environment to execute it on multiple virtualized nodes connected with a TCP/IP network. The kernel variants can be built using methodologies and tools described in WP4, either on the same cluster where the runtime environment will execute or elsewhere (in such case, the variants can be moved to the runtime cluster via common file-sharing mechanisms).

The EVEREST runtime environment can manage the execution of variants on multiple nodes of a cluster. It spawns virtual machines on the cluster nodes and executes FPGA kernel variants on their attached FPGA devices. If no device is available, the kernels can also be executed on a Central Processing Unit (CPU) directly in an adaptable way. Based on the the behavior of the executed kernels, the runtime environment can dynamically modify various parameters of the kernel, and also move its execution between the CPU and an FPGA, according to the current workload.

This is made possible by the three main components of the EVEREST runtime environment:

- Evkit, a distributed runtime library designed and developed within EVEREST. It distributes and load balances workloads supporting CPUs and FPGA accelerators. It offers client Application Programming Interfaces (APIs) for the Python programming language.
- mARGOt, a dynamic adaptation and autotuning framework that can choose the best runtime configuration according to hardware status and input features.
- EVEREST SR-IOV FPGA Manager (ESFM), an FPGA virtualization solution designed and developed during the project. It uses The Single Root I/O Virtualization (SR-IOV) to enable direct communication between the virtual machines and the FPGA kernels. It can attach and detach accelerators to Virtual Machines (VMs) flexibly.

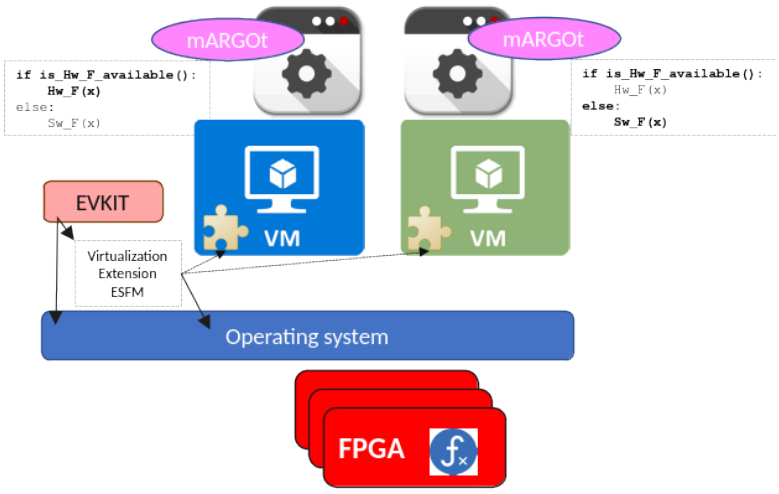

Figure 1 – EVEREST runtime components overview

The three tools mentioned above together are integrated together in the *runtime environment*, which can be seen in Figure 1. ESFM provides flexible attachment/detachment of virtual machines VMs that are hardware accelerated through FPGA devices. Evkit is deployed on these machines, and receives computational requests from a client (for example the Traffic simulator, from the Traffic modelling use-case). It then executes the requests using a kernel, either on a CPU or an FPGA. mARGOt is integrated with Evkit, and implements adaptability functions, which decide how and where will the kernel be executed. Note that these three tools can also be used independently, outside of the runtime environment.

A thoughtful description of how each of the components works is available later in section 3 of this document. Moreover, the WP5 partners prepared and published a video demonstration as part of the EVEREST webinar series. This webinar is available at: https://youtu.be/BEtelSfarZs.

The video shows the key benefits of the project runtime environment:

1. Possibility of reconfiguring hardware functions assignment according to application feedback.
2. The same binary is executed no matter if/what hardware accelerator is available.
3. Evkit can communicate with a client over the network and execute its computational requests on an attached FPGA device.

Next sections will describe each of the runtime environment technologies in detail.

3 EVEREST components of the runtime environment

The runtime environment consists of several components that can either be used as standalone tools or that can be combined and integrated together to provide a comprehensive set of features, as demonstrated by the selected EVEREST use-cases that use these components together.

3.1 EvKit

EvKit[1] is a crucial component of the EVEREST runtime system. It facilitates the distribution of computation between a *client* and a set of distributed nodes (*workers*). It provides a simple interface designed for embarrassingly parallel computations, which makes it easy for users to parallelize and distribute their computational kernels. EvKit can also facilitate execution of FPGA kernels, either using the Olympus tool for physical FPGAs (see Deliverable D4.5 [11]) or the Virtualization extensions for virtualized FPGAs (cf. Section 3.3).

Past deliverables related to WP5 (Deliverable D5.1 [6] and Deliverable D5.2 [7]) have described a previous version of EvKit that provided a Python interface based on the Dask Bag API. This interface was based on a task-based programming model, which was originally intended to be used to design an enhanced version of the Traffic simulator use case. However, we then realized that implementing the traffic simulator using this task-based programming model presents many challenges, and it is not a good fit for its architecture. The problems were caused mostly because the traffic simulator introduced more and more shared mutable global state over time (e.g. routing maps and Probabilistic Time-Dependent Routing (PTDR) probability profiles), which became difficult to manage and synchronize efficiently using a task graph program description.

Therefore, the original interface to EvKit is no longer used. We have instead decided to use a simpler interface for the final version of EvKit (described below), which is more suitable for the updated versions of traffic use-case applications (traffic simulator and traffic prediction). With no need for task-based programming, the HyperQueue task runtime is no longer needed (nor used) in the final EVEREST runtime environment.

Deliverable D5.6 [12] describes how to install EvKit and how to use it together with the Traffic Simulator use-case. This section focuses more on its general architecture.

3.1.1 EvKit Architecture

EvKit is a library for distributed computations, which consists of two components – a *client* and a *worker*. Note that this section mostly describes a concrete usage of EvKit within the Traffic simulator use case. However, the described concepts are general and can be applied to other applications.

The worker component provides a service that can execute specific computational functions (kernels). When a worker is started, it connects to a client (which is specified using two Transmission Control Protocol/Internet Protocol (TCP/IP) addresses) and waits for *computational requests* and *broadcast messages*.

Computational requests are messages from a client that are always handled by a single worker. An example of how these requests are exchanged between a client (in this case, the Traffic simulator) and multiple Evkit workers is shown in Figure 2. The client does not specify which worker should execute a request, but EvKit automatically load-balances these requests amongst all workers that are currently connected to the client. The main use case for computational requests is to parallelize the computation of a single function (kernel) invocation, potentially on a remote node. Once the worker receives the request, it executes it synchronously and returns a response.

The second kind of message that clients can send are *broadcast messages*. Figure 3 shows how broadcast messages are sent to EvKit workers. A broadcast message is handled in a fire-and-forget way, sent from the client to all currently connected workers without waiting for any response or acknowledgment. The primary use case for broadcasts is to share and synchronize some shared global state that is changing during the client's lifetime. For example, in the Traffic simulator, broadcasts are used to update the actual speeds on map

[1] https://code.it4i.cz/everest/evkit

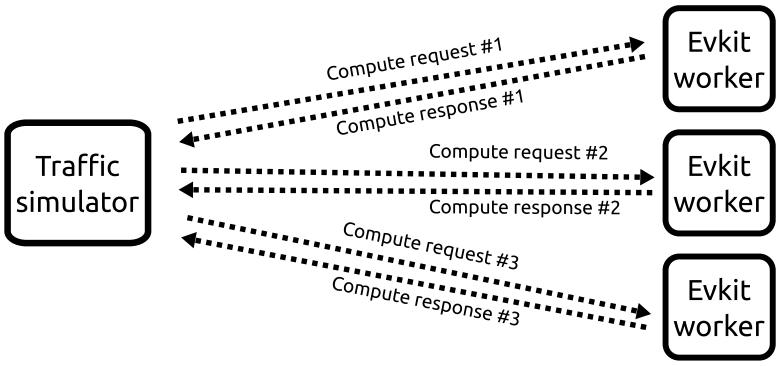

Figure 2 – Exchange of Evkit computational requests between the Traffic simulator and EvKit workers

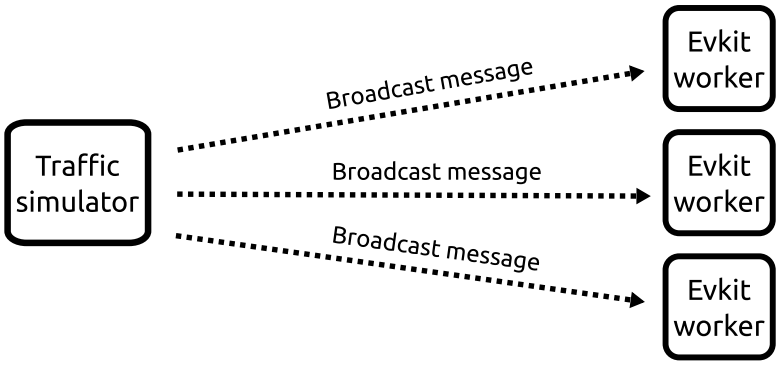

Figure 3 – Evkit broadcast messages sent from the Traffic simulator to EvKit workers

segments and to update PTDR probability profiles during the simulation so that all workers always work with the latest input data. The shared state is either sent directly within the broadcast message, or it can be loaded by the worker, for example, from a shared networked file system. Note that broadcast messages are usually not handled by kernels, but rather by the worker adapter, which might e.g. modify some shared state that will then be later passed to a kernel in response to a computational request.

To implement support for handling computational requests and broadcast messages, an adapter needs to be implemented in EvKit. The adapter handles network message (de)serialization and defines the logic that should be performed in reaction to each message.

In the case of the Traffic simulator, the adapter layer is written in `C++`. It can work in two modes, either CPU or FPGA. In the CPU mode, it directly executes the specified kernel (either a Monte Carlo PTDR simulation or alternative routing path-finding algorithm) using a provided `C++` implementation.

Figure 4 contains a diagram that depicts how EvKit can be deployed on a node with an attached FPGA. In the FPGA mode (which is available only for the Monte Carlo simulation), it uses an API for communicating with an attached FPGA, generated by the Olympus library, which was developed within EVEREST by PDM, to execute the PTDR Monte Carlo kernel on an attached FPGA. To use the FPGA mode, the user has to provide a corresponding FPGA bitstream containing the Monte Carlo kernel when starting the EvKit worker (this is described in Deliverable D6.3 [13]).

The cooperation of Evkit with outputs of WP4 is in the FPGA kernel bitstream, which is generated by the Olympus tool.

3.1.2 EvKit Client interface

Note that even though EvKit is a library, the applications that use it will always communicate with it over a network, through a defined protocol. To use EvKit, the following steps have to be performed by the user:

1. Implement an adapter using the EvKit worker component, which will specify what kernels are available and what computation should they perform. These kernels can run either on a CPU or an FPGA. The

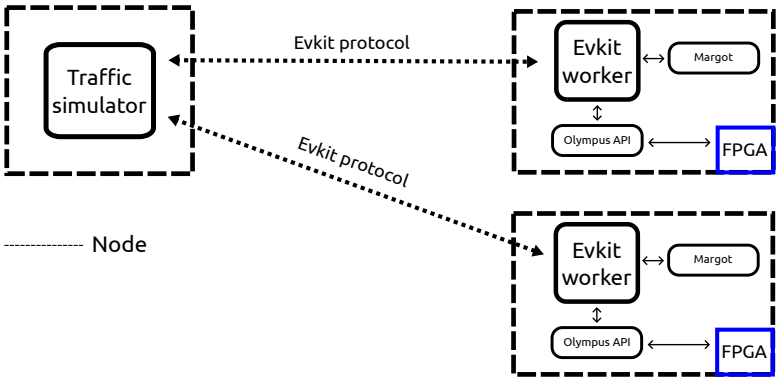

Figure 4 – Evkit deployed on a node with an FPGA

adapter can be implemented in any language that can consume and expose a C ABI-compatible interface, most likely Rust or C++.

2. Implement a client interface that will use the ZeroMQ message broker to send the corresponding computational requests or broadcast messages to the worker(s). The client can be implemented in any language that has bindings to the ZeroMQ library (for example Python) or that can consume a C ABI-compatible interface. The communication protocol is documented in Section 3.1.4, and an example of a client implementation is described in Section 3.1.3.

Once these two steps are done, users can deploy EvKit workers on computational nodes that should execute their kernels, and then start a client application which will send the computational requests to EvKit workers. To connect the client and the workers, it is enough to specify a TCP/IP hostname and two TCP/IP ports, the rest is handled automatically by EvKit. Since ZeroMQ does not require a separate deployment, it is enough to start the EvKit worker as a normal Linux process, there are no additional runtime dependencies or services needed for it to run.

3.1.3 Using EvKit

As described above, the concrete EvKit client interface depends on the programming language and use-case of the specific application that uses EvKit. Here is an example interface that is implemented within the Traffic simulator use-case[2], which offloads the PTDR Monte Carlo simulation kernel to an FPGA attached to a node that with an EvKit worker:

```
def calculate_routes(evkit_client, route_possibilities):
    messages = [Message(kind="monte-carlo", data={
        "routes": routes,
        "frequency": vehicle.frequency.total_seconds(),
        "departure_time": vehicle.departure_time
    }) for (vehicle, routes) in route_possibilities]

    shortest_routes = evkit_client.compute(messages)
    return shortest_routes
```

The client serializes the data into JavaScript Object Notation (JSON) and sends it as a multi-part ZeroMQ message. The message broker then load-balances the request onto one of the currently connected workers, which executes the kernel either on a CPU or on an FPGA, and returns the result (in this case a set of simulated route durations, one for each vehicle).

[2] https://github.com/It4innovations/ruth

3.1.4 EvKit Implementation

The EvKit worker library is implemented in Rust, which (amongst many other benefits) makes it trivial to deploy – it is compiled as a single, statically linked binary without any external dependencies, which can be easily deployed on a node in a cluster simply by starting a single executable. The communication between the client and the worker(s) is performed using the ZeroMQ[3] message broker, which works on top of the TCP/IP protocol.

The client part of EvKit is a relatively thin wrapper over ZeroMQ and can be implemented in various languages based on the corresponding use case. For the traffic simulator, the client part is implemented in Python[4], but any programming language that has bindings to ZeroMQ (or can bind to a C library) can implement an EvKit client. Workers have to be able to connect to the client using TCP/IP. The client should provide two separate TCP/IP ports (one for computational requests and another one for broadcast messages) for the workers to connect to.

To communicate with an EvKit worker, the client will need to use the EvKit communication protocol for sending computational or broadcast messages. The protocol is documented in the EvKit repository[5], but you can also find a short description of how does it work below.

The protocol is different for computational requests and broadcast messages, since these are also exchanged over separate ZeroMQ channels (running on separate TCP/IP ports).

Computational requests For sending computational requests, clients should create a `DEALER` ZeroMQ socket, and connect to the corresponding worker port. Then they can start sending computational requests, which should be multi-part ZeroMQ messages with the following message parts (delimited sequences of bytes):

1. Message ID. This is an opaque binary blob that identifiers the message. Practically, it can be for example a 4-byte integer that is unique for each message. The client uses this ID to match responses from the worker with its previously sent requests.
2. Message Kind. A UTF-8 encoded name of the kernel function that should be computed. The worker has to support this message, if it's unknown, it will result in an error.
3. Serialized kernel input. These bytes contain the actual input for the kernel that should be executed. The data format is completely up to the client, it is not interpreted by EvKit directly. Instead, the EvKit adapter implemented by the user will know how to deserialize and evaluate this data. It can be for example function/kernel parameters serialized as JSON or MsgPack.

Once the worker receives this message, it will invoke the corresponding kernel and return a response, which can be either a success or a failure.

In case of success, the worker will respond with a message containing the following three message parts:

1. Message ID. This helps the client correlate the response with the original request.
2. UTF-8 string with the content `"ok"`.
3. Serialized kernel output. These bytes contain the serialized output of the corresponding adapter that executed the given kernel. These bytes are again opaque to EvKit, they have to be interpreted by the client.

In case of failure, the worker will respond with a message containing the following two message parts:

1. Message ID. Same as for the success case.
2. UTF-8 string containing the error message.

[3] https://zeromq.org/

[4] https://github.com/it4innovations/ruth/blob/5dfb1801421a3a596bfa5c34e33b4db769508351/ruth/zeromq/src/client.py#L24

[5] https://code.it4i.cz/everest/evkit/blob/55511ad35f0d2f0e6804f6a0a8b7934c480868c4/crates/worker/src/lib.rs#L96

Broadcast requests For sending broadcast requests, clients should create a `PUB` ZeroMQ socket, and connect to the corresponding worker port. Then they can start sending broadcast requests, which should be multi-part ZeroMQ messages with the following message parts (delimited sequences of bytes):

1. Message kind, prefixed with the subscription topic. The worker side of EvKit creates a SUB ZeroMQ socket which subscribes to a given topic (broadcast channel), which should be selected by the user when implementing the worker adapter. This message should start with that topic, and it should be followed by an identifier that specifies what kind of message topic this is. For example, if the subscription topic was *"my_everest_app"*, and the broadcast message was sending some data update to all workers, the message kind could be e.g. *"my_everest_appdata_update"*.
2. Serialized input. These bytes contain the actual input for a worker adapter function that should be executed in response to the broadcast message. This data is opaque to EvKit, the adapter has to be able to interpret it.

Since broadcast message are designed to be executed in a "fire-and-forget" way, there is no message produced by EvKit in response to broadcast messages.

3.2 mARGOt

The mARGOt autotuning framework is the runtime environment component that operates closest to the application, ensuring its adaptation to a changing environment. In Deliverable D5.1 [6], we explored the framework's elements, their purpose and interactions. In the following sections, we analyse the current status of the framework after its evolution that helped us better fit the applications' and other runtime environment actors' needs. The following content is an updated version of the Deliverable D5.1 [6]. In particular, we introduced a new knowledge analysis and review system within the runtime observer and a new component that automates instance handling for multiple programming language support.

3.2.1 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, which monitors the execution and decides when there is a need for model retraining.

We can represent the entire framework as in Figure 5, highlighting the components' interactions. The interaction between the components uses telemetry protocols (MQTT), using the JSON format for data exchange.

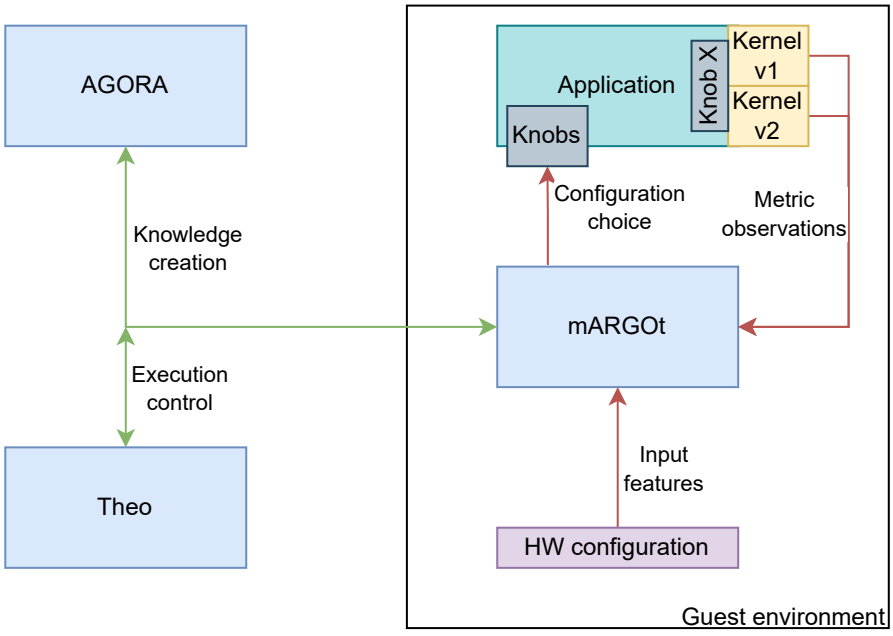

Figure 5 – Structure of the mARGOt autotuning framework.

The *mARGOt autotuning library* performs runtime optimisation by choosing the best configuration among the ones offered by the application. It consists of a C++ library 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 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. Input features, variables of the code observed by mARGOt, reflect the latter. The application knowledge is the foundation for the best knob value selection process. The knowledge consists of a collection of configurations with their expected effect on the metrics of interest. Given an application, producing its knowledge is a non-trivial task yet mandatory for autotuning. The mARGOt library alone does not provide methods to extract it, as its management is the duty of the other two framework components (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 connects to the application via the MQTT telemetry protocol. Its focus is to enable the learning of the application knowledge at runtime, but it can also extract the knowledge during profiling. AGORA operates from application launch time, suggesting the configurations to run and gathering 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 optimised way. The combination of AGORA and mARGOt is enough to provide dynamic adaptivity but cannot ensure an effective optimisation during the entire execution: the learning process done by AGORA is performed only once (when there is no prior knowledge). This limitation means that a change in the execution environment that radically affects the application's behaviour 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 observe the interactions between mARGOT and AGORA by listening on the same MQTT channel. Whenever it notices something has changed, it restarts the AGORA knowledge-learning procedure and combines the results obtained with the previous ones.

We can summarise the interactions between the components as in the sequence diagram shown in Figure 6. The diagram describes the sequence of messages 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 that can serve multiple applications at the same time. After receiving the welcome message, AGORA suggests application configurations to execute (and monitor) to build the application knowledge. Once AGORA correctly creates the application knowledge, AGORA broadcasts this info to the local instance of mARGOt and starts selecting the best configuration on its own (autonomously). Theo also receives the application knowledge and stores it for future reference. During its autonomous execution, the application sends monitoring data to AGORA and Theo.

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

Alongside monitoring the quality of the knowledge, Theo analyses it to extract valuable information for other components of the EVEREST runtime environment. In particular, the knowledge contains information about how satisfied the application is with the current resource allocation: this is a valuable asset to achieve better resource allocation. This information is present in the application knowledge. Since Theo has access to the application knowledge, we integrated a mechanism to compute the *happiness* of the application with the current execution resources. The happiness value represents the ratio between the actual value of the optimisation problem (obtained using the available resources) and the best one found inside the knowledge (considering only feasible configurations). A higher happiness value indicates a more optimised solution.

At each iteration, Theo checks the hardware configuration through the information in the observations and fetches the knowledge to extract configurations with different resources from the current one. For each configuration, it computes the difference between the happiness and the one achievable with other resources

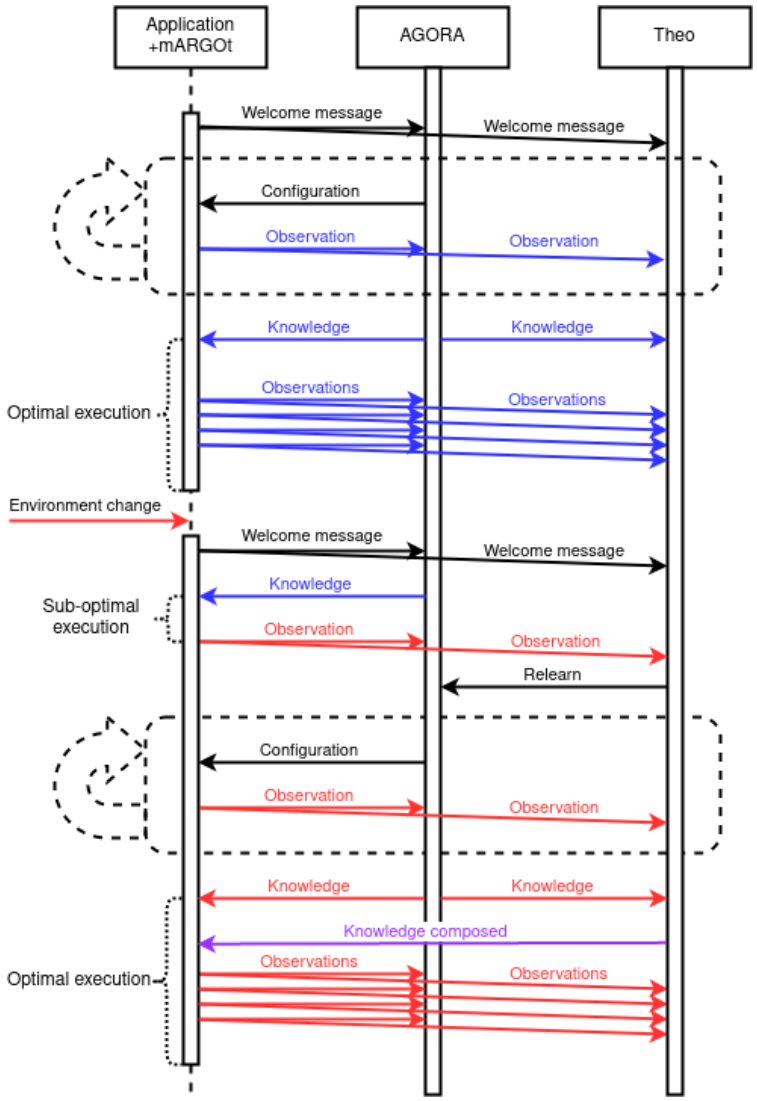

Figure 6 – Typical flow of execution of the optimisation process. The red and blue arrows identify results obtained in different execution environments. The purple arrow indicates that the knowledge contains data about the blue and red configurations.

available. It stores all this information in a JSON file so a resource manager can use it for resource allocation. We used this mechanism in the last scenario of the webinar [1], showing how changing the resources would also produce feedback in the happiness hints produced by Theo.

3.2.2 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 7 contains a C++ code snippet of an application using the mARGOt library interface to optimise its execution. The application first calls an initialisation 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 changed). After those steps, it starts the monitors (line 24), executes the code to be optimised (line 25), stops the monitors that are computing all the needed metrics (lines 26-27), and prints information about the execution (line 29).

The code snippet shown in Figure 7 can only work if we provide a compatible configuration file during the compilation. The configuration file includes information about the different variables involved in the optimisation process and settings for the other parts of the framework. The configuration file specifies knobs variables,

```
int main() {
    margot::init();
    function_ptr_t do_work_array[2];
    do_work_array[1] = do_work_hw_boosted;
    do_work_array[0] = do_work_sw;
    int version = 0;
    int hw_available = 0;
    int data_feature = 0;
    srand(time(NULL));
    int repetitions = 1000;
    for (int i = 0; i<repetitions; ++i) {
        data_feature = extract_data_feature();
        const char* presence = std::getenv("HW_MODULE_AVAILABLE");
        if(presence)
            hw_available = 1;
        else
            hw_available = 0;
        //check if void configuration is different wrt the previous one
        if (margot::block1::update(data_feature, hw_available, version)) {
            margot::block1::context().manager.configuration_applied();
            std::cout << "< < < CHANGE APPLIED > > >" << std::endl;
        }
        //monitors wrap the autotuned function
        margot::block1::start_monitors();
        int error = do_work_array[version](data_feature, hw_available);
        margot::block1::stop_monitors();
        margot::block1::push_custom_monitor_values(error);
        //print values to file
        margot::block1::log();
    }
}
```

Figure 7 – Code snippet from an application using the mARGOt interface.

metrics observed (together with the monitors which will observe them), features of the inputs and the optimisation 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 Deliverable D5.3 [8].

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 [19] limit the project's target to C++ applications, the use case developments Deliverable D6.1 [9] added the need to support other languages, particularly 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 optimisation and will then send the optimised 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 needs to develop an additional interface to extract data from the communication medium, and then it is possible to use the optimised values provided by mARGOt. Figure 8 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 optimise the Adam component, whose values will be the same as the original application. The application then receives the optimisation results and 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 differs), it introduces overheads due to the communication over the MQTT medium, so it is better to use it only when strictly necessary.

The Adam component use is mandatory to support applications not written in C++, but it also allows the

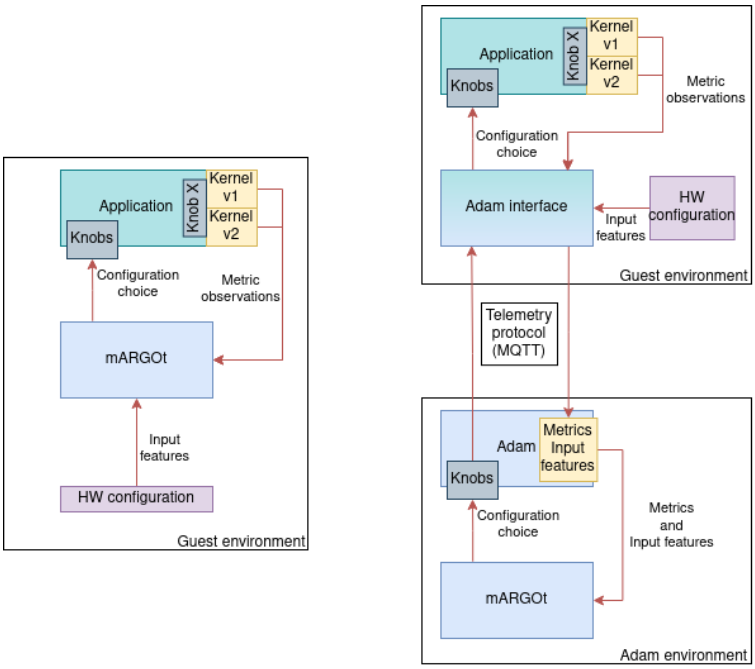

Figure 8 – Comparison between normal integration (on the left) and Adam integration (on the right).

splitting of the control logic of the mARGOt library from the application. Each application instance requires an Adam counterpart, and the pairing happens through identifiers. Creating Adam instances, compiling and pairing them with the respective applications is mechanical and easy to automate. For this reason, we developed the Brian broker to automatically initialize Adam instances upon need. Using the Brian component, the application does not directly pair with Adam instances but contacts the broker providing its configuration file. Brian handles the execution, pairing, and destruction of Adam instances, freeing the user from all these duties and simplifying the deployment of mARGOt-optimised non-C++ applications. The use of Brian does not directly impact any component of the mARGOt autotuning framework, except for the Adam interface exposed to the application: it must include APIs to contact the broker and retrieve the identifier to use for pairing.

The typical use of Brian may introduce overheads in the execution. Upon first execution, the broker must create and compile the code of the Adam instance while the application just waits for the configuration to run. To solve this issue, we introduced the possibility of pre-allocating Adam instances inside the Brian broker so that applications do not have to wait for compilation but can leverage an already initialized instance. If no pre-allocated instance is available, Brian operates as usual, compiling the code of the Adam instance.

3.2.3 Multiple-version Kernel Integration

One of the tuning parameters of interest for exploring within the EVEREST project is the selection of the most suitable 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 hardware point of view and the accelerator point of view (availability of FPGAs or Graphics Processing Units (GPUs)). This structure makes it possible to have versions prepared considering the exploitation of the peculiar characteristics of the architecture used for the execution;
- In the case of FPGA-enabled nodes, we can optimise the modules/versions deployable on the FPGA considering different objectives, e.g. different points from a Pareto curve for performance-resource usage. This choice 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 accelerators available or deployable is known only at runtime;

- The implemented versions can consider different utilisation scenarios regarding the data to be processed and how to transfer them to the accelerators, e.g. single data processing vs batch execution. All this information is available only at runtime;
- The different versions require different values of compile-time parameters, allowing to optimise those values (and the related code) without the need for a dynamic recompilation;

We can use the mARGOt autotuning framework on code featuring alternative implementations to choose the most promising version for each execution. Figure 7 shows an example implementation: in lines 4 and 5, we initialise an array with two alternative versions of the same function `do_work`, and the one to execute is chosen depending on a knob variable (version, see line 25). While this pattern is quite general, it requires the functions to provide the same signature.

To ease the integration of mARGOt into a multi-versioned kernel, we developed a set of scripts that works at compilation time and assists in the generation of the versions, their collection and the integration with the mARGOt library. The scripts generate the versions' interface 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. Figure 9 shows the entire flow.

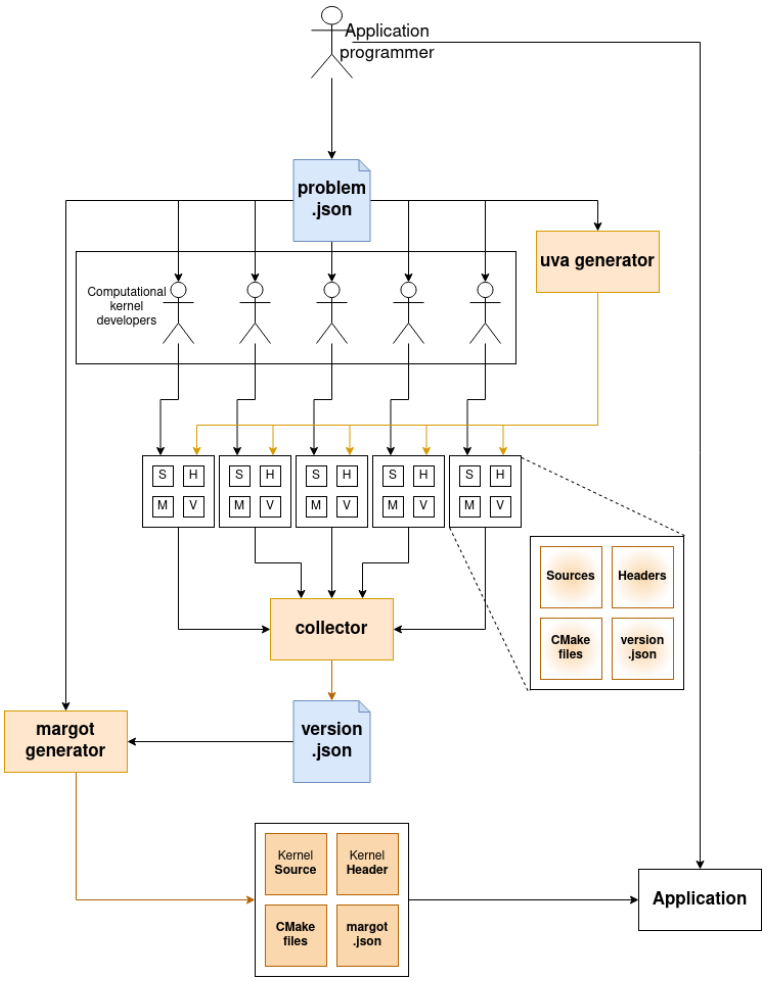

Figure 9 – mARGOt generator flow for multi-versioned kernels.

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 10). The file must also collect information about eventual data features present in the input values. Version developers can use this file to produce an alternative implementation of the problem but also feed it to a generator script (uva generator) to pre-generate the interface of the versions. Each produced version must feature a `version.json` file containing information about the function it implements. All these configuration files are collected into a single one (see

Figure 11), 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). We can use this collective file with a generator script that produces the C++ sources with mARGOt that can choose the best version. This last script will also generate the CMake files needed to compile and use the generated sources correctly. Deliverable D5.3 [8] contains additional details on the flow and the scripts.

```
{
  "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 10 – Example of a `problem.json` file to manage different versions of the `inverse_helmholtz` kernel. It describes the interface 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 11 – 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 9 supposes human-generated versions of a kernel, the flow is general and can receive versions automatically generated from higher-level tools. As discussed in Deliverable D4.2 [5] and Deliverable D4.5, the kernel compiler can generate multiple variants from the high-level DSL for tensor expressions. Currently, the variants differ in their internal implementation (e.g., different tensor factorisations, loop nestings or different loop optimisations). Selecting a particular variant is non-trivial [21] for which the mARGOt autotuning approach is very useful. This assumption is especially true when generating pure software variants and code variants that must be processed using High-Level Synthesis tools to create hardware accelerators [23]. The compiler framework described in Deliverable D4.5 can generate the files and the interfaces

as specified in Figure 9. The mechanisms and interfaces are now in place and can be used to implement more intelligent variant selection.

3.3 The EVEREST Virtualization Extensions

Previously, in Deliverable D5.1 [6], VOS demonstrated FPGA virtualization activities focusing SoC attached FPGA. In this deliverable however, the focus is on Peripheral Component Interconnect Express (PCIe) FPGAs. In general, the work done aims at simplifying the use of FPGA in virtualized environments, while keeping performance and overhead bare metal. As a result, the components (FPGA virtualization framework with its drivers and ESFM) and functions (QEMU pause) shown in this section go in the direction of FPGA-cloudification.

In fact, the management of FPGA PCIe devices in cloud/High-Performance Computing (HPC) environments can be challenging, especially when dealing with multiple VMs and frequent reconfigurations. The Single Root I/O Virtualization (SR-IOV) technology helps, by partitioning the device Physical Function (PF) into multiple Virtual Functions (Virtual Functions (VFs)) and thus allowing multiple VMs to share the same physical device. However, SR-IOV is not always straightforward to use and can lead to complex scenarios, especially when there is the need to change the VF configuration on the fly. In fact, SR-IOV requires the reset of all the VFs before changing one of them.

The EVEREST Virtualization Framework was designed and built to address these challenges. More in details, the ESFM is the key component of such framework that leverages the SR-IOV support of the Xilinx QDMA IP to automate the creation, attachment, detachment and reconfiguration of accelerators to different VMs. It improves flexibility and usability of VFs with a set of scripts that automates several operations, as well as with the implementation of a novel pause functionality that minimizes the guests accelerator downtime by enabling the VFs reconfiguration without detaching them from the guest.

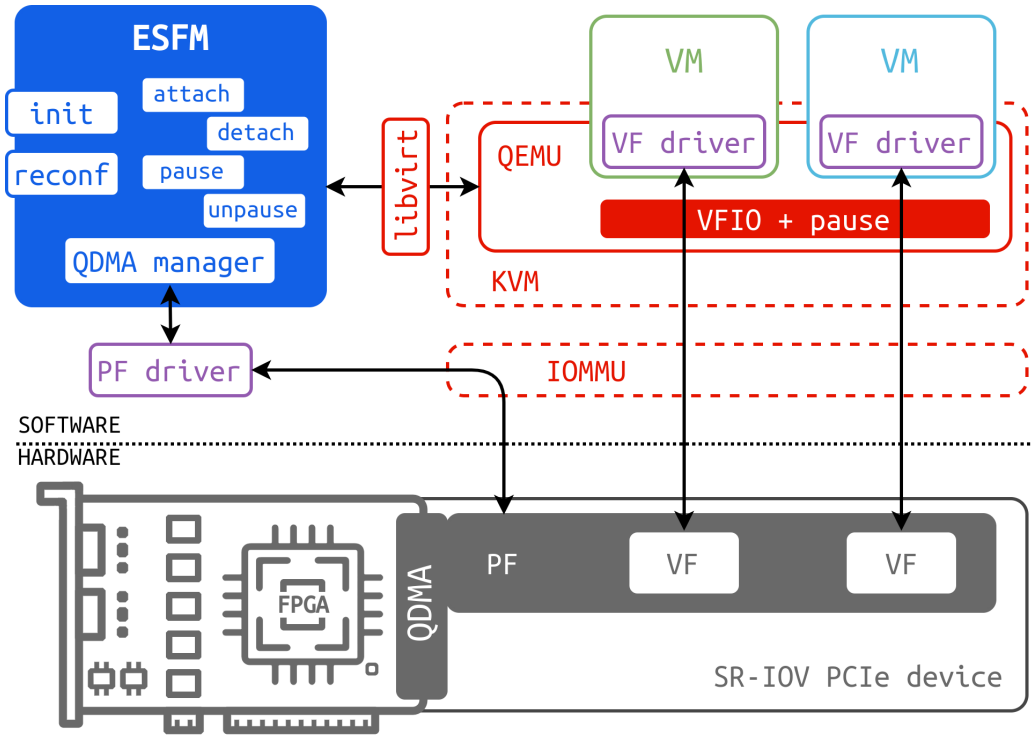

Figure 12 – An overview of the the EVEREST Virtualization Framework and its components

More in details, Figure 12 shows an overview of the EVEREST virtualization framework architecture, together with its main components:

- Everest SR-IOV FPGA Manager (ESFM)
- QDMA Physical Function (PF) and VF Drivers & Manager
- QEMU extensions for the pause function

In the remaining part of this section, each of these components will be described in detail.

3.3.1 Everest SR-IOV FPGA Manager (ESFM)

ESFM is a bash script that handles various tasks including FPGA bitstream flashing, QDMA driver interactions, and VF attachment, detachment and pause. *ESFM* can be run with one of the following arguments:

- init: initialize the FPGA device the first time is connected to the host, eventually flashing the bitstream. The init process involves a recursive search for all the VFs associated with the PFs of the device, detaching them from the VM, and removing the PF from the Peripheral Component Interconnect (PCI) bus unloading its driver. Afterwards, the bitstream can be flashed to the FPGA, the PF is discovered and configured. The device is now ready to be used.
- reconf: change the number of VFs of an FPGA device. After the initial number of VFs is set using the init option, the configuration may need to be changed using the reconf command. This could involve increasing or reducing the number of VFs with the optional reflashing of the FPGA bitstream to change the hardware configuration.
- attach: With the attach command, a VF is attached to the VM, making it available to the guest, using the vfio-pci as the backend driver on the host and the qdma-vf driver on the VM, leaving to QEMU the management of the passthrough operations. On the host system, the qdma-pf driver provides a sysfs interface to configure and control various parameters of the PF exposed by the QDMA IP.
- detach: detach a VF from the VM it is attached to.
- list_vfs: list the available VFs in the system.
- list_vf_vm: lists for every VF currently in use, what is the VM that is using it.

As far as the interactions with other components is concerned, attaching and detaching VF devices to VMs is automated through *ESFM* and facilitated through the use of *virsh*, which serves as the primary interface for managing guest domains with libvirt. Interactions with QDMA driver are mediated with the QDMA manager that will be further explained in the next section.

3.3.2 QDMA Drivers & Manager

To maximize performance, the implemented virtualization framework also involves the PCI passthrough technology. Device passthrough is about attaching a device to a given guest operating system so that the device can be used exclusively by that guest. In this way, the device (or its partition) cannot be shared, and the VM attached to it can use it natively without performance overhead. Device passthrough is made possible thanks to a driver inside the Linux kernel called VFIO driver; such driver is attached in the host to the passed through device (VFIO device) instead of its original driver. On the other hand, the guest sees that device and uses the device's original driver.

All the part related to SR-IOV support and VFIO attachment in the EVEREST Virtualization Framework is handled by the QDMA manager (running inside ESFM) and the QDMA drivers (running in the kernel of the host and guests). In particular, the QDMA manager component has been designed and developed to take into account all the operations needed to leverage the passthrough technology, thus simplifying the management of the drivers, offering various functionalities such as unbinding a PCI device from its driver, binding the VFIO driver to it, removing the device and its related VFs from the PCI bus, rescanning the PCI bus to detect new devices, etc. On the other hand, the Xilinx QDMA driver has been modified and wrapped via a tailor-made API. The application developer needs to use the QDMA to access the kernels (IPs) inside the FPGA; hence, the provided API is helpful because it allows the application developer to work with higher-level functionalities.

For what concerns the driver modification at the driver level, the extension is related to the ability to detach a VF dynamically while the VM is executing an application using one of the FPGA kernels (IPs). In fact, the Xilinx drivers don't support such flexibility in the detachment and trigger a kernel panic in the guest every time a VF is detached while someone is still using the FPGA (for instance another VM attached to a completely unrelated VF).

#VF	Detach/Attach		Pause/Unpause		Overhead	
	avg ms	σ	avg ms	σ	%	ms/VF
1	4151	40	4068	56	-2.00	-83
4	12988	183	12665	171	-2.49	-80
10	31129	497	30285	505	-2.71	-84

Table 1 – VF detach-attach vs pause-unpause overhead, AVG of 100 runs

To overcome this problem, the EVEREST QDMA driver extensions leverage the *sysfs* used by the QDMA driver and an internal message mechanism called *mailbox* to let the host userspace (ESFM in this case) know when and which kernel is in use before actually detaching the VF from the VM. This *sysfs* communication mechanism based on QDMA drivers extensions is described more in detail in Deliverable D3.2.

3.3.3 QEMU Pause Extension

The main drawback of SR-IOV is that it requires a reset of all the VFs before changing their number (for instance when booting a new VM). This would trigger a general interruption of service that is not acceptable in most cases. To overcome this limitation, a novel QEMU device pause functionality has been developed to provide a way to temporarily detach a VFIO device connected to a VM from the host side only. In this way, the removal of the device from the guest is avoided without causing interruption of the services due to driver unloading.

This solution acts only on the host system, leaving complete flexibility on the guest that can use *unmodified* drivers. The pause procedure of a (VFIO) device consists of 3 main steps:

1. save the PCI device config space including emulated config space and Message Signaled Interrupts (MSIs) state to be restored in the unpause;
2. perform unregister operations on the PCI device, including deletion of PCI memory subregions, optional device ROM, and interrupt bits;
3. perform unregister operations on the VFIO device, including un-registration of device request and error notifier (requests from the guest to the device will be ignored), disabling and teardown of interrupts, deletion of VFIO device BARs and exiting from IOMMU group (this is needed to detach the device from the host completely).

To better understand the advantages of the *pause functionality*, hereafter it is analyzed the benchmarks related to the evaluation of this new functionality. In particular, we are going to compare the standard attach/detach overhead with the optimized pause functionality operations we developed (pause/unpause).

Before diving into the results examination, it is important to know the hardware and software configuration used to obtain these results:

- *Hardware Configuration*: it has been used the *Xilinx Vivado 2022.1* to generate the bitstream for the *Xilinx Alveo U55C* target
- *Software Configuration*: the host system used is a 32 cores, 128GB RAM x86_64 server running Red Hat Enterprise (RHEL) 8.7 with Linux Kernel 4.18, hosting the Alveo FPGA card, managed by the Xilinx Runtime Library (XRT) 2.13.466 branch 2022.1. A modified version of QEMU (7.1.0) to add the pause functionality. The VM used is a 2 cores, 2GB RAM x86_64 Ubuntu 22-04 with Linux kernel 5.15.

Table 1 shows the average delays on 100 runs for both attach/detach and pause/unpause when operating on 1, 4, and 10 VFs on as many VMs. The overhead values show that there is a minimal reduction in the delays from 2% to 2.71% with a reducing time of around 80ms per VF in all the scenarios. The delays are not linearly increasing with the number of VFs, meaning that some operations are not dependent on it.

	1 VF		4 VFs		10 VFs	
Step	**D/A**	**P/U**	**D/A**	**P/U**	**D/A**	**P/U**
	ms	ms	ms	ms	ms	ms
1. rescan	138	138	144	141	139	139
2. remove VF	1265	1273	5417	5505	14360	13878
3. change #VF	1256	1295	1460	1412	1817	1730
4. add VF	1472	1346	5946	5653	15042	14448
total	**4131**	**4052**	**12967**	**12711**	**31358**	**30195**

Table 2 – Timings of VF detach-attach and pause-unpause operations

To clarify more the results, the duration of the different macro-operations performed in the two cases for a single run with 1, 4, and 10 VFs are inspected, showing the values in Table 2. The removal (detach or pause) of the VF in step 2 takes on average the same time in both scenarios, although the pause skips some device removal operations performed by detach but spends some more time allocating and saving the device configuration data structures. Step 3 shows the delay for changing the number of VFs of a PF and it is quite similar in both cases, slightly increasing with the number of VF involved. In step 4, it is noticeable the main gain of the unpause operation with respect to the attach. This is justified by the fact that the former skips some of the realize operations of the device (because it does not detach it from the guest side), copying back the device configuration data instead.

This benchmark analysis demonstrated that the *pause functionality* can be used as a suitable alternative to device removal, as it incurs no significant overhead and can even provide a slight performance improvement of a few percentage points, with the main advantage of avoiding device removals on the guest side.

4 Related Works

4.1 Evkit

There are existing tools that can be used to distribute embarrassingly parallel computation, for example messaging brokers like ZeroMQ[6] or Celery[7]. These do not directly allow executing kernels on an FPGA, nor switching between a CPU and an FPGA variant of a function (kernel). EvKit leverages the ZeroMQ messaging broker, which allows creating arbitrary distributed communication patterns for networked applications, and uses it to build a fault-tolerant distributed service that can parallelize the computation of kernels amongst multiple nodes, execute kernels on an FPGA device, and also manage the sharing of global state between the client application and the Evkit worker nodes.

4.2 mARGOt

The literature contains many examples of autotuning frameworks. We can divide them into categories according to the moment they make decisions and how they gather the information needed. A first set of works explores a vast space at compile time, deciding the best configuration for the knobs: we refer to those efforts as static autotuners since their decision does not change at runtime. In the literature, we can find many examples of static autotuners like ATune-IL [22], OpenTuner [2], and the ATF framework [20]. Among dynamic autotuners (i.e., those that perform runtime decisions), we can distinguish between those that act according to pre-defined knowledge of the system and those that work without it, learning it during the execution. The literature features many examples for both categories, with the first containing efforts like Petabricks [4] and Capri [24]; on the other hand, the second contains [15, 17, 14], able to understand the effect of the decisions on the observed metrics.

The requirements of the EVEREST project proposed in Section 5 are not completely compatible with any available frameworks from the literature. While dynamic autotuners can manage dynamic scenarios, one of the main requirements for the EVEREST runtime, the full set of requirements is not satisfied by any of the listed approaches. We thus decided to extend the work presented in [14] with additional features since it was fully under our control. Among its additions, the mARGOt EVEREST extension allows to manage HW/SW versions as they are simple tunable knobs, and it provides features such as multi-language support, continuous runtime observation, interaction with the virtualisation engine, decoupling between execution and decision-making, and runtime update of the application knowledge.

4.3 EVEREST SR-IOV FPGA Manager (ESFM)

Different approaches and solutions exist to enable virtual machines access to FPGAs. One of them, virtioFPGA, was developed in EVEREST and presented in Deliverable D5.1 [6] and was targeting Arm devices and System on Chip (SoC) attached FPGAs. Other solutions (vFPGAmanager [3], FPGAVirt [16], pvFPGA [25]) involve custom hardware development that makes them hard to deploy and maintain because of their non-standard interfaces.

Conversely, the EVEREST SR-IOV FPGA Manager (ESFM) has been build on top of SR-IOV, a standardized solution for PCIe devices virtualization. In addition, ESFM specifically takes into account the need for modifying the VF configuration (typical of reconfigurable devices like FPGAs), and is generic enough to handle various types of VFs, whereas most current solutions for SR-IOV either focus solely on network/storage virtualization or lack of the VF management flexibility provided by ESFM.

[6] https://zeromq.org/
[7] https://docs.celeryq.dev/en/stable/

5 Assessment of SDK Requirements

This section describes how the components of the runtime environment and the environment as a whole satisfy the requirements described in Deliverable D2.5 [10].

5.1 Evaluation of Requirements on Orchestration Large Application Flows (DAGs)

5.1.1 REQ 2.1 - Front-end for EVEREST Applications

- **Priority** - Must have
- **Notes** - Specific interface for distribution of tasks in traffic simulator. Framework on top of HyperTools for easy use-cases driven development.
- **Assessment** As was mentioned in Section 3.1, the task-based interface was not used for the traffic simulator, since it was not able to properly capture the semantics of global shared mutable state. Instead, the Evkit tool was developed, which allows the distribution of computational kernels using a simple API that can distribute a batch of kernel computations amongst remote nodes.

5.1.2 REQ 2.2 - Dynamic data sharing between DAG tasks

- **Priority** - Should Have
- **Notes** - Extend framework to support spawning a dynamic service-like tasks that may serve data independently on fixed dependencies defined in a task graph. In some frameworks it is known as actor model (e.g. in Ray).
- **Assessment** Since the programming task model was not used for computation distribution, dynamic data sharing was not needed. Data required for kernel computational requests are attached to the requests themselves, they are transferred over the TCP/IP protocol using the ZeroMQ message broker. This is implemented in Evkit.

5.1.3 REQ 2.3 - API for communication with virtualization environment

- **Priority** - Must Have
- **Notes** - If a dynamic reconfiguration of the environment is possible, the protocol have to be able to notify the scheduler about the changes. The goal is to establish a way of communication between the scheduler and the environment to exchange all important properties and constraints.
- **Assessment** Evkit communicates with mARGOt and ESFM to dynamically change whether a kernel is executed on a CPU or an FPGA.

5.2 Evaluation of Requirements on Autotuning and Virtualized Environment

5.2.1 REQ 5.1 – Application Knobs

- **Priority** - Must Have

- **Notes** - The autotuning framework should have access to the application knobs. The access should be provided by means of the DSL or by the application itself. The dynamic autotuning framework is only a decision engine.
- **Assessment** - The application must use the mARGOt autotuner API to benefit from its presence. Among the API calls, the function `update` allows to set the best value of the knob passed as a parameter according to the input features and the knowledge available. Examples of usage of the mARGOt API are available within the SDK repository.

5.2.2 REQ 5.2 – Adaptive autotuning

- **Priority** - Must have
- **Notes** - The dynamic autotuning framework should be able to adapt depending on decisions taken in the virtualized environment. Changes in the available resources should trigger the application adaptation.
- **Assessment** - By modelling the execution environment as an external feature that affects the decision procedure, the mARGOt autotuning framework can fetch for changes on available resources and take appropriate decisions. This behaviour is also featured within the 12th EVEREST webinar, where the availability of the FPGA was changing, even at runtime.

5.2.3 REQ 5.3 - Integration with runtime

- **Priority** - Could have
- **Notes** - The dynamic autotuning framework should be able to interact with the virtualised environment. Given the knowledge of the application, the dynamic adaptation framework can provide hints to the virtual manager to steer decisions
- **Assessment** - The interaction between the autotuning framework and the virtualised environment happens through the fetches for available resources. On the other hand, it is possible to set up the mARGOt autotuning framework to produce a value representing its satisfaction with the current resources: this information can be valuable for the virtual manager to select the best assignment of resources. We show all these interactions within the 12th EVEREST webinar.

5.2.4 REQ 5.4 – Autotuning and optimization

- **Priority** - Must have
- **Notes** - The dynamic autotuning framework should manage the software knobs exposed by the application by autonomously selecting a near-optimal configuration. The adaptation should be triggered automatically and not by hand.
- **Assessment** - Through the use of the `update` function, the autotuning framework can change the value of the knobs following an optimisation problem provided by the user. The adaptation happens when the application calls the function. The 12th EVEREST webinar contains multiple examples of this behaviour.

5.2.5 REQ 5.5 – Variant Selection

- **Priority** - Must Have
- **Notes** - The dynamic autotuning framework should be able to manage code variants selection. Code variants should be managed as for the parameters of the application.

- **Assessment** - mARGOt implemented the variant selection as a discrete application parameter. Each variant is associated with an integer parameter whose value represents the variant to be executed. An example of this can be seen within the 12th EVEREST Webinar where two variants (with and without FPGA accelerator) are managed. A verification procedure for the requirement is also available within the SDK repository.

5.2.6 REQ 5.6 - Design and deploy-time information

- **Priority** - Must Have
- **Notes** - The dynamic autotuning framework should be able to make a decision based on knowledge collected at design time or at deployment time. The knowledge for making the decision can be directly injected by some analysis of the compilation flow or extracted by online profiling of the kernel.
- **Assessment** - The mARGOt autotuning framework features the AGORA component, which is responsible for extracting the knowledge during the execution. Users can configure mARGOt to leverage AGORA, making the execution go through a knowledge-learning phase followed by optimized execution once the knowledge becomes available. Alongside AGORA, the mARGOt autotuning framework also features Theo, a component that ensures that the knowledge produced by AGORA is correct and aligned with the execution status. If the user configures mARGOt to leverage AGORA, mARGOt will start deciding on optimality once the knowledge is available. Alternatively, the user can perform profiling offline and provide the knowledge directly to mARGOt. The 12th EVEREST webinar executions feature offline-profiled knowledge, while the SDK offers multiple examples of dynamic knowledge extraction.

5.2.7 REQ 5.7 - Language support

- **Priority** - Must Have
- **Notes** - The dynamic autotuner requires C++ applications. mARGOt is a C++ library to be linked with the application.
- **Assessment** - The dynamic autotuning framework links to the application through mARGOt, a C++ library. Nonetheless, we developed a group of modules to generate a C++ application that models the behaviour of a general autotuned application. We use these models as a medium to interact with the mARGOt library for non-C++ applications.

5.2.8 REQ 5.8 - HW Knobs

- **Priority** - Must have
- **Notes** - The dynamic autotuning framework should have access to HW knobs. In the case of a configurable accelerator that can be dynamically configured, the knobs have to be exposed to the SW layer.
- **Assessment** - The dynamic autotuning framework can handle hardware knobs provided by the execution environment and treats them as software ones. For example, the dynamic autotuning framework decides whether to execute the FPGA or CPU variant of the kernel according to the values observed from the environment. The 12th EVEREST webinar shows the hardware knob management capabilities of the dynamic autotuning framework.

5.2.9 REQ 5.9- HW monitors

- **Priority** - Could have

- **Notes** - HW accelerators should expose a monitor interface to the run-time to trigger dynamic decisions. Monitoring information should be exposed to the SW and will be used by the run-time environment (dynamic autotuning and virtualized environment) to take dynamic decisions.
- **Assessment** - The dynamic autotuning framework can monitor specific hardware characteristics and adapt its decisions based on the observed values. The dynamic autotuning framework can handle a broad range of hardware monitors since it models them as observable features of the execution environment. It then uses those values to select the best values of hardware and software knobs for the execution. The 12th EVEREST webinar shows how the dynamic autotuning framework detects the presence of the FPGA (a hardware monitor) and uses this information in its decision process.

5.2.10 REQ 5.10 - Execution

- **Priority** - Must have
- **Notes** - The run-time environment requires a CPU where to execute. The runtime environment is composed of software modules that require a core where to execute. Pure HW environments are not considered.
- **Assessment** - The autotuning framework and virtualisation extensions are CPU libraries and cannot run entirely on an accelerator. All use cases use have a host CPU to run the code.

5.2.11 REQ 5.11 - Virtual Environment

- **Priority** - Must have
- **Notes** - Virtualization environment has to enable execution of applications on different hardware in terms of accelerators. The EVEREST Virtualization runtime enables direct FPGA accelerators access for virtual machines, providing the best performance and hardware utilization efficiency.
- **Assessment** - The EVEREST Virtualization extensions framework enables virtual machines direct access to hardware accelerators, with no mediation between the two entities and thus enabling maximum performance. In addition, the developed solution is able to dynamically map one accelerator to a different virtual machines, thus enabling an optimal hardware utilization. The 12th EVEREST webinar shows both direct assignment and dynamic allocation capabilities of the EVEREST Virtualization runtime.

6 Conclusions

This deliverable describes the final state of the EVEREST runtime environment infrastructure that was the main topic of WP5 activities. It describes the three main components of the runtime environment. Evkit, a distributed runtime for parallelizing embarrassingly parallel tasks, Margot, an auto-tuning framework used to select FPGA kernel variants, and ESFM Virtualization extensions, which provide a way of spawning virtual machines that provide access to virtualized FPGA devices.

The WP5 runtime environment demonstrates that EVEREST components developed within WP5 can be meaningfully combined and work together. This environment provides a way for potential users to execute a set of FPGA virtual functions in a distributed fashion, on a set of nodes, in a flexible manner, which is able to dynamically attach and detach virtualized FPGA devices. This could be used in data centers or clusters that provide access to FPGA devices, where these accelerators could be dynamically added or removed from computational jobs based on priority, without having to stop or restart said jobs.

Acronyms

API Application Programming Interface. 6, 8, 9, 17, 22

CPU Central Processing Unit. 6, 9, 25, 26

ESFM EVEREST SR-IOV FPGA Manager. 6, 21–23, 26, 30

FPGA Field Programmable Gate Array. 5–10, 17, 21, 22, 25, 26, 30

GPU Graphics Processing Unit. 17

HPC High-Performance Computing. 21

JSON JavaScript Object Notation. 10, 11, 13, 15

MSI Message Signaled Interrupt. 23

PCI Peripheral Component Interconnect. 22, 23

PCIe Peripheral Component Interconnect Express. 21

PF Physical Function. 21, 22, 24

PTDR Probabilistic Time-Dependent Routing. 8–10

RHEL Red Hat Enterprise. 23

SR-IOV The Single Root I/O Virtualization. 6, 21–23

TCP/IP Transmission Control Protocol/Internet Protocol. 8, 10, 11

VF Virtual Function. 21–24

VM Virtual Machine. 6, 21–23

XRT Xilinx Runtime Library. 23

References

[1] Everest webinar: SDK run-time components.

[2] Jason Ansel et al. Opentuner: An extensible framework for program autotuning. In PACT. IEEE, 2014.

[3] Spyros Chiotakis, Sébastien Pinneterre, and Michele Paolino. vFPGAmanager: A hardware-software framework for optimal FPGA resources exploitation in network function virtualization. In 2019 European Conference on Networks and Communications (EuCNC), pages 47–51, 2019.

[4] Yufei Ding et al. Autotuning algorithmic choice for input sensitivity. In ACM SIGPLAN Notices, volume 50. ACM, 2015.

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

[6] Everest Consortium. D5.1 - Intermediate runtime environment report, 2022.

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

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

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

[10] Everest Consortium. D2.5 - Refined Definition of Language Requirements, 2023.

[11] Everest Consortium. D4.5 - Final report of the compilation framework, 2024.

[12] Everest Consortium. D5.6 - Beta release of the virtualized runtime environment, 2024.

[13] Everest Consortium. D6.3 - Final version of the EVEREST applications, 2024.

[14] Davide Gadioli, Emanuele Vitali, Gianluca Palermo, and Christina Silvano. margot: a dynamic autotuning framework for self-aware approximate computing. IEEE Transactions on Computers, 2018.

[15] Michael A Laurenzano et al. Input responsiveness: using canary inputs to dynamically steer approximation. ACM SIGPLAN Notices, 2016.

[16] Joel Mbongue, Festus Hategekimana, Danielle Tchuinkou Kwadjo, David Andrews, and Christophe Bobda. FPGAVirt: A novel virtualization framework for FPGAs in the cloud. In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pages 862–865, 2018.

[17] Joshua San Miguel et al. The anytime automaton. In ACM SIGARCH Computer Architecture News. IEEE Press, 2016.

[18] Christian Pilato, Subhadeep Banik, Jakub Beránek, Fabien Brocheton, Jeronimo Castrillon, Riccardo Cevasco, Radim Cmar, Serena Curzel, Fabrizio Ferrandi, Karl F. A. Friebel, Antonella Galizia, Matteo Grasso, Paulo Silva, Jan Martinovic, Gianluca Palermo, Michele Paolino, Andrea Parodi, Antonio Parodi, Fabio Pintus, Raphael Polig, David Poulet, Francesco Regazzoni, Burkhard Ringlein, Roberto Rocco, Katerina Slaninova, Tom Slooff, Stephanie Soldavini, Felix Suchert, Mattia Tibaldi, Beat Weiss, and Christoph Hagleitner. A system development kit for big data applications on FPGA-based clusters: The EVEREST approach. In Proceedings of the 2024 Design, Automation and Test in Europe Conference (DATE), DATE'24, page 6pp, March 2024.

[19] 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.

[20] Ari Rasch et al. Atf: A generic auto-tuning framework. In High Performance Computing and Communications. IEEE, 2017.

[21] 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.

[22] Christoph A Schaefer, Victor Pankratius, and Walter F Tichy. Atune-il: An instrumentation language for auto-tuning parallel applications. In European Conference on Parallel Processing, pages 9–20. Springer, 2009.

[23] Stephanie Soldavini, Karl Friebel, Mattia Tibaldi, Gerald Hempel, Jeronimo Castrillon, and Christian Pilato. Automatic creation of high-bandwidth memory architectures from domain-specific languages: The case of computational fluid dynamics. ACM Transactions on Reconfigurable Technology and Systems, 16(2):1–34, 2023.

[24] Xin Sui et al. Proactive control of approximate programs. ACM SIGOPS Operating Systems Review, 2016.

[25] Wei Wang, Miodrag Bolic, and Jonathan Parri. pvFPGA: Accessing an FPGA-based hardware accelerator in a paravirtualized environment. In 2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pages 1–9, 2013.