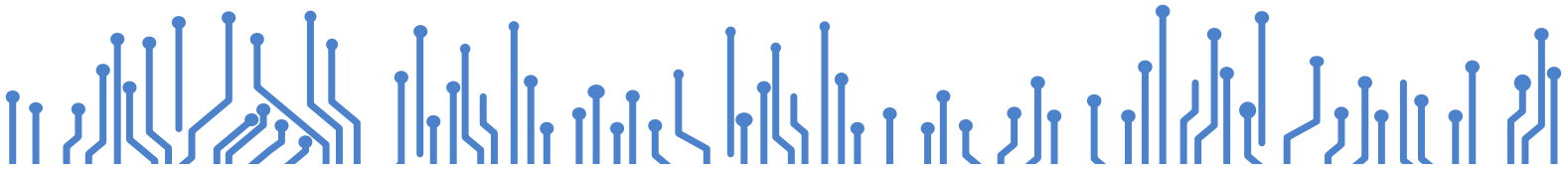
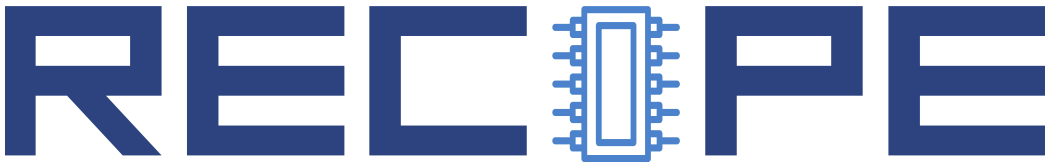
**REliable Power and time-ConstraInts-aware Predictive management of heterogeneous Exascale systems**





# WP2 Runtime Resource Management Infrastructure

## 2.5 RECIPE Software Stack Integration

http://www.recipe-project.eu

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 801137

**Grant Agreement No.: 801137**
**Deliverable: 2.5 RECIPE Software Stack Integration**

**Project Start Date**: 01/05/2018 **Duration**: 42 months
**Coordinator**: *Politecnico di Milano, Italy*

| | |
|---|---|
| **Deliverable No**: | 2.5 |
| **WP No**: | 2 |
| **WP Leader**: | Giuseppe Massari |
| **Due date**: | 30/07/2021 |
| **Delivery date**: | 06/08/2021 |

**Dissemination Level**:

| | | |
|---|---|---|
| PU | Public Use | X |
| PP | Restricted to other programme participants (including the Commission Services) | |
| RE | Restricted to a group specified by the consortium (including the Commission Services) | |
| CO | Confidential, only for members of the consortium (including the Commission Services) | |

# DOCUMENT SUMMARY INFORMATION

| | |
|---|---|
| **Project title**: | **REliable Power and time-ConstraInts-aware Predictive management of heterogeneous Exascale systems** |
| **Short project name**: | RECIPE |
| **Project No**: | 801137 |
| **Call Identifier**: | H2020-FETHPC-2017 |
| **Thematic Priority**: | Future and Emerging Technologies |
| **Type of Action**: | Research and Innovation Action |
| **Start date of the project**: | 01/05/2018 |
| **Duration of the project**: | 42 months |
| **Project website**: | http://www.recipe-project.eu |

## 2.5 RECIPE Software Stack Integration

| | |
|---|---|
| **Work Package**: | WP2 Runtime Resource Management Infrastructure |
| **Deliverable number**: | 2.5 |
| **Deliverable title**: | RECIPE Software Stack Integration |
| **Due date**: | 30/07/2021 |
| **Actual submission date**: | 06/08/2021 |
| **Editor**: | G. Massari |
| **Authors**: | G. Massari, F. Reghenzani, F.Sciamanna, M. Peta, G. Agosta, W. Fornaciari, A. Cilardo, G. Tiano, L. Del Barone, M. Zapater, J.M. Martinez, J. Flich |
| **Dissemination Level**: | PU |
| **No. pages**: | 47 |
| **Authorized (date)**: | 06/08/2021 |
| **Responsible person**: | W. Fornaciari |
| **Status**: | Plan Draft Working Final **Submitted** Approved |

**Revision history**:

| Version | Date | Author | Comment |
|---|---|---|---|
| v.0.1 | 25/05/2021 | POLIMI | Initial skeleton |
| v.0.2 | 01/06/2021 | CERICT | First version of the LRM/PM contributions |
| v.0.3 | 03/06/2021 | POLIMI | C/R first parts |
| v.0.4 | 05/06/2021 | POLIMI | NVIDIA Tegra integration |
| v.0.5 | 20/06/2021 | UPV | FPGA monitoring for MANGO |
| v.0.6 | 03/07/2021 | POLIMI | First revision and fixes |
| v.0.7 | 06/07/2021 | POLIMI | LRM section draft |
| v.0.8 | 15/07/2021 | POLIMI | Reliability support |
| v.0.9 | 16/07/2021 | POLIMI | Energy support |
| v.0.10 | 21/07/2021 | POLIMI | Timing support |
| v.0.11 | 25/07/2021 | POLIMI | LRM Platform integration revision |
| v.0.12 | 28/07/2021 | CERICT | CERICT contribution revised |
| v.0.12 | 29/07/2021 | POLIMI | C/R experimental results |
| v.0.13 | 06/08/2021 | EPFL | GRM section draft |

**Quality Control**:

| | Who | Date |
|---|---|---|
| **Checked by internal reviewer** | CERICT | 06/08/2021 |
| **Checked by WP Leader** | Giuseppe Massari | 06/08/2021 |
| **Checked by Project Technical Manager** | G. Agosta | 06/08/2021 |
| **Checked by Project Coordinator** | W. Fornaciari | 06/08/2021 |

# COPYRIGHT



# ACKNOWLEDGEMENTS

RECIPE is a project that has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No 801137. Please see http://www.recipe-project.eu for more information.

The partners in the project are Universitat Politècnica de València (UPV), Centro Regionale Information Communication Technology scrl (CeRICT), École Polytechnique Fédèrale de Lausanne (EPFL), Barcelona Supercomputing Center (BSC), Poznan Supercomputing and Networking Center (PSNC), IBT Solutions S.r.l. (IBTS), Centre Hospitalier Universitaire Vaudois (CHUV). The content of this document is the result of extensive discussions within the RECIPE ©Consortium as a whole.

# DISCLAIMER

# Contents

# 1 Introduction

In every High-Performance Computing (HPC) platform the software stack plays a fundamental role in terms of both functionalities exposed to the users and management strategy of the workload (scheduling, resource allocation) and the hardware (thermal, power, energy management).

In the RECIPE project, the software stack is a collection of frameworks and libraries cooperating for achieving the objectives stated in the DoW, i.e., providing target HPC platforms with guarantees in terms of reliability and timing requirements for the hosted applications, while keeping their energy efficiency.

The RECIPE reference platforms are characterized by heterogeneous configurations, featuring multi-core CPUs, high-performance GPUs and custom accelerators deployed on integrated FPGA boards. A further aspect to consider is that, due to the HPC domain, a certain (complex) hardware configuration is typically replicated over a certain number of computational nodes.

Give these premises, the design and development of the RECIPE software stack has posed a challenging effort. The heterogeneity of the hardware platforms and the contributions coming from different work packages have required the definition of several interfaces and the continuous revision of the developments in order to achieve a smooth and maintainable integration of all the components.

The focus of Deliverable D2.5 is sketched in Figure 1. The picture shows the software components lying on top of the hardware and the interaction flows among them. Starting from the top, the Global Resource Manager (*SLURM*) runs resource management policies at the scope of the entire (multi-node) platform, given also the input received from the Local Resource Manager (*BarbequeRTRM*). This is responsible for the resource management actions at the level of the single computational node, considering the monitoring and profiling of data collected through the Platform Integration Layers and the WP3 libraries (*Hardware Reliability* and *Timing Analysis*). The former is used for predicting the reliability of the managed hardware, with respect to the local resource allocation choices. The latter provides models allowing us to perform statistical execution times analysis, to identify the distribution of the WCET with respect to given resource allocation choices and thus drive the local resource manager policies. The Platform Integration Layers box instead must be considered a collection of low-level interfaces, exploited by the local resource manager to built an internal abstraction layer on one side, and to support a wide set of hardware configurations, on the other side.

Finally, the programming models aim at masquerading the complexity of the hardware, while providing support to the local resource manager in terms of profiling activities, when possible. As already explained in the previous deliverables, we provided some libraries and execution models to make the application execution run-time manageable and allows the application to adapt itself to the changing resource assignments. However, we developed also an approach to support legacy applications, at the price of loosing the possibility of implementing an adaptive behaviour.

In the previous deliverables, we described the single software components of the stack in detail. In this deliverable, we tried to give a technically detailed view over the integration of the different components. It is finally worth remarking that this represents the ground on top of which the actual resource allocation policies are built, as it will be described in the next deliverable D3.7.

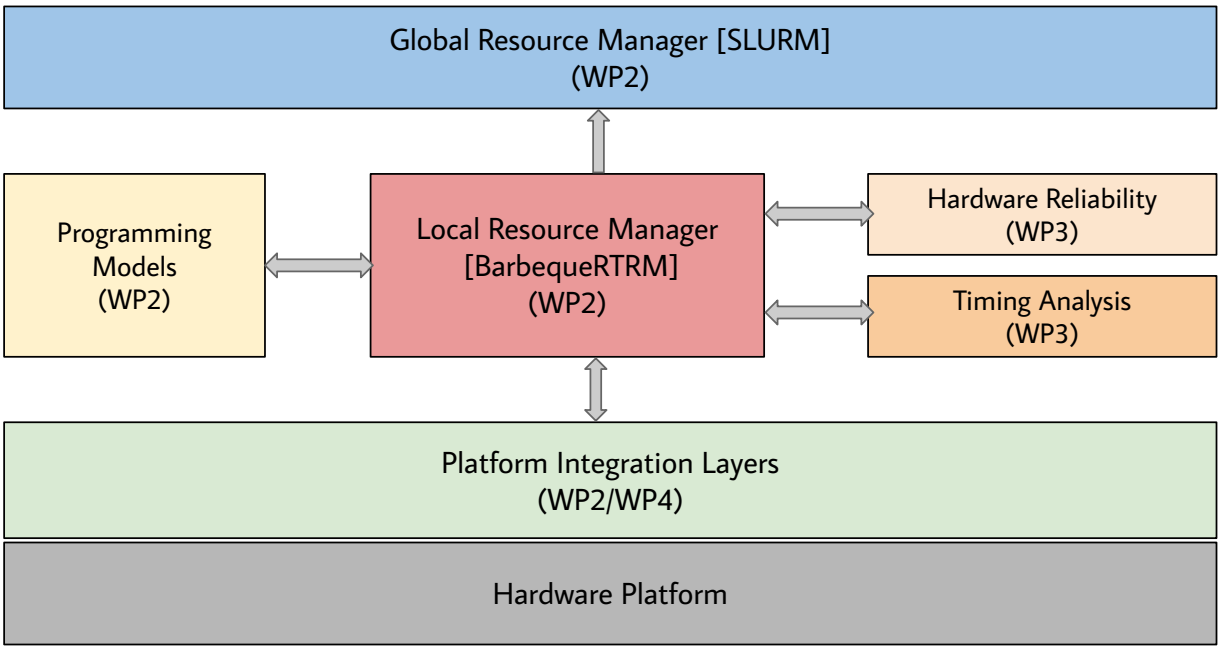



Figure 1: Block diagram of the RECIPE software component / groups

## 1.1 Overview and document structure

In Deliverable D2.1, we introduced the overall design of the stack, discussing the initial interaction flows among the resource managers, the programming models and the platform integration layers. In deliverables D2.2 and D2.3 instead, we discussed the design and implementation details of the local and the global resource manager, respectively. Finally, in D2.4, we covered several aspects regarding the programming models and the challenges due to the programmability of heterogeneous computing platforms, introducing the programming models inherited from the H2020 MANGO Project experience.

Eventually, this D2.5 is the last deliverable planned by the Work Package 2. The goal of this document is to provide a complete view over the entire software stack deployed on the RECIPE platforms, according to the final developments and integration efforts. The software stack includes not only the effort carried out during the Work Package 2 activities, but also some outcome, in terms of software components, developed by Work Packages 3 and 4.

The deliverable is structured as it follows. In Section 2, we go through the local resource manager structure, focusing on the recent updates, with special attention to the pillars built for energy, reliability and timing-aware resource allocation policies.

In Section 3, we report the last integration developments introduced in SLURM, with a focus on the platform deployment, the integration of the HELENNA engine, used in UC3 application, and the support to checkpoint/restore at the global scale.

In Section 4, we discuss the last developments of the MANGO Programming Library and the Adaptive Execution Model, plus the support for exploiting the Remote Direct Memory Access mechanisms on HPC applications spawned on multiple nodes and access FPGA accelerators.

Finally, in Section 5 we draw the conclusions, considering the experience gained in the effort put for the WP tasks, the addressed challenges and the possible future research and developments that could be taken into account as next steps.

# 2 Local Resource Manager: BarbequeRTRM

During the last period of the RECIPE project, the *Local Resource Manager* (Barbeque Run-Time Resource Manager) has been developed further to complete the process of integration of the target platforms and the external software frameworks. Such components of the RECIPE software stack represent the outcome of different work packages (WP2, WP3, WP4).

In Figure 2, we show an updated version of the BarbequeRTRM internal components, according to a layered view. It is worth saying that the architecture in the picture provides a partial view of the local resource manager design. We deliberately omitted some components in order to simplify the picture and let the reader focus on the developments carried out for the purpose of accomplishing the RECIPE project objectives.

More in detail, some of the last developments involved the *Application Manager*, which keeps track of the status of the managed applications. Here we further developed the application profiling support by adding the timing analysis, as described in Section 2.4, through the exploitation of the *Timing Analysis Library*.

For the reliability management part instead, in D2.2 we introduced the internal *Reliability Manager* as the module responsible for monitoring the reliability of the hardware resources. We extended this module, as explained later in Section 2.3, to adapt the checkpoint activities to the status of the system at run-time, also based on the prediction provided by the *Hardware Reliability Library*.

Finally, we introduced a new additional module, the *Energy Monitor*, specifically to measure/estimate the energy consumption of each managed hardware resource (processing units) of the system. This has been enabled by the implementation of the missing Power Manager and Platform Proxy derived modules, required for supporting all the expected configurations of the RECIPE platform (`NVIDIA_PowerManager`, `TEGRA_PowerManager`, `RECIPE_PowerManager`, `NVIDIA_PlatformProxy`, `TEGRA_PlatformProxy`, `RECIPE_PlatformProxy`), and the extension of `CPU_PowerManager`, with the energy monitoring support. This is explained in the Section 2.2.

The implementation effort just summarized represents the basis for the next and final step: the implementation of the resource allocation policies. The policies will contribute to address the problem of improving the energy-efficiency of the execution of the applications, maximizing the reliability of the hardware and providing the user applications with timing guarantees when requested. The description of the policies and the related experimental results and will be accurately reported in the next deliverable D3.7.

## 2.1 Platform Integration

In order to support a specific hardware platform we needed to integrate the access to the low-level interfaces providing A) hardware resources enumeration and assignment; B) run-time monitoring of their status (load, power consumption, temperature). Then, depending on the platform, we could rely on standard interfaces (Linux' `sysfs` and `procfs`), vendor libraries (e.g., NVIDIA, AMD, Intel, etc. . . ) or custom runtime layers, built on top of device drivers.

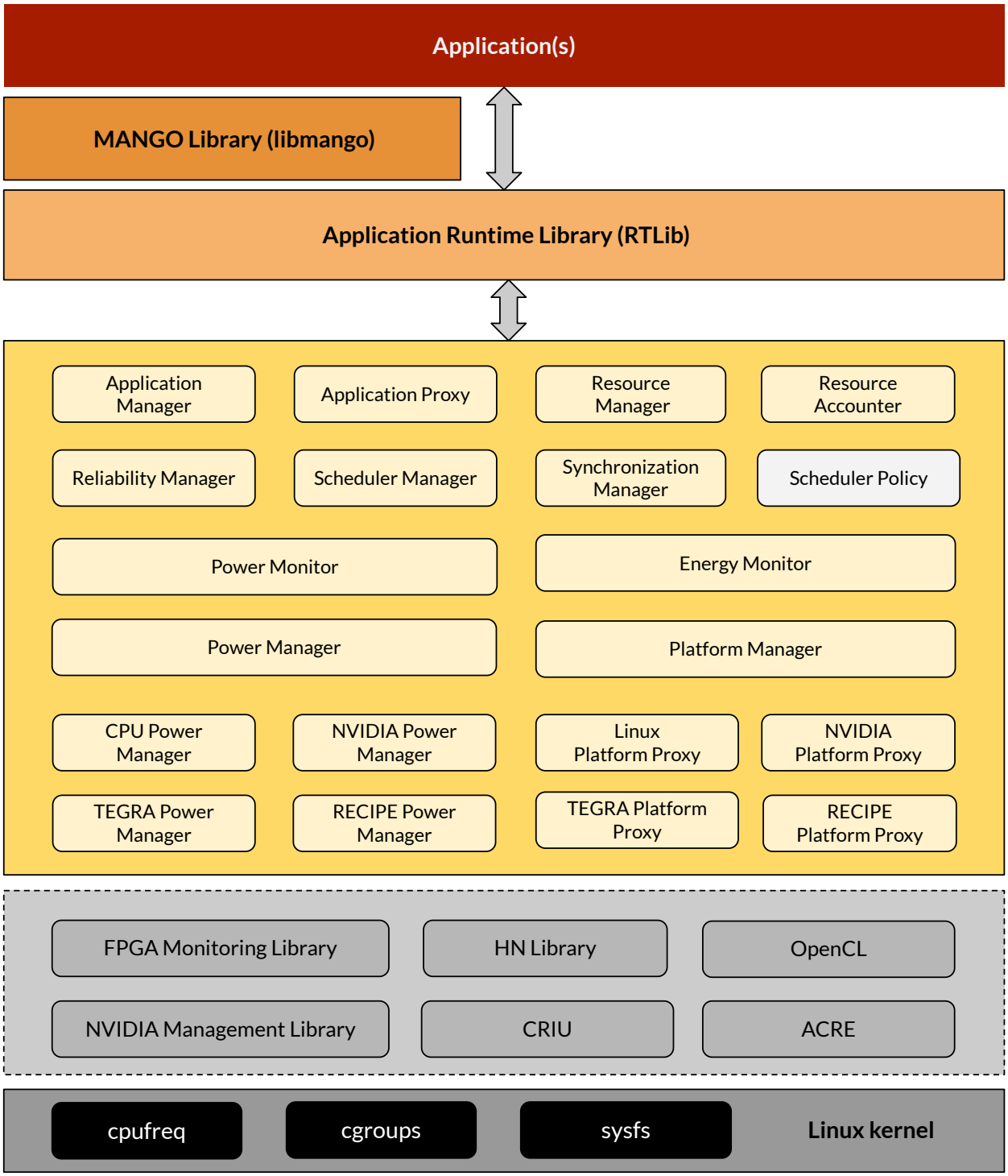



Figure 2: BarbequeRTRM internal layers

### 2.1.1 Platform Interfaces

In this subsection we report the aforementioned low-level interfaces exploited by the Platform Proxy and Power Manager derived modules, to get access to the status information of the hardware resources and possibly control them. Such interfaces are grouped in the grey box

of Figure 2. As we will see, for CPU and GPU we relied on interfaces already provided by the operating system of the device vendors. For FPGA resources instead, we had to follow a different approach, as explained in the following.

**Linux cgroups** Linux-based operating systems provide us with the *control groups* framework [6], more commonly called cgroups. This framework represents the OS-level interface exploitable to allocate resources, such as CPU time quota, CPU cores mapping, memory, network bandwidth, etc. . . , among processes running on the system. In the BarbequeRTRM, we integrated the access to cgroups to reserve CPU, memory and networking bandwidth resources. A work in progress is also focusing on integrating the I/O bandwidth control of block devices. In the specific context of RECIPE, we must consider the Linux cgroups as the knob for enforcing the reservation of CPU resources only. Cgroups is mainly exploited by the `Linux_PlatformProxy` module.

**OpenCL Runtime** OpenCL (Open Computing Language) has been already mentioned in previous deliverables. It is an open standard for cross-platform, parallel programming of diverse accelerators, aiming a providing a consistent API and runtime layer, whatever is the target heterogeneous computing system: desktop, supercomputer, cloud infrastructure, mobile devices or embedded system [5]. Since most of the FPGA vendors come with their OpenCL support, we decided to exploit the very small set of functions of the OpenCL API, to enumerate the OpenCL platforms and the connected devices available on the system. This approach has been implemented in the `RECIPE_PlatformProxy` module. Then, depending on the target FPGA platform selected, the BarbequeRTRM can be configured to link the OpenCL runtime or other custom runtime libraries (see later). In summary, the OpenCL intermediation is required at boot-time only. No other OpenCL functions are currently required by the BarbequeRTRM for management purposes. This means that, the resource allocation is *weak*ly performed. In other words, the local resource manager notifies the application (through the programming model library) about the assigned devices. However, since there is no chance of actually constraining the access to the computing devices, it is up to the OpenCL application to actually select the assigned device. Overall, exploiting the vendor-provided OpenCL runtime is a portable and low latency approach, but at the price of a low degree of control over the assignment of computing resources.

**HN Library** This library represents the low-level interface towards the MANGO FPGA cluster. It implements a minimal OpenCL runtime, such that the MANGO FPGA can be made accessible to resource manager and application. This layer has been deeply revised with respect to the previous version released with the MANGO project, in order to improve the compatibility and the portability of the upper software layers. More details on the last version of the library are reported in D4.3.

**Intel RAPL** The last generations of Intel processors are characterized also by the presence of the *Running Average Power Limit (RAPL)* [9], which is made by a hardware control logic for thermal/power management and a set of counter registers, providing energy and power consumption information. Such information typically does not come from an actual power meter, but rather they are the output of a power model. In Linux-based operating systems, such an interface is made accessible via the `sysfs` file-system [7]. According to the last developments introduced in the BarbequeRTRM, the RAPL information are accessed by the `CPU_PowerManager` module, to get the information about the current energy consumed by the CPUs. In this regard,

we would like to remark how the evaluation of the accuracy of the sampled values is out of the scope of the RECIPE project.

**NVIDIA Management Library** NVIDIA provides the user with a library, called NVML [8], including a C-based API for monitoring and controlling the GPU devices. A glimpse of the monitoring capabilities of the library is provided by the `nvidia-smi`. The NVML is intended to be a framework for building third-party software. In fact, in the BarbequeRTRM, the `NVIDIA_PowerManager` and the `NVIDIA_PlatformProxy` implementations use the NVML functions for enumerating the GPU devices installed in the system, reading the current temperature and power consumption and potentially performing performance/DVFS-scaling.

**NVIDIA Tegra** According to the last DoW amendment, we added to the set of target platforms for the experimental evaluations, a heterogeneous system based on NVIDA Tegra devices (Jetson Xavier model). NVIDA Tegra is a System-on-Chip (SoC) series developed for mobile devices and designed to emphasize performance, while maintaining good energy efficiency in complex applications, like gaming and machine learning. In the RECIPE project, we installed this kind of devices on a distributed system devoted to run machine-learning based workloads. Unfortunately, for this family of devices we had to proceed with a further integration effort, since Tegra is not included among the architectures supported by the aforementioned NVIDIA Management Library. The status information of the SoC (including CPU and GPU) are in fact exposed via `sysfs` file system and the `tegrastats` utility.

Therefore, we extended the *BarbequeRTRM* in order to enable power and resource management on devices based on Tegra SoC, by introducing additional modules derived from the base class `PowerManager`, as shown in Figure 3: we created the `TEGRA_CPUPowerManager` and `TEGRA_PowerManager`: the former is in charge of accessing the CPU status, while the latter is in charge of reading the GPU status. Both modules operate by parsing the system information exposed via sysfs.

Finally, from the resource management standpoint, we added a new class derived from the base class `PlatformProxy`: the `TEGRAPlatformProxy` class. The singleton instance is in charge of registering the list of computing resources actually available, by interfacing the internal module `ResourceAccounter`. A similar registration procedure is performed also for the `PowerMonitor` and the `EnergyMonitor` modules, in order to enable the periodical retrieval of the temperature, the power consumption and the current energy consumption information.

### 2.1.2 FPGA Monitoring Library

Monitoring the status of FPGA-based processing resources is much more challenging than for the GPUs. This comes from the massive heterogeneity due to the (re)configurable nature of the FPGAs. While for the on-board sensors, like power meters or thermal sensors, the vendors can provide the user with a suitable interface (library or filesystem-based), for the monitoring of other information, like the level of activity (or load) of custom accelerators, we may need to introduce in the hardware design additional counter registers, along with the core processing architecture.

In this regard, we decided to design and implement a single *FPGA Monitoring Library (libfpgamon)*, coming with a unique interface for the local resource manager and hiding the platform-

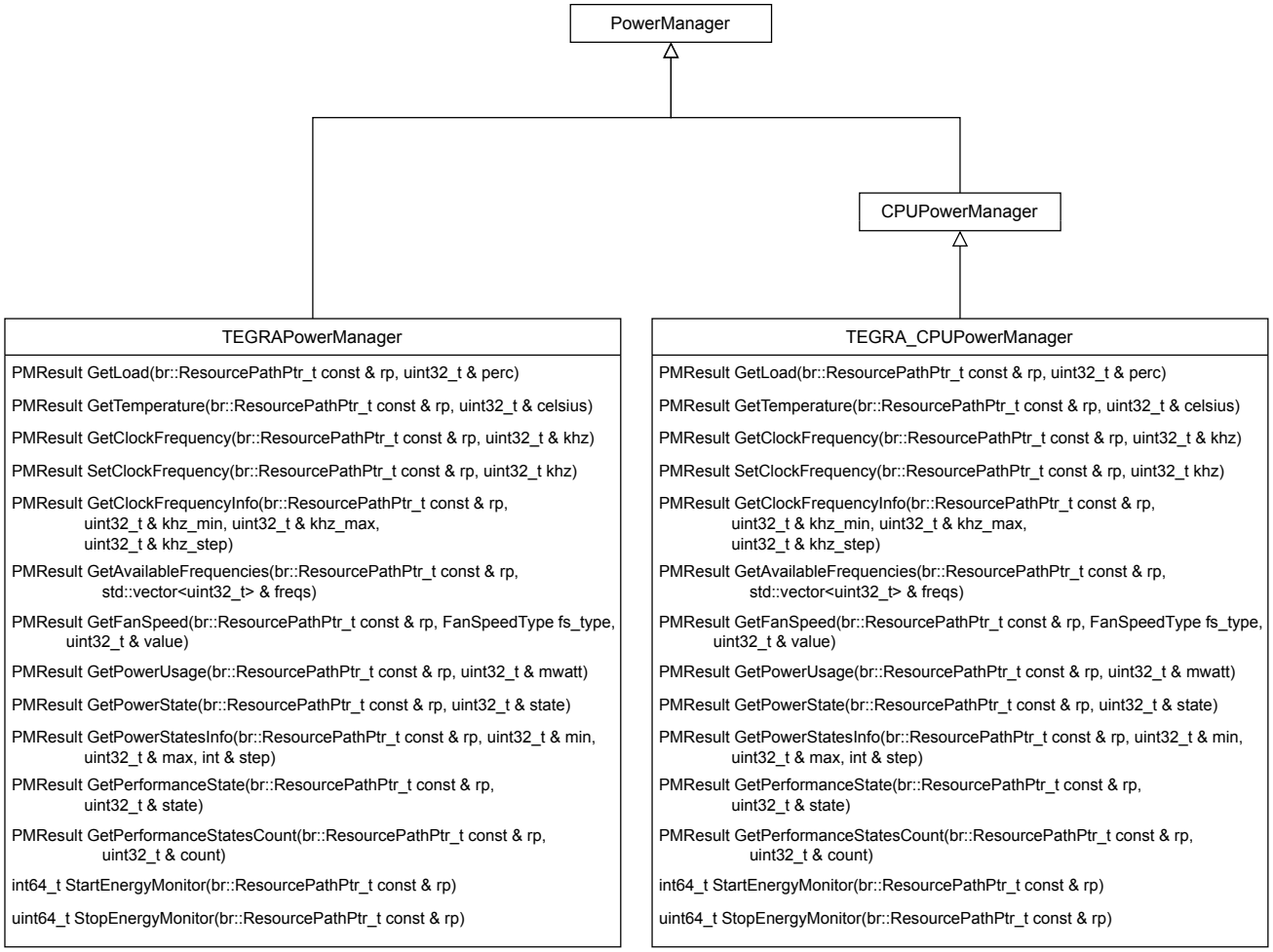



Figure 3: BarbequeRTRM: NVIDIA Tegra Power managers

specific implementations under-the-hood. The library includes two main header files: one exposing the user interfaces (`fpgamon.h`) and actually included by the BarbequeRTRM integration code of the `RECIPE_PowerManager`, and the other defining the interfaces for the platform-specific contribution (`fpgamon_hal.h`).

The main content of the former is reported in Listing 1. We defined the `fpgamon_context_t` data type to store the general information of the system, in terms of number of specific platforms included, pointers to the platform-specific implementations and vector of all the counters provided. A data structure of such a type must be filled by the initialization function(`fpgamon_init()`) and then provided as first argument to all the library functions. The functions currently defined in the header file are mainly getters for status information like temperature, power consumption and load, plus three functions devoted to the control of the, optionally provided, hardware counters (get/reset actions). The functions take as additional arguments a pair of identification numbers: one for the platform and one for the device. This approach is coherent with respect to what commonly happens with OpenCL. We use the term *platform* for identifying the runtime layer provided by a vendor and the term *device* to refer to a specific processing unit deployed on the platform. The identification numbers of platforms and devices must be coherent with respect to the values returned by the OpenCL runtime linked by the BarbequeRTRM and exploited by the `RECIPE_PlatformProxy` module.

Listing 1: fpgamon.h

```
/** fpgamon.h extract... */

typedef struct context
{
   unsigned short int nr_platforms;
   fpgamon_platform_t ** platforms;
   fpgamon_ctr_array_ptr * counters;
   unsigned short int * nr_counters;

} fpgamon_context_t;

typedef uint32_t fpgamon_platform_ids_t;

uint32_t fpgamon_get_version(void);

int fpgamon_init(fpgamon_context_t * ctx,
                 uint32_t nr_platforms,
                 fpgamon_platform_ids_t * platform_ids);

void fpgamon_shutdown(fpgamon_context_t * ctx);


uint16_t fpgamon_get_load(fpgamon_context_t * ctx,
           uint16_t plat_id,
           uint16_t device_id);

uint32_t fpgamon_get_temperature(fpgamon_context_t * ctx,
             uint16_t plat_id,
             uint16_t device_id);

uint32_t fpgamon_get_power(fpgamon_context_t * ctx,
                           uint16_t plat_id,
                           uint16_t device_id);

uint16_t fpgamon_get_nr_counters(fpgamon_context_t * ctx,
                                 uint16_t plat_id);

fpgamon_ctr_array_ptr fpgamon_get_counters(fpgamon_context_t * ctx,
                                           uint16_t plat_id,
                                           uint16_t device_id,
                                           uint16_t * count);

void fpgamon_reset_counters(fpgamon_context_t * ctx,
                            uint16_t plat_id,
                            uint16_t device_id);
```

The `fpgamon_hal.h` header instead, provides the platform integrators with a simple interface to plug-in their specific support. As we can see in Listing 3, the interface is basically represented

by the `fpgamon_platform_t` data structure, featuring a string to store the platform name, plus a set of function pointers. This means that, at run-time, the library will load all the platforms configured at compile-time, by properly filling a vector of such data structures and forwarding the library function calls to the target platform, on the basis of the platform identification number specified.

Listing 2: fpgamon_hal.h

```
/*
 * File: fpgamon_hal.h
 * ...
 */

typedef struct platform
{
   char name[FPGAMON_PNAME_MAXLEN];

   int (*init)();
   void (*shutdown)();
   uint16_t(*get_load)(uint16_t);
   uint32_t(*get_temperature)(uint16_t);
   uint32_t(*get_power)(uint16_t);
   uint16_t(*get_nr_counters)();
   uint32_t(*get_counters)(uint16_t,
                           fpgamon_ctr_array_ptr * ctrs,
                           uint16_t len);
   void (*reset_counters)(uint16_t);

} fpgamon_platform_t;
```

**The MANGO cluster platform-specific support** A first FPGA-based platform exploited by RECIPE and inherited from the H2020 MANGO project, is represented by the MANGO cluster, based on several proFPGA modules. The support, developed by UPV, hides, as much as possible, the low-level details and specificities of the cluster communication requirements, for accessing the status information of each single FPGA module.

The MANGO cluster modules provide real-time information of the physical devices building the system for: number of devices attached, status of the boards (up-down), number of extension boards connected to the FPGA modules (memories, communication interfaces, . . . ), temperature and real-time voltage and current of the in-board power supplies.

Access to the information provided by the sensors in the different FPGA modules of the MANGO cluster is done through a separate communication interface, i.e., the FPGA monitoring library utilizes a different communication interface, with a different communication protocol, with respect to the interface used for compute data exchange between host applications and FPGA devices. This relies on the *MMI64 PCI Express* driver, described in D4.3. This approach shows the advantage of decoupling the control path from the data path, thus preventing MANGO cluster management and monitoring tasks from impacting the performance of the communications between host applications and compute FPGAs. In the same manner, the local resource manager monitoring activity will not be affected by any possible congestion of the communication

interface between host applications and the compute FPGAs, due to data transfers. This way, the FPGA monitor library prevents any degradation on compute data transfer rates that the monitoring task of the devices in the MANGO cluster may cause.

In order to handle the dedicated communication interface with the MANGO cluster, we implemented in the the FPGA monitor library a couple of additional functions. These functions are necessary to start and stop the dedicated communication interface with the MANGO cluster.

This platform-specific contribution implemented the previously described API as it follows:

**init()** initialize a dedicated communication interface with MANGO cluster specified in function parameters. Check system configuration and initial system status. Start sensor reading task.

**shutdown()** Finish all related processes with MANGO cluster specified in function parameters. Finish all ongoing sensor reading processes. Stop sensor reading task. Close dedicated communication interface with MANGO cluster. Release communication interface.

**get_load()** Get current utilization of FPGA module specified in function parameters.

**get_temperature()** Get current temperature of FPGA module specified in function parameters.

**get_power()** Get current power consumed by FPGA module specified in function parameters. Value is calculated from the information reported by the different sensors in the board.

The MANGO cluster provides accurate and detailed information of voltage and current supplied by the different power supplies present in each FPGA module. Each FPGA module includes dedicated power supplies for each extension site of the board, for the module communication transceivers, for the board surrounding logic, and for the FPGA chipset itself.

The information returned by the MANGO cluster may vary depending on the type of FPGA module, since some FPGA modules may have different number and type of hardware sensors. The FPGA monitor library implements a unique data structure capable to store all the information returned by any FPGA module variant, so after reading the information from the FPGA module, it is stored in a common format independent of the FPGA module type it comes from.

The hardware monitoring data structure is:

Listing 3: fpgamon_hal.h

```
1  typedef struct {
2
3      float pvio_ta1; //!< voltage value of PVIO for connector TA1
4      float pvio_ta2; //!< voltage value of PVIO for connector TA2
5      float pvio_tb1; //!< voltage value of PVIO for connector TB1
6      float pvio_tb2; //!< voltage value of PVIO for connector TB2
7      float pvio_ba1; //!< voltage value of PVIO for connector BA1
8      float pvio_ba2; //!< voltage value of PVIO for connector BA2
9      float pvio_bb1; //!< voltage value of PVIO for connector BB1
10     float pvio_bb2; //!< voltage value of PVIO for connector BB2
11     float pv_fpga_core; //!< voltage value of FPGA core voltage
12     float pv_p0v9; //!< voltage value of FPGA aux P0V9
13     float pv_p1v8; //!< voltage value of FPGA aux P1V8
14     float pv_rgxb; //!< voltage value of FPGA GXB receiver
```

```
15      float pv_tgxb; //!< voltage value of FPGA GXB transmitter
16      float current_pvio_ta1; //!< current value of PVIO for connector TA1
17      float current_pvio_ta2; //!< current value of PVIO for connector TA2
18      float current_pvio_tb1; //!< current value of PVIO for connector TB1
19      float current_pvio_tb2; //!< current value of PVIO for connector TB2
20      float current_pvio_ba1; //!< current value of PVIO for connector BA1
21      float current_pvio_ba2; //!< current value of PVIO for connector BA2
22      float current_pvio_bb1; //!< current value of PVIO for connector BB1
23      float current_pvio_bb2; //!< current value of PVIO for connector BB2
24      float current_pv_fpga_core; //!< current value of FPGA core voltage
25      float current_pv_p0v9; //!< current value of FPGA aux voltage
26      float current_pv_p1v8; //!< current value of FPGA aux voltage
27      float current_pv_rgxb; //!< current value of FPGA GXB receiver
28      float current_pv_tgxb; //!< current value of FPGA GXB transmitter
29      int core_temperature; //!< current die temperature [°C]
30      int gxb_tile1_temperature; //!< current transceiver tile temperature [°C]
31      unsigned char temp_error; //!< over temperature error
32
33  } hw_sys_fm_status_t;
```

Accessing the sensors of an FPGA module takes a few milliseconds, with an upper bound less than ten milliseconds. So, the total amount of time to get the statistics of the complete system will depend on the number of devices to read the sensors.

To avoid the time overhead needed to acquire the data from the sensors, the FPGA monitoring library launches a thread during the library initialization function call to get the data before the resource manager requests it. This thread triggers the read of the values of the sensors in the FPGA modules in a loop fashion function. Therefore, when the resource manager sends the request of data for a specific FPGA module, the FPGA monitor library processes the data stored in the structure associated to that FPGA module, and immediately returns the requested information to the resource manager.

The FPGA monitor task thread runs and collects the information from the MANGO system until the resource manager triggers the shutdown function. The shutdown function will command the thread to finish the data collection process, to finally release the dedicated communication interface with the MANGO cluster. This process is shown in Figure 4.

By means of the FPGA monitor library, the resource manager can seamlessly access the readings of the different sensors in the FPGAs of the MANGO cluster from the FPGA monitoring library via the common HW monitoring API functions without having to handle the communications with the MANGO cluster.

**The Alveo platform-specific support.** A second FPGA platform included in the RECIPE prototype is an Alveo FPGA board mounted in the CERICT prototype. The platform-specific support, named RECIPE *rFPGA*, is contained in the `alveo_xdma` sub-directory. The platform-specific code implements the same API already presented:

**init()** initialize the communication interface with the rFPGA shell.

**shutdown()** release the communication interface.

**get_load()** get current utilization of FPGA module specified in function parameters.

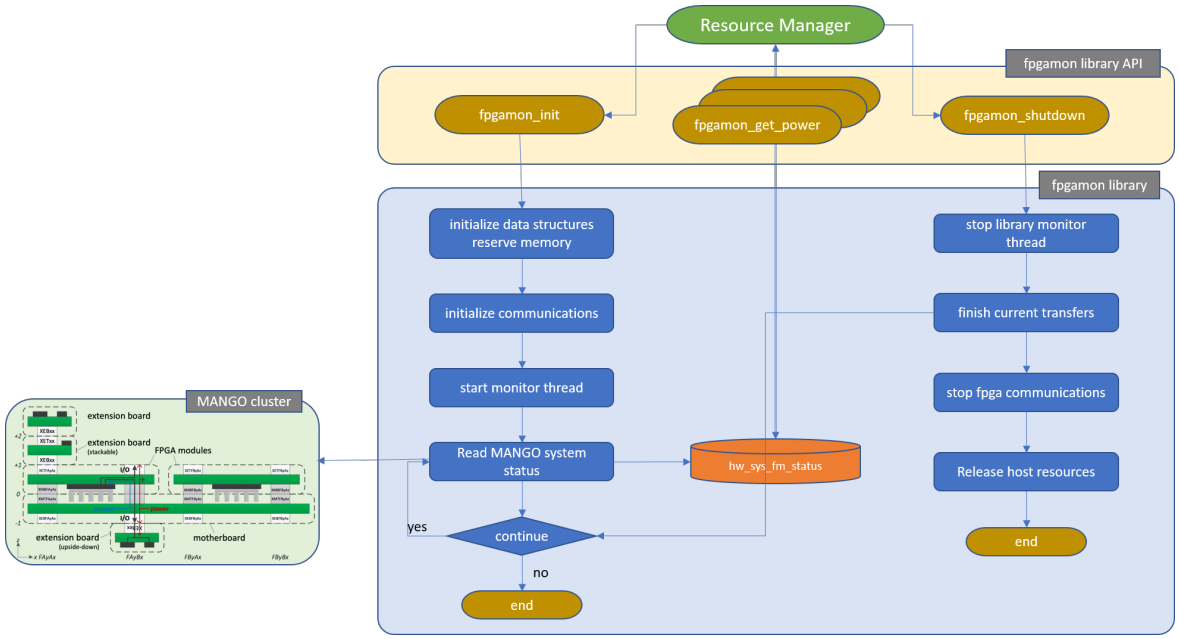



Figure 4: MANGO cluster integration in the FPGA Monitoring Library

**get_temperature()** get current temperature of FPGA module specified in function parameters.

**get_power()** get current power consumed by FPGA module specified in function parameters.

For the Alveo-based prototype we rely on two resources available on the COTS acceleration cards used at CERICT:

- The SYSMON primitive and associated IP wrapper available for Xilinx UltraScale devices, which was used in the RECIPE prototype for accessing the on-chip temperature sensors.
- The Card Management Solution Subsystem (CMS Subsystem), a MicroBlaze-based design equipped with a firmware which autonomously reads sensor information from an external satellite controller mounted in the card.

Different temperature reads are associated with each Super Logic Region (SRL) found in the FPGA device. The platform-specific code in the library is used to compute a single value, as required by the API, by computing in software a weighted average which takes into consideration the actual occupancy of each SRL. For power values, the CMS Subsystem firmware polls for sensor information every 100ms from the external satellite controller. In particular, the physical reads are related to the voltage and current values associated with three power sources in the card, namely 12V-PEX, 12V-AUX, and VCCINT. In the library, we use them to derive instantaneous, maximum, and average power consumption values, then passed to the host-side local runtime manager as a single combined value. The initialisation of the platform requires a configuration file, found in the `etc` sub-directory, containing some information about the monitoring IPs contained in the hardware design. More detail on the hardware design of the RECIPE rFPGA custom shell are provided in Deliverable 4.3.

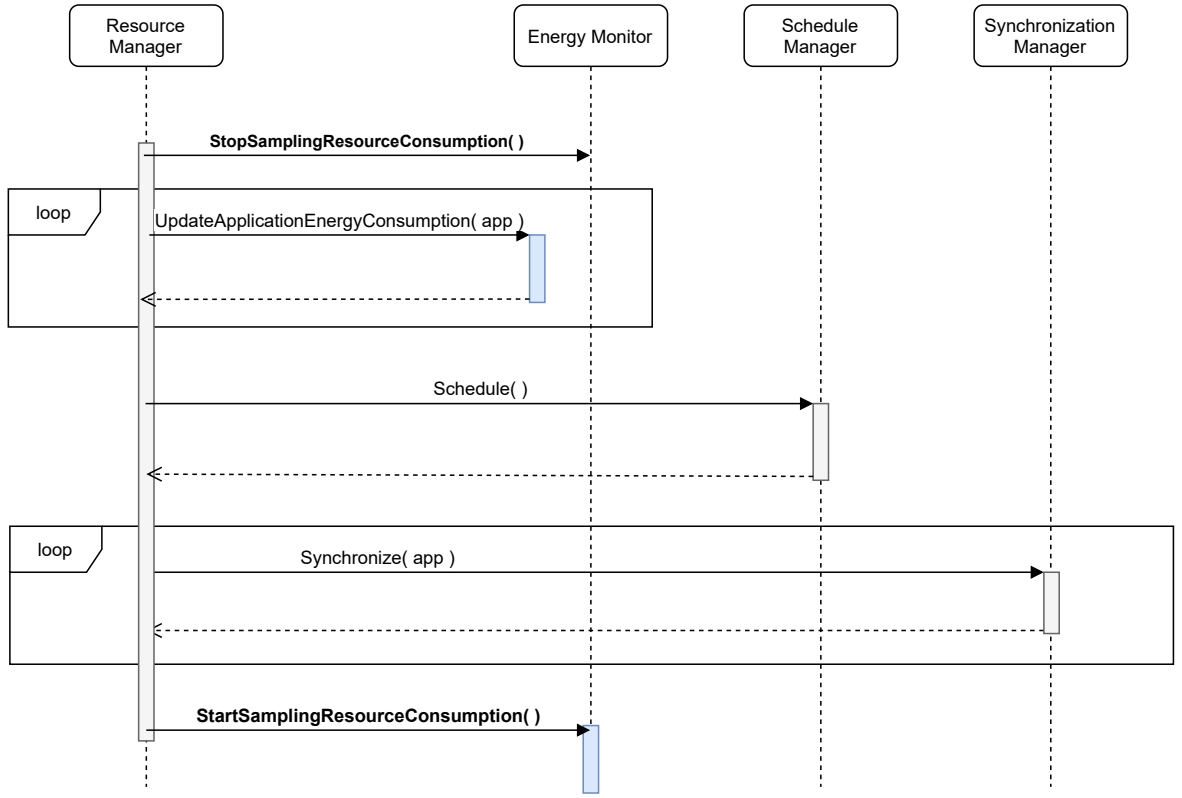



Figure 5: BarbequeRTRM resource allocation process with energy monitoring integration.

## 2.2 Energy Monitor

Part of the monitoring activity, enabled by the integration of the interfaces just described, is exploited by a new BarbequeRTRM module called *Energy Monitor*. This new component plays a key role in driving the local resource manager towards energy efficient resource allocation choices. In Figure 5, we sketched the sequence diagram of the core component of the BarbequeRTRM (*Resource Manager*), while coordinating the energy monitoring activity with the scheduling of the managed applications and the enforcement of the resource allocation decision.

When a new resource allocation procedure is launched, the current energy consumption monitoring is stopped (`StopSamplingResourceConsumption()`). In case of first resource allocation, this action has not effect. On the contrary, if the system has been working for a while, this provides the local resource manager with a snapshot of the energy consumption for each of the managed resources. The function `UpdateApplicationEnergyConsumption()` then is invoked for each managed application to estimate the quota of energy consumption due to the execution of the application. Hence, this profiling activity can be exploited by an energy-aware resource allocation policy, once invoked through the `Schedule()` function of the *Schedule Manager*. Once the resource allocation policy terminates, the invocation of `Synchronize()` from *Synchronization Manager* leads to enforcement of the resource assignments, by accessing the low level interfaces integrated in the Platform Proxy modules of each managed resource (e.g., CPU, GPU, FPGA accelerators). Finally, the energy consumption monitoring can be resumed by calling the `StartSamplingResourceConsumption()` member function of *Energy Monitor*. One last detail has been omitted for not overcomplicating the picture. The calls to the *Energy Monitor*'s

member functions are dispatched to all the Power Manager modules (e.g. `CPU_PowerManager`, `RECIPE_PowerManager`), triggering the specific start/stop action for each managed resource (member functions `StartEnergyMonitor()` and `StopEnergyMonitor()`).

Overall, what happens is that we provide the resource allocation policy with energy consumption profiles for each hardware resource and managed application, in the time interval elapsed since the last policy execution. The length of this period can be then configured on the basis of the specific target system.

### 2.2.1 Application Energy Profiling

The estimation of the energy consumption contribution due to each managed application relies on the profiling support provided with the Run-Time Library (RTLIB) (see D2.1, D2.2, D2.4) through the Adaptive Execution Model. We briefly recall the concepts behind this execution model. The application runs following the flow shown in Figure 6. The execution of the application is essentially synchronized with the BarbequeRTRM, such that for each update of the set of assigned resources, the application can jump in its `onConfigure()` function to reconfigure itself with respect to the new availability of resources.

The regular execution instead is characterized by cycles of `onRun()` and `onMonitor()` invocations. These functions execute, respectively, the core of the computation and (optionally) a custom performance monitoring routine. At the end of each cycle, the library generates a profiling report, including the counter of the cycles and an estimation of the CPU load generated by the application. This report is then sent to the resource manager.

At this point, the previously mentioned `UpdateApplicationEnergyConsumption()` function can combine this profiling information and the *Energy Monitor* data to produce an energy efficiency metric. This metric can be exploited by an energy-aware resource allocation policy. We called this metric *Energy-Per-Cycle (EpC)*, which is therefore an estimation of how much energy has been consumed, on average, for each application execution cycle, since the last resource allocation policy run. We will provide an example of exploitation of the EpC, by describing the energy-aware resource allocation policy in D3.7.

## 2.3 Reliability Management Integration

According to the DoW, we stated that, in order to properly maximize the reliability of HPC/Exascale computing platforms, we need to put in place a mix of *proactive* and *reactive* strategies. To this aim, the resource managers need to 1) access run-time hardware status information; 2) exploit low-level mechanisms to restore the status of a faulty execution. The first point is addressed by the monitoring components previously described. The status information retrieved will be then exploited as explained in WP3 deliverables and briefly mentioned later in this section.

The second point instead has been already discussed in deliverable D2.2, where we described the overall reliability management strategy and the implementation of the `ReliabilityManager` module of the BarbequeRTRM. In this deliverable, we focus on the Checkpoint/Restore mechanisms integrated or developed ad-hoc for workload running on CPUs or FPGA located processing units.

Checkpoint/Restore for GPUs instead deserves a separate chapter. While the need for transpar-

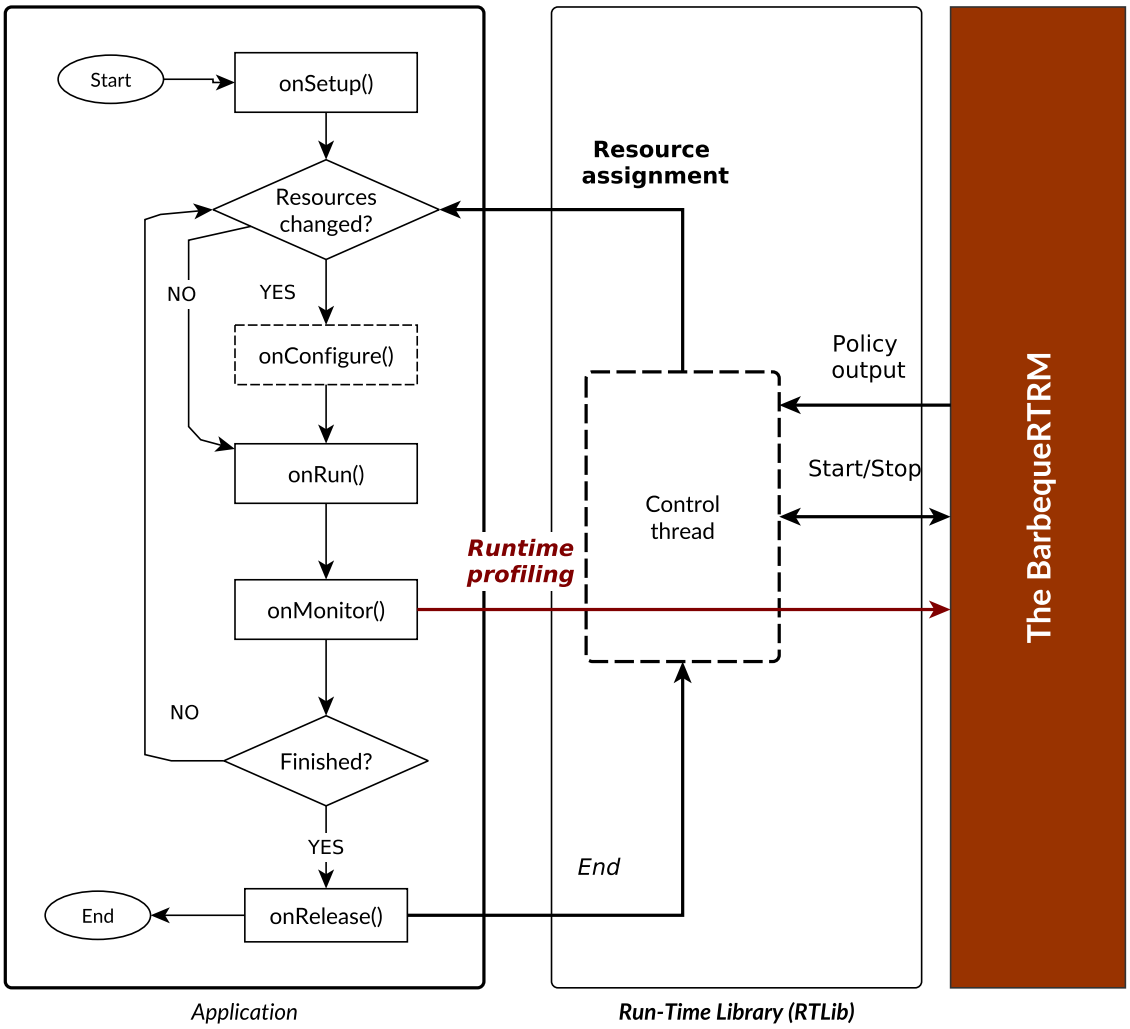



Figure 6: BarbequeRTRM Adaptive Execution Model for managed applications.

ent checkpointing of GPUs has grown in the last decade, the support for it has been dropped [4]. Since 2011, many efforts have been made to develop C/R tools for (NVIDIA) GPUs, but all of them stopped working with the release of CUDA 4.0, due the introduction of the *Unified Virtual Addressing (UVA)* between host and GPU device, later refined, with CUDA 6.0, to *Unified Virtual Memory (UVM)*. After these upgrades, all the previous checkpointing technologies became ineffective, since they created inconsistencies between host and GPU device address space during the restore of the checkpointed CUDA library and the associated allocated memory at their original address. Recently, two new tools, CRCUDA [11] and CRUM [3], were released, trying to solve the problem through the use of separate proxy processes, however exposing limitations in terms of overhead and partial support for UVM. To the best of our knowledge, only a recent DMTCP plugin, *CRAC* [4], has had successful results in the workaround of the UVM problem. However, although its source code is available [2], the project is for the time being at an embryonic state.

### 2.3.1 Checkpoint/Restore in Userspace (CRIU)

As already mentioned in D2.2, for the managed processes running on CPUs, we rely on the well-known *Checkpoint/Restore in Userspace (CRIU)* [1]. CRIU is in fact a software tool for Linux Operating Systems providing several features, starting from the simple Checkpoint/Restart functionality, arriving at the more complex live migration and remote debugging.

CRIU is currently integrated into a variety of software (e.g. OpenVZ, Docker, Podman) and it is packaged for many Linux distributions. It copes with the complexity of an in-kernel checkpoint/restore approach by implementing as many functionalities as possible in the user space, using existing interfaces to implement the services.

The services offered by CRIU can be exploited through three different interfaces: *Command Line Interface (CLI)*, *Remote Procedure Call (RPC)*, which uses *Google Protocol Buffers* to encode its calls, and a *C Application Program Interface (C API)* called `libcriu`.

BarbequeRTRM links `libcriu` in the Linux Platform Proxy, for the Checkpoint/Restore functionalities. In order to carry out the checkpoint routine, it exploits several functions, including `criu_set_images_dir_fd()`, `criu_set_log_level()` and `criu_set_log_file("dump.log")`, to manage target directory and log files, `criu_set_leave_running()`, in order to allow the resumption of the application after the completion of the checkpoint, and, of course, `criu_dump()` to perform the dump of the image.

The checkpoint procedure is carried out by CRIU in three main steps:

**Collect and freeze process tree.** It walks through the `/proc/pid/task/` directory to recursively collect the children in `/proc/PID/task/TID/children`. In the process, tasks are seized by `PTRACE_SEIZE` command.

**Collect and dump tasks' resources.** From `/proc` file system all the needed information is retrieved: VMAs areas are parsed from `/proc/PID/smaps`, mapped files are collected from `/proc/PID/map_files`, file descriptors from `/proc/PID/fd` and core parameters from `/proc/PID/stat`. Then, *parasite code* is injected to perform the dump.

**Cleanup.** The parasite code gets dropped out and the original code gets restored.

As for the restore functionalities, BarbequeRTRM makes use of `criu_set_ext_unix_sk()` and `criu_set_tcp_established()`, to handle possible socket and TCP connections, while resorting to `criu_set_evasive_devices()`, to manage paths become inaccessible. The actual restore is, then, initiated with the command `criu_restore_child()`.

In CRIU, the restore procedure is, as well, composed by a series of steps:

**Restore shared resources.** CRIU reads from the image files which processes share which resources and charges one process with their restore. At this point, the others can obtain the shared resources in a variety of ways, e.g. by inheriting them in a later session or by `SCM_CREDS` messages.

**Fork the process tree** CRIU calls `fork()` many times in order to re-create the checkpointed processes.

**Restore basic task resources** CRIU restores almost all resources (e.g. opening files, preparing namespaces, mapping the private memory areas), except (i) memory mappings exact

location (ii) timers (iii) credentials (iv) threads.

**Switch to restored context, restore the rest and continue** The exceptions above are made since CRIU needs to morph into the target process to complete the restore. This means that it has to unmap all its memory to map the target's and, in this act, the code doing the `munmap` and `mmap` must exist. For this reason, a small piece of code, independent of both CRIU's and the process' mappings, is launched to finalize the restore.

### 2.3.2 Advanced C/R Engine Library

Checkpoint/Restore for FPGA resources instead has required considerations similar to the FPGA Monitoring Library case. We designed and implemented an additional library, called *Advanced C/R Engine Library (libacre)*, providing a well-defined API to both the local resource manager and the developers of the platform-specific support. Differently from the *libfpgamon*, the ACRE Library code base has been written in C++. Therefore the API is mainly exposed by the virtual class `CheckpointRestore`, as show in Listing 4.

Listing 4: acre.h

```
1  class CheckpointRestore
2  {
3
4  public:
5     using image_location_t = std::string ;
6     using task_id_t = int
7     using hardware_id_t = std::string;
8
9     virtual ~CheckpointRestore() {};
10    virtual void checkpoint(const hardware_id_t &, task_id_t &) = 0;
11    virtual void restore(const hardware_id_t &, task_id_t &) = 0;
12    virtual void freeze(const hardware_id_t &, task_id_t &) = 0;
13    virtual void thaw(const hardware_id_t &, task_id_t &) = 0;
14    void set_image_path(const image_location_t &) noexcept;
15    image_location_t get_image_path() const noexcept;
16
17 private:
18    std::string image_path;
19
20 };
```

All the member functions provided by the class take a pair of identification numbers, one for the hardware platform and the other for the specific running task, which could have been directly implemented in hardware. The set of control actions, for which the class requires the platform-specific implementations, include performing the checkpoint of a target task, freeze/thaw the execution of the target task, restore the execution given a previous checkpoint image. The path of the checkpoint images can be configured through the `set_image_path()`.

This means that for each platform-specific support we expect the developers to extend the library code base, by introducing the definition and implementation of additional classes derived from `CheckpointRestore`. Therefore, on the user side, we can instantiate the specific derived class,

representing the interface towards the FPGA platform actually installed into the system.

**FPGA Checkpoint/Restore support.** Platform-specific support in the library includes the contribution for FPGA Checkpoint/Restore provided by CERICT for the COTS-based prototype. In that respect, CERICT has developed two low-level libraries: 1) the *XDMA Communication Library*), providing the low-level functions used to interact with the Alveo FPGA card, and 2) the *XDMA Checkpoint Library*, containing several functions and structures used to provide the FPGA Internal Configuration Access Port Controller with the information needed to perform a partial reconfiguration or a readback of a dynamic region on save/restore operations. The *XDMA Checkpoint Library* functions are declared in `xdma_checkpoint.h` and are independent of the particular dynamic region, while design-specific information is provided by means of two structures: 1) `checkpoint_info`, containing high-level information such as the location and size of the golden partial bitstream, the readback file, and the readback bitstream, as well as low-level information related to the internal structures of the bitstreams, e.g. the number and the offsets of the write sequences in the bitstream; 2) `conf_write_sequence`, containing the mapping between the golden partial bitstream and the full bitstream, which is necessary to update the contents of a partial bitstream with the dynamic execution state information extracted from the FPGA. The mapping information is static and can be stored in a configuration file. Note that all the functions interacting with the ICAP Controller require the *XDMA Communication Library* to be installed. The XDMA Communication Library and the XDMA Checkpoint Library are thoroughly described in Deliverable 4.3 as a part of the RECIPE FPGA acceleration middleware.

The integration in the ACRE Library is supported by two header files, located in the `src/xdma` and `include/xdma` sub-directories:

- `XDMA_Target.h` includes the header files from both the XDMA Communication Library and the XDMA Checkpoint Library.
- `libacre_xdma.h` contains the declaration of the `XDMACheckpointRestore` class, which is used to implement the `CheckpointRestore` interface.

### 2.3.3 Reliability Manager Developments

The finalization of the integration layers of the target platforms and the availability of the aforementioned Hardware Reliability Library has enabled the introduction of the last developments into the *Reliability Manager* module of the BarbequeRTRM. The initial version of this component of the local resource manager has been described in D2.2. We improved the previous implementation by operating on two aspects:

1. Completing the hardware resource monitoring support with the exploitation of the *Hardware Reliability Library*
2. Adapt the checkpoint rate at run-time on a per-application basis.

The first point allows the reliability-aware resource policies to monitor the status of the hardware and then trigger a suitable management action, as already described in D2.2. In other words, in case of unreliable processing cores, the policy can decide to migrate the workload occupying such resources, to freeze the execution of some tasks or to control the processor by switching-off (or idling) the involved cores, or scaling the voltage-frequency pair.

More specifically, we integrated the access to the Hardware Reliability Library (`libhwrel`) functions, such that the local resource manager can periodically check for hardware reliability prediction. This functionality must be properly enabled via the `make menuconfig` command of the BOSP build system. Similarly to the FPGA Monitoring Library, also the Hardware Reliability Library provides a well-defined interface to the resource manager, while leaving the possibility of implementing multiple models under the hood. Currently, the only model provided is the one developed by BSC in the WP3, called `bschwrel`. At run-time, in order to properly feed the `libhwrel` with the required data, the BarbequeRTRM needs to read the performance counters of the system. This is done by the corresponding `PlatformProxy` (`LinuxPlatformProxy` in our case) function, which makes the hardware call transparent to the `libhwrel`.

For the second point instead, in D2.2 we started with a strategy based on performing the checkpoint of the managed applications, on a periodical basis, with the length of the period set to a fixed (configurable) value. Now, other than keeping the possibility of configuring the BarbequeRTRM with this option set, we introduced the *Dynamic Checkpoint Scheduler*. This extension has made the reliability management strategy run-time adaptive and aware of the application requirements. The Reliability Manager in fact, can now define a per-application checkpoint rate, based on the status of the platform and the impact on the application execution, as it will be briefly explained in 4.3.

Further details on the policy implementing this adaptive checkpoint rate will be provided with deliverable D3.7.

## 2.4 Timing Analysis Integration

The timing analysis has been integrated by introducing a suitable library, the *Timing Analysis Library* (shorten `libta`, already described in detail in Deliverable D3.4) and then by adding the needed glue logic in the BarbequeRTRM. We briefly recall the structure of the library here. The main component is represented by the abstract class `TimingAnalyzer`, which provides a common interface for enabling the integration of multiple timing analysis tools. An object derived from `TimingAnalyzer` should take input objects of type `Request` and produce a `Response` object as output. The `Response` can then be specialized in `ResponsePWCET` or `ResponseWCET` depending on whether the output is a distribution or the Worst-Case Execution Time (WCET) value only. Starting from this library interface, we implemented two timing analyzers for Measurement-Based Probabilistic Timing Analysis (MBPTA):

- `bscta`: developed by BSC and implementing the MBPTA-CV technique, as described in D3.4; and
- `chronovise`: developed by POLIMI and implementing the traditional MBPTA techniques as described in the related paper [10].

The implementation of these two tools demonstrated the flexibility of the library interface to enable the integration of any timing analysis tool.

Having a common and general interface for the timing analysis tools, we extended BarbequeRTRM to integrate the execution time collection, the call to the timing analysis library, and the storing of the results in order to give the resource management policy access to them[1].

[1]The details on how a resource management policy exploits such data is a subject of the deliverable D3.7 at

In particular, each `Application` object in the BarbequeRTRM, represents a descriptor of a AEM-integrated application (see D2.2, D2.4). This object includes also the application profiling information (`struct RuntimeProfiling_t`). We extended this set of information by adding the the `pwcet` sub-structure shown in Listing 5.

Listing 5: The `pwcet` sub-struct

```
1 struct
2 {
3    std::list<int> cycle_times; // In ms
4    double wcet; // pWCET @ 10^-9
5    double mu; // pWCET - location
6    double sigma; // pWCET - scale
7    double xi; // pWCET - shape
8 } pwcet;
```

The first member of the structure (`cycle_times`) is filled with the execution time values of each `onRun` of the application (see Figure 6). When a sufficient number of execution time values are collected, then the `libta` analysis execution function is called by the Application object itself.

However, to determine the minimum number of execution time values is non-trivial. On the BarbequeRTRM side, a constant threshold is selected at compile-time, which is usually set very low, so still insufficient for the `libta` to produce an estimation of the (p)WCET. However, it helps to avoid useless calls to the `libta` when the number of samples is low, which would obviously generate a "too low number of sample error". In fact, each implementation of the `libta` defines its own function to determine the minimum amount of samples required. For instance, `chronovise` uses a sequence of statistical tests to determine the validity of the generated (p)WCET. The minimum amount of samples required is a non-constant value, and usually depends on the specific application and system under analysis.

Finally, when the `libta` function returns a successful value, the requested WCET or pWCET is stored in the above structure, respectively in `wcet` (computed on the pWCET distribution at a probability of violation of $p = 10^{-9}$, and the three variables `mu`, `sigma`, `xi` representing the three parameters of the Generalized Extreme Value Distribution or Generalized Pareto Distribution.

A resource allocation policy can thus exploit the results of such analysis, by considering the relationship with the assigned resources or the presence of co-scheduling conditions. As a consequence, the policy can revise the allocation of resources in order to meet the actual application timining requirements. This is the goal of the timing-aware resource allocation policy that will be explained in D3.7.

the end of the project.

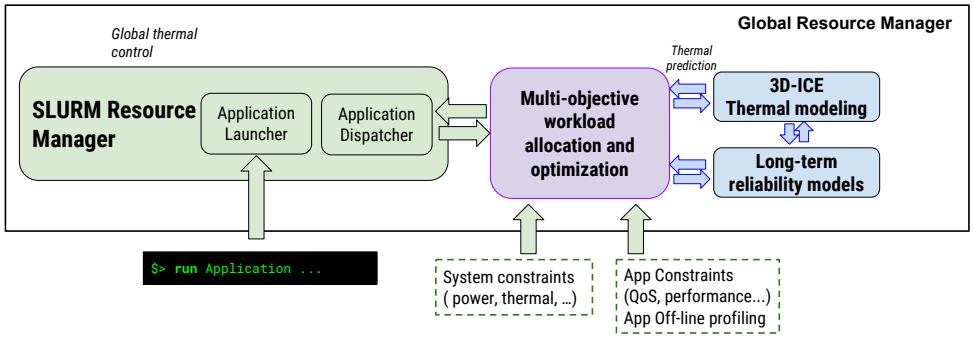



Figure 7: Overview of the Global Resource Manager (SLURM) with global scale software components.

# 3 Global Resource Manager: SLURM

In this deliverable we report the changes performed in the infrastructure of the Global Resource Manager (GRM) with respect to those provided as part of D2.3. More specifically, we provide a global picture of how the GRM works as part of the integration with the remaining SW stack of RECIPE, and provide details of the integration with the Local Resource Manager (LRM) and with the other SW tools (such as HELENNA) that are integral part of the RECIPE platform. To showcase this, we focus on Use Case 3 (Epilepsy detection application), which exploits the full SW/HW stack, using the GRM, LRM and HELENNA, while also exploiting all the heterogeneous resources (FPGA, GPU and CPU - with and without MKL libraries).

The final assessment of the energy savings achieved by the overall SW/HW stack will be provided as part of D1.9. In this deliverable we limit ourselves to explaining the status of SW developments to achieve integration.

## 3.1 Platform Deployment and Integration

The overall picture of the GRM is the same than the one provided in D2.3, as can be seen in Figure 7.

One of the main changes is the way in which the SLURM GRM is deployed. In order to ensure that checkpointing remains consistent across nodes, instead of deploying the different slurmd daemons as a docker container, we rolled back to a native deployment of slurmd daemons, while keeping the slurmctld controller as a container, still deployed using docker swarm. A shared unit is also put in place so that In this way, we aim at keeping the solution lightweight and easy to deploy, while still providing the required capabilities for checkpoint and restore. This is discussed in greater detail in the next subsection.

In order for the GRM to be able to correctly understand and list all the resources available, we stick to the same procedure used in D2.3, which consists on using a cluster architecture builder, which lists the available resources as a JSON file, as shown 6

Listing 6: Example of initial architecture input file

```
1 {
```

```
  "name": "RECIPE Cluster",
  "version": "1.0.1",
  "master": "gn0",
  "architecture": {
    "recipe0": {
      "id": "recipe0",
      "master": 1,
      "components": {
        "fpga-acc1": 0,
        "fpga-acc2": 0,
        "fpga-acc3": 0
      }
    },
    "recipe1": {
      "id": "recipe1",
      "master": 0,
      "components": {
        "gpu": 0,
      }
    },
    "recipe2": {
      "id": "recipe2",
      "master": 0,
      "components": {
        "gpu": 0,
      }
    }
  }
}
```

The cluster file specified in JSON is internally parsed to change the configuration files of slurm and understand the available resources internally. Therefore, SLURM will be able to transparently understand the existence of new available HW resources that were not still supported in D2.3, such as FPGAs.

Policies for choosing the available resources and cluster availability are therefore updated in the same way than before, that is, by creating a directional graph where all resources are constantly updated (via a *Networkx* object that is instantiated in this step has to be a DiGraph), as shown in Figure 8.

This architecture is kept as it was, as it allows to integrate new hardware resources and fine tune policies. The major changes to the structure of the GRM are therefore provided below.

## 3.2 Local Resource Manager and HELENNA Integration

The new version of the GRM allows integration not only with the LRM, but also with the underlying HELENNA platform in a transparent way, so that resources that run over HELENNA

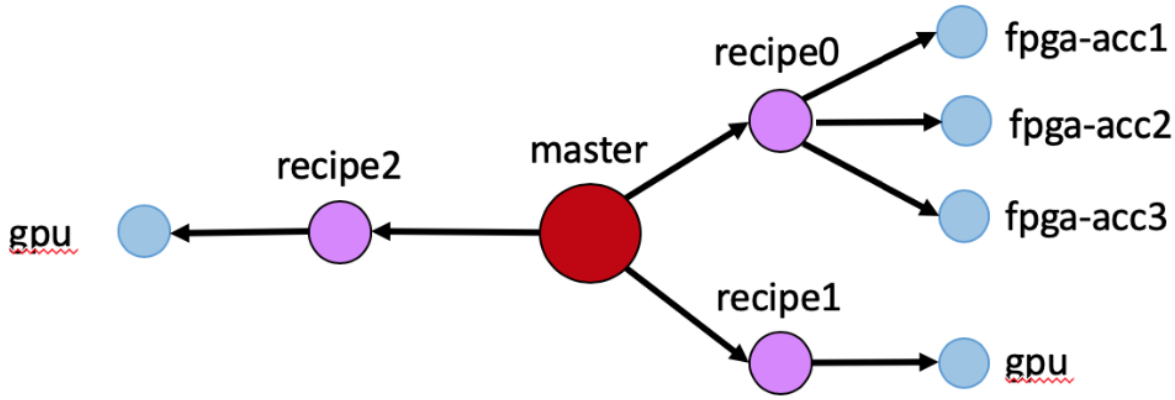



Figure 8: Network obtained after creating the initial network architecture given the configuration file shown in Listing 6

can be acquired via the GRM. In this sense, SLURM provides support for both applications running using the Adaptive Execution Model (AEM) (which is described to a greater extent in the next section) and using HELENNA.

The way in which this allocation is performed is as follows:

- The JSON cluster architecture file has to be properly filled-in with the available resources in the cluster. This process can be automated with the use of scripts, but is currently performed in a manual way.
- Creating the appropriate SLURM-compliant sbatch scripts to launch the workloads.
- Within the sbatch scripts, it suffices to call applications already integrated with the AEM through the BBQ RTRM command line.
- Within the sbatch scripts, to launch applications that require HELENNA support, it suffices to call applications through the HELENNA command line. This implies that, for instance, for the specific case of UC3, we first need to provide the description of the CNN using the format of HELENNA (creating input file in the HELENNA format, as described in deliverable D1.8) and then calling HELENNA through SLURM using the sbatch script, or by simply calling HELENNA within srun.

The assessment of overheads and performance of the overall software stack will be provided as part of D1.9.

## 3.3 Integration and support for consistent checkpoint/restore

One of the challenges within the integration of GRM and LRM is keeping consistent copies of the checkpoints that can be created at the local level by the LRM at the global level, making sure that restore is consistent across nodes. To tackle this issue, we provide two different capabilities at the GRM level:

- Firstly, we create a common shared disk space that is controlled and created by the SLURM controller, to ensure that there exists a shared and consistent place where checkpoints can be created and globally stored.
- Secondly, similarly to what we did when listing availability of nodes for allocation, we also create a *Networkx* object that holds the status of the checkpoints at the global level. However, in this case, instead of marking resources as available or unavailable, we update the graph and assign weights to the resources depending on how old the checkpoints assigned to them are.

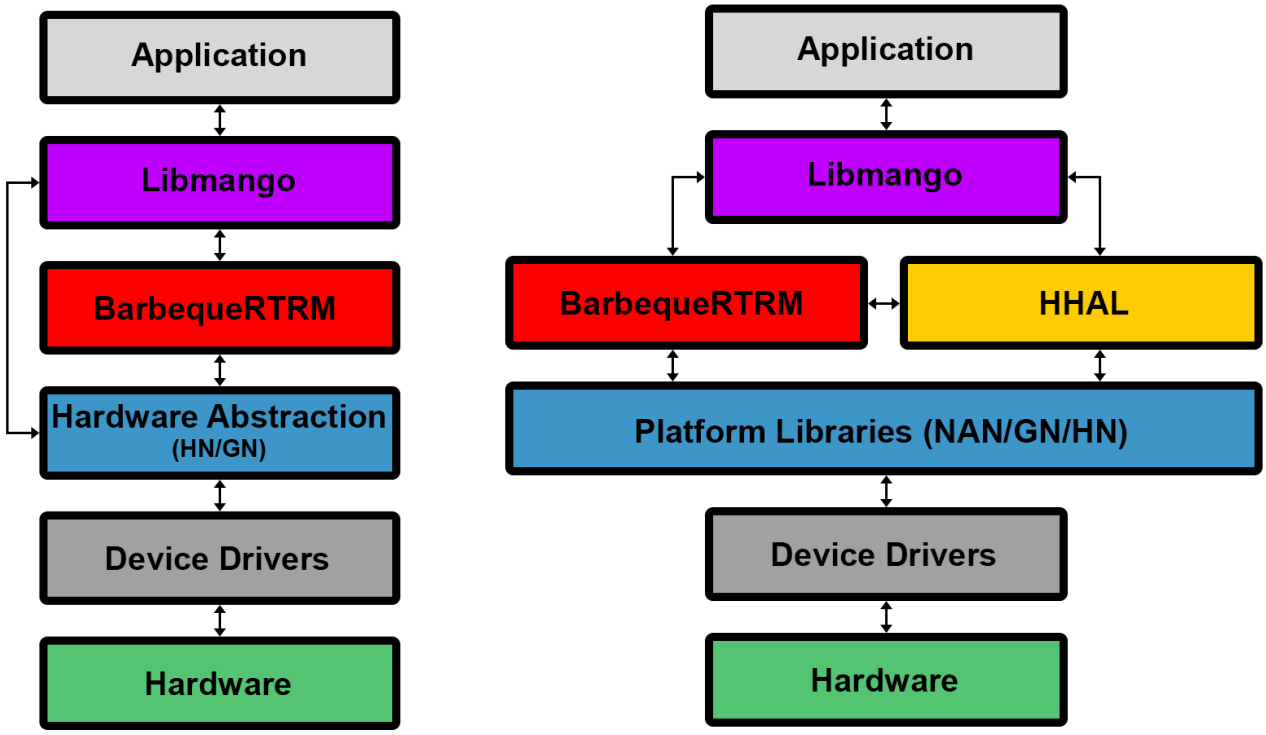



Figure 9: Comparison of the original and refactored `libmango` architecture

# 4 Programming Models Integration

## 4.1 MANGO Programming Library refactoring

The MANGO Programming Library (`libmango`) was developed as part of the MANGO project to enable the programmability of deeply heterogeneous architectures, primarily targeting the MANGO hardware cluster. As such, its software architecture was tied to the MANGO hardware architecture and its abstraction layer, and not suitable for managing multiple architectures at the same time (while the original hardware abstraction layer supported both the MANGO node and a general purpose node, the two could not be employed at the same time by the resource manager). In RECIPE, we performed a full refactoring of `libmango` to achieve greater architectural independence, and to better isolate the hardware platform abstraction layer from the programming model implementation, thus reducing the coupling between them and the resource manager as well. Figure 9 depicts the original architecture of `libmango`, and the refactored one, showing how the introduction of a new component, the Heterogeneous Hardware Abstraction Layer (HHAL), enabled the introduction of multiple concurrent platform libraries.

The new component, HHAL, is described in more detail in Figure 10. One key feature is the introduction of a dynamic compilation support, as well as the platform managers, which allow the coexistence at runtime of multiple accelerators.

In addition to these changes, other, less visible modifications entailed a refactoring of internal data structures to simplify them and reduce the overheads, as well as the introduction of a new binding for the Python programming language, which allows the development of `libmango`

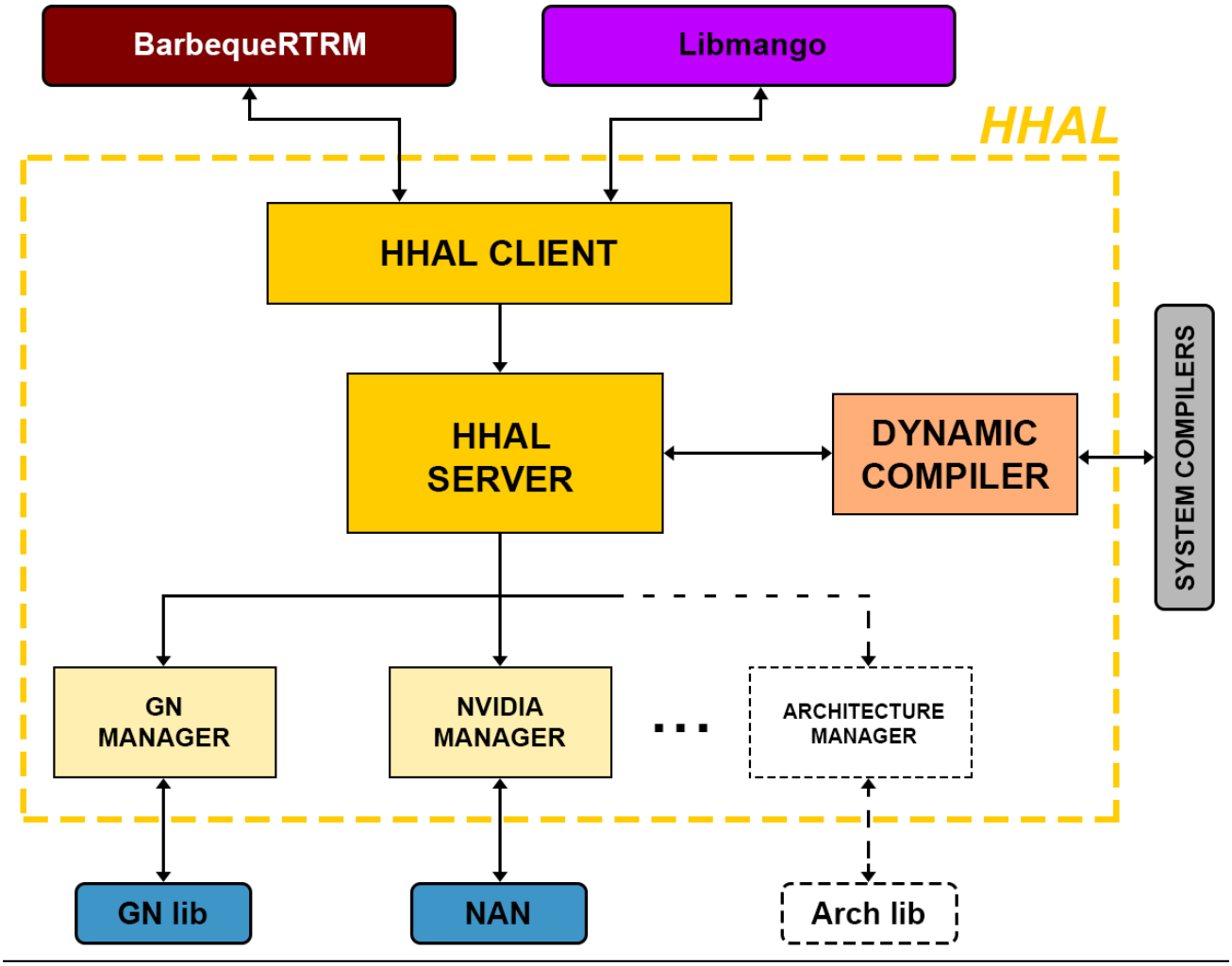



Figure 10: Architecture of the HHAL

applications in Python (for the host code – device code is constrained by the specific device).

### 4.1.1 NVIDIA Platforms Integration

One key advance enabled by the refactoring of `libmango` is the ability to support off-the-shelf accelerators in addition to the MANGO cluster. In particular, we demonstrated this capability by developing an Architecture Node to support Nvidia GPU accelerators. The main role of an Architecture Node in `libmango` is to map the `libmango` primitives to those specific of the target architecture. Figure 11 depicts the role of the Nvidia Architecture Node (NAN) in translating the MANGO primitives to the CUDA APIs.

## 4.2 Adaptive Execution Model (AEM) Fortran wrapper

The Adaptive Execution Model (AEM) is a general-purpose resource-managed execution model. In AEM, the application is designed to run its computationally intensive activities within a controlled workflow divided in the five stages shown in Figure 6, which are, in the C++ binding, obtained by implementing the following methods:

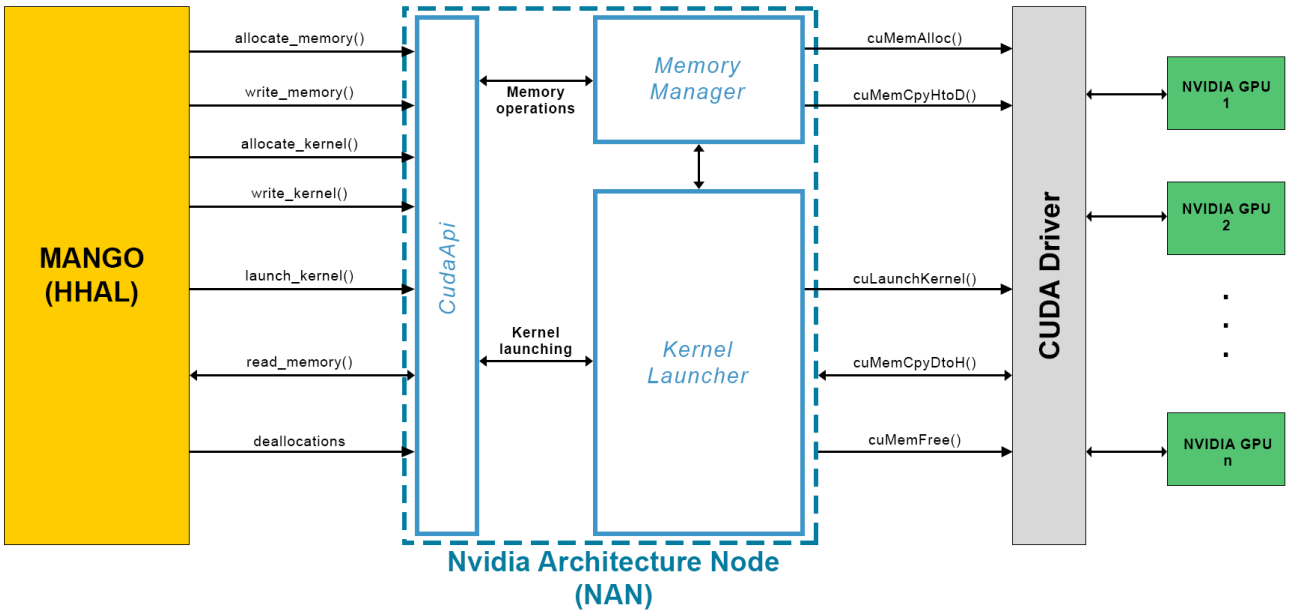



Figure 11: Nvidia Architecture Node

**OnSetup** activities performed during the setup of the kernel;

**OnConfigure** activities performed when assigned resources are changed;

**OnRun** activites performed during each round of the kernel;

**OnMonitor** activities performed at the end of each round of the kernel (typically for performance monitoring);

**OnRelease** activities performed at the kernel ending.

To expose these to Fortran, a wrapper has been designed. The Fortran compilers employ the same mangling scheme as C, with some minor differences. Parameter passing is also compatible with C. However, AEM is developed in C++, so some adaptation is needed to expose a C interface compatible with the Fortran compilers. In particular, a `bbqFortranMain` function is exposed to perform the generation of the AEM object and a `BBQGetAssignedResources` function is exposed to allow the application to inspect the assigned resources.

The scheme of an example application is reported in Listings 7 and 8. Listings 7 reports the (very simple) main code which contains application code not controlled by the resource manager (none here) and the call to the resource managed code. Listings 8 contains the implementations of the five stages of the managed code. The management loop in `bbqFortranMain` is generated automatically.

Listing 7: Example Fortran AEM Application.

```
1 program main
2   print *, "FORTRAN: main: start"
3   call bbqFortranMain("HelloFortran")
4   print *, "FORTRAN: main: end"
5 end program main
```

Listing 8: Example Fortran AEM Application.

```
! Initialization
      subroutine BBQonSetup(*)
         real alpha, beta
         common /coeff/ alpha, beta
         alpha=1.0
         beta=5.1
         print *, "FORTRAN: onSetup"
         return 0
      end subroutine BBQonSetup

! Configuration
! Configure parallelism and/or input data related parameters
      subroutine BBQonConfigure(*)
         use RTLIB_ResourceType
         real alpha, beta
         integer proc_quota
         external BBQGetAssignedResources
         common /coeff/ alpha, beta
         print *, "FORTRAN: onConfigure: ", alpha, beta
         call BBQGetAssignedResources(PROC_ELEMENT, proc_quota)
         print *, "proc_quota: ", proc_quota
         return 0
      end subroutine BBQonConfigure

! Run
! start the execution; computational code should be here
      subroutine BBQonRun(*)
         use RTLIB_ExitCode
         real alpha, beta
         integer ret, BBQCycles
         external BBQCycles
         common /coeff/ alpha, beta
         print *, "FORTRAN: onRun", alpha, beta

         ret = 0
         ! Return after 5 cycles
      if (BBQCycles() >= 5) then
         ret = RTLIB_EXC_WORKLOAD_NONE
      end if

      return ret

      end subroutine BBQonRun

! Monitor
! Get information needed to decide whether to reallocate resources
      subroutine BBQonMonitor(*)
         real alpha, beta
```

```
49          common /coeff/ alpha, beta
50          print *, "FORTRAN: onMonitor", alpha, beta
51          return 0
52        end subroutine BBQonMonitor
53
54  ! Cleanup
55  ! Deallocate as needed (probably nothing), also write out data, etc.
56        subroutine BBQonRelease(*)
57          real alpha, beta
58          common /coeff/ alpha, beta
59          print *, "FORTRAN: onRelease", alpha, beta
60          return 0
61        end subroutine BBQonRelease
```

## 4.3 Adaptive Execution Model (AEM) and Reliability management

In D2.2, we mentioned the need of allowing the application to specify their reliability requirements such that for each application we can dynamically set the most suitable checkpoint period. During the actual design and implementation phase, we decided to follow a different approach, based on the idea of setting an upper bound to the checkpoint overhead, leaving the local resource manager in charge of maximizing the reliability, given the overhead constraint.

With this idea in mind, we extended the application *recipe* file in order to allow the user to set up the reliability support we integrated in the BarbequeRTRM. In the following, a brief overview of such support will be provided, leaving a deeper analysis to Deliverable 3.7.

As explained in Section 2.3.3, we introduced in the framework the *Dynamic Checkpoint Scheduler*, as part of the Reliability Manager. This development allows the user to set an application-specific upper bound for the checkpoint overhead. This value is defined as the desired ratio of the checkpoint time and the total application time. In the *recipe*, when the *Dynamic Checkpoint Scheduler* is enabled, such value can be specified through the tag `<chk_overhead ratio=""/>`. The checkpoint scheduler will manage the trade-off between checkpoint overhead and application performance requirements accordingly.

A second development, not mentioned yet, is the reliability-aware policy. This is a work in progress, but at the time of the release of the deliverable we already extended the application *recipe* file, to enable a per-application configurability of the resource allocation policy. Being part of the policy made by a PID-based controller, in order to adapt the allocation of the computing resources, we introduced the possibility of specifying a different set of values of the PID parameters for each application. The parameters of such controller, i.e. the proportional, integral and derivative contributions, respectively, $k_p$, $k_i$ and $k_d$, are set through the recipe using the tag `<pid_controller kp="" ki="" kd=""/>`.

Both modules are set with default values if the dedicated tags are not provided. An example of application *recipe* providing the setting of the reliability support is shown in Listing 9.

Listing 9: Example of application Recipe.

```
1  <?xml version="1.0"?>
2  <BarbequeRTRM recipe_version="0.8">
```

```
3    <application priority="4">
4      <platform id="org.linux.cgroup" hw="mango">
5        <awms>
6          <awm id="0" value="1" config-time="150">
7            <resources>
8              <cpu>
9                <pe qty="100"/>
10               <mem qty="2" units="MB" />
11             </cpu>
12             <net qty="50" units="Kbps">
13           </resources>
14         </awm>
15         <chk_overhead ratio="0.2"/>
16         <pid_controller kp="0.5" ki="0.3" kd="0.2"/>
17       </awms>
18     </platform>
19   </application>
20 </BarbequeRTRM>
```

## 4.4 Integration of FPGA heterogeneous acceleration

This section describes the integration of the runtime management and remote direct accelerator memory access (RDaMA) mechanisms explored in RECIPE based on PCI Express peer-to-peer interactions and made available to user applications through a high-level parallel programming model. The model of choice is the popular Message Passing Interface (MPI) API, which is widespread and commonly used for large-scale HPC applications. Figure 12 shows the RECIPE software stack supporting the integration of heterogeneous acceleration, monitoring functions, and remote direct accelerator memory access capabilities.

We addressed several choices for the actual implementation of MPI, showing the applicability of the infrastructure developed in RECIPE for various MPI runtimes, but also identifying the best match for the different technolgies explored in RECIPE. For the MANGO-based prototype developed by UPV, we selected the MVAPICH2-GDR implementation of the MPI standard. It features built-in CUDA support, which was readily extended to provide MANGO support, as thoroughly explained in Deliverable 4.3. Based on this type of support, we can send and receive memory buffers directly among the host, the GPUs, and the FPGAs of the MANGO cluster. For the COTS-based prototype developed by CERICT relying on Xilinx acceleration cards, we explored a layered software infrastructure, showing the potential of the Unified Communication X (UCX) framework as an underlying communication support for generic MPI implementations (in our case, OpenMPI), improving the portability and applicability of the solution developed in RECIPE. Some low-level details of the setup for the latter scenario are given in the following subsection.

### 4.4.1 OpenMPI relying on UCX for RDMA support

The RECIPE prototype located at CERICT was equipped with the OpenMPI runtime over UCX, which offers an optimised communication layer for the MPI libraries. In our hardware setup, we have two different nodes (called CERICT-RECIPE-0 and CERICT-RECIPE-1, referred to as

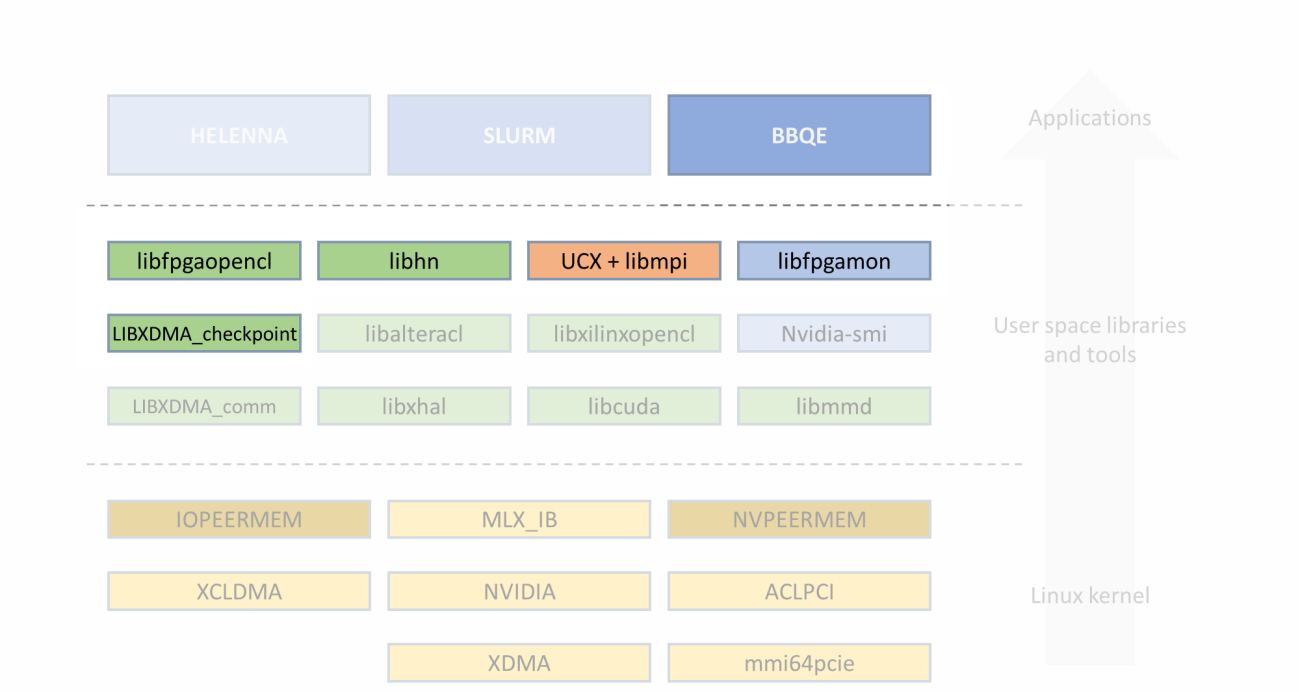



Figure 12: RECIPE software stack for heterogeneous acceleration, monitoring, and remote communication (the whole set of components in the stack is fully described in Deliverable 4.3)

*Server 0* and *Server 1* here). Server 0 features an FPGA card, so it is responsible for exposing the FPGA memory for remote access. The two servers are connected to each other through an Infiniband EDR link, using two Mellanox Host Channel Adapters (HCAs), as planned in the Grant Agreement. Thanks to the UCX layer, OpenMPI is able to use the Infiniband drivers to perform RDMA through this channel. A basic FPGA design that can be used to test RDMA on FPGA memory can be found on CERICT Server 1.

In general, the FPGA resources, including the card memory and any component in the FPGA which is memory mappped, can be accessed through the PCIe bus by means of the Xilinx DMA (XDMA) driver and an associated FPGA-side component, also called XDMA, acting as a DMA controller. As an alternative explored in RECIPE, we disabled the DMA capabilities of the IP, just using the XDMA IP to expose device memory, and relied on the InfiniBand HCA for direct remote memory access. So, two alternative scenarios were considered:

- Using RDMA with XDMA drivers: in this situation, the FPGA memory is not directly accessible by the remote node, but the remote communication is performed from the remote memory of Server 1 to the local (host) memory of Server 0. Then, data is transferred to the FPGA memory using the XDMA drivers, i.e., the transfer to FPGA memory is handled locally.
- Using RDMA with PCIe Peer-to-Peer (*peer direct*): in this case, the FPGA memory is directly accessible from the remote node, without passing through the local memory.

The custom FPGA shell developed in RECIPE, rFPGA, described in Deliverable 4.3, is configured in such a way to enable the second scenario. In this case, we need to enable the

M_AXI_BYPASS interface on the XDMA, since it is the interface actually used to access the FPGA memory when using both RDMA and PCIe Peer-to-Peer, while the XDMA drivers are not used at all. In fact, the interaction with the FPGA memory is not achieved using the XDMA device files, but using the `resource2` file, located under `sys-fs`, following the path associated to the FPGA card `/sys/bus/pci/devices/0000:3b:00.0`. This file represents all the memory resources accessible using the XDMA.

The above mechanisms have been made accessible to MPI applications, particularly to the OpenMPI runtime based on UCX. Like any MPI code, OpenMPI applications are made of different processes, which belong to the same *communicator* and have each a unique associated identifier. The identifier can be used to differentiate the behaviour of each process. The processes can be launched on the local node or on a remote node.

In the CERICT prototype, some practical settings need to be applied for the setup to work, as listed below. In a production environment, the system administrator would be in charge of checking such requirements while configuring the machine to enable the RECIPE mechanisms.

- An instance of OpenSM must be running on one of the servers (command `sudo opensm`).
- The HCA adapters must be addressable on the involved servers. This can be achieved, for example, with the command `sudo ifconfig ib0 10.0.0.2/24`, using different addresses for the local node and the remote node.
- If using PCIe Peer-to-Peer, we need to make sure that the FPGA memory can be registered and exposed as a remote memory area. To do that, we need to mount the `io_peer_mem` kernel module [2].
- The MPI runtime executing on the local node (the one launching the application) will spawn the remote processes using an SSH connection. However, to enable that, the local node must be able to identify itself to the remote node using a certificate, e.g. running a command like `ssh-add id_mpi` (the authentication via password cannot be used in this case).
- The executable must be placed at the same path for both the local and the remote node, so that it is possible for the MPI runtime to locate it when launching the application.
- Using XDMA drivers or the `resource2` file to access FPGA memory requires root privileges. However, running an MPI application with root is not allowed by default and it is discouraged. For this reason, it is required to enable any user to read and write the necessary files before running the MPI application. For example, you can use

  ```
  sudo chmod 666 /dev/xdma0_h2c_0
  ```

  to allow the application to use channel 0 of the XDMA to perform write operations to the FPGA memory. This step has to be performed whenever the FPGA is programmed.

For compiling MPI applications, the `mpicc` command must be used, while `mpirun` is used for launching the execution. The parameters passed to `mpirun` are used to require the OpenMPI runtime to execute over UCX and use the HCAs and the InfiniBand drivers. They are also used to identify the servers on which the application will run.

[2] The `io_peer_mem` kernel module is not strictly necessary for the application to work. However, when not using it, the performance of peer-to-peer transfers is degraded.

**Communication models.** An MPI application can be structured according to different communication models. One aspect we need to be aware of is the distinction between an *eager* protocol and a *rendezvous* protocol:

- the *eager* protocol is preferred when a small amount of data is to be transferred. In this case, the sender does not have to wait for the receiver to allocate a buffer to receive the data. Instead, data is immediately sent to the HCA, which can buffer them until the receiver in ready for reading.
- the *rendezvous* protocol is used with larger amounts of data. In this case, the sender has to wait for the receiver to explicitly allocate a buffer before sending data.

Using the UCX layer, OpenMPI is able to automatically switch between the two protocols, based upon the amount of data involved in a communication. However, it is possible to directly control the threshold for this switch by passing a few additional parameters to the `mpirun` command.

### 4.4.2 Demo Stencil application

Stencil computation is a common kernel in many HPC applications, including computer simulations for scientific and engineering applications, e.g. for fluid dynamics, partial differential equations, cellular automata, image processing, etc. Stencil computation is also a core part of RECIPE Use Case 1. Generic iterative stencil codes work by sweeping a multi-dimensional data structure, typically a two- or three-dimension grid, in each iteration by using a fixed-size window, updating the whole structure for the next iteration. The code is representative of computation kernels and memory access patterns that are typically found in HPC codes. The local and remote communication mechanisms enabled by RECIPE for dedicated FPGA acceleration provide an optimal fit for such patterns. In Deliverable 4.3 we show the detail of an FPGA accelerator handling the computation and data access for a demo stencil application, particularly a five-point Jacobi stencil, matching the potential of dedicated acceleration and High-Bandwith Memory (HBM) technology available for local (on-card) communication. The infrastructure described in Deliverable 4.3 also allows the FPGA card memory to be directly exposed across the InfiniBand fabric in order for other nodes in the system to access the card memory and the stencil acceleration functionality.

For the purposes of this deliverable, we have exploited this possibility by writing a distributed MPI-based application based on the above-described software stack, particularly on the OpenMPI runtime, which in turn relies on the UCX communication framework and hence on RDMA capabilities. The application assumes the availability of distributed FPGA acceleration nodes featuring stencil acceleration. Each acceleration node is assigned a *tile* of the large matrix processed by the overall MPI-based stencil application. Each tile, having a size in the order of GigaBytes of memory, is transferred by means of RDMA mechanisms directly to the accelerator memory, the HBM memory in the case of our experiments. For parallelizing the stencil application, standard programming practices have been adopted, including the definition of a halo region redundantly transferred with each tile to the acceleration nodes ad properly sized according to the number of iterations executed independently by each node. Tiling is performed row-wise. Figure 13 shows a sequence diagram of the demo stencil application, highlighting in a compact form the main MPI primitives used in the code.

The application was run on the CERICT prototype, including two servers connected by an In-

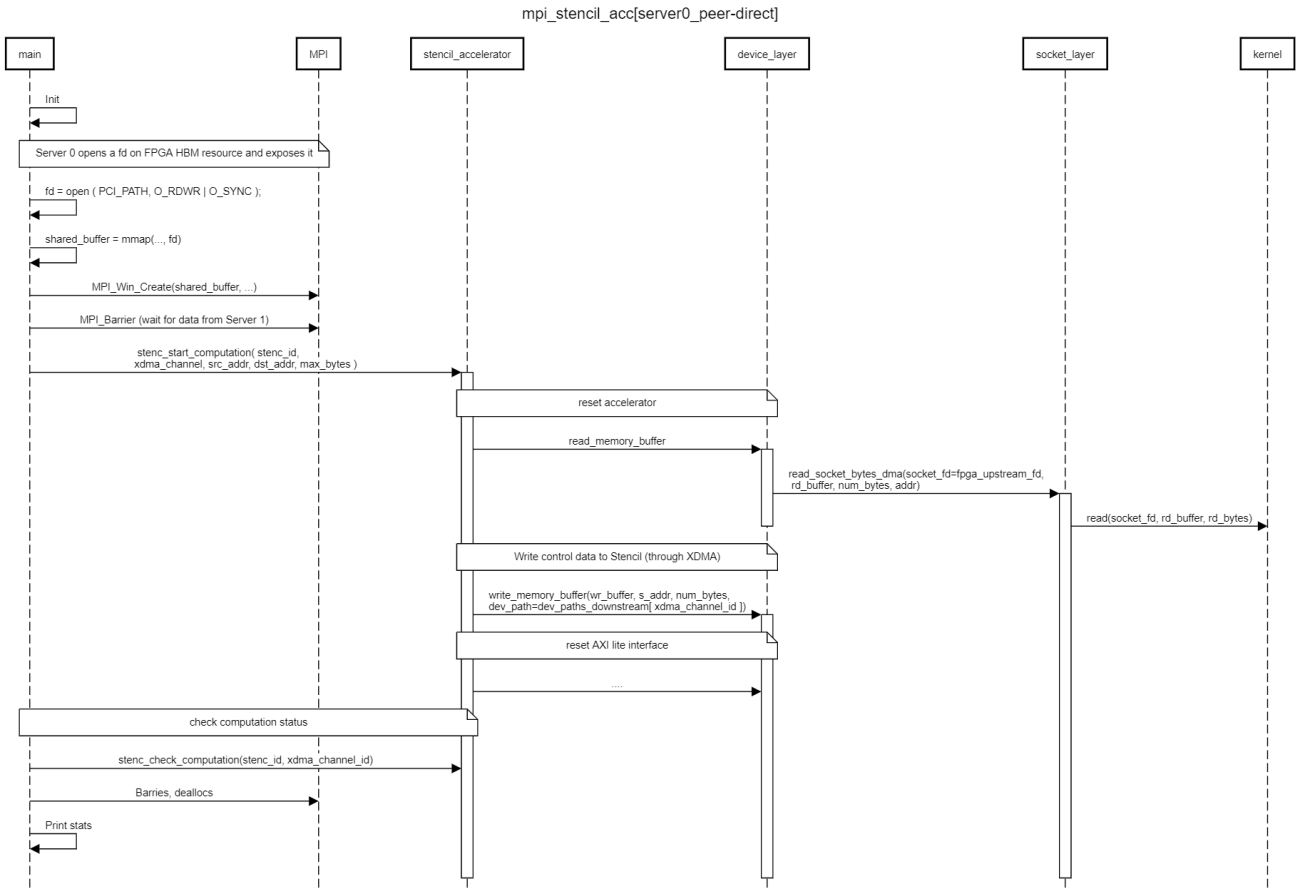



Figure 13: Sequence diagram of the demo stencil application relying on RDMA, highlighting the main MPI primitives used in the code.

finiBand EDR link, where Server 1 is assumed to be the master node distributing the work to acceleration nodes, while Server 0 hosts the FPGA card (namely a Xilinx Alveo U280 card) exposing the HBM memory for PCIe peer-to-peer access by the InfiniBand adapter (see Deliverable 4.3), which enables RDMA for the data transfer. Server 0 of course is assumed to host an MPI process, which is involved in the coordination with the master process on Server 1, but note that application data *never flow* to the host memory of Server 0, as the stencil tiles are directly transferred to the FPGA card memory.

Last, for the purposes of this delverable, the implementation of the stencil code was adapted to the AEM model featured by BarbequeRTRM. Based on the interfaces described earlier in this deliverable, the adaptation of the user code was straightforward. To implement the stencil application following the AEM, we implemented a new class derived from `BbqueEXC`, which has a set of methods used by the AEM runtime to launch, monitor and handle the execution. The actual computation is performed in the `onRun()` method. More precisely, the execution of a single stencil calculation is split in multiple `onRun()` cycles. At the end of every cycle, the application can ask the resource manager if the available resources have changed: if this is the case, the next `onRun()` cycle will be executed on a different resource. Our stencil application is able to execute both on the CPU or on the FPGA, using the stencil sccelerator. The command used to launch the application is:

$$\texttt{aem-stencil --r } R \texttt{ --w } W \texttt{ --m } \quad M$$

where $R$ is the total number of rows of the stencil, $W$ is the size of a single work unit (that is to say, the amount of rows calculated in a single `onRun()` cycle), and $M$ is the minimum work unit allowed (if the last `onRun()` cycle would remain with less rows than $M$, it is avoided and the rest of the stencil is calculated in the current cycle). A separate class, called `stencil_status`, is used to keep track of the status of the computation between different `onRun()` cycles. As highlighted above, we relied on common programming practices for parallelizing the stencil application. In particular, when dividing a stencil computation in multiple cycles, we need to guarantee an appropriate treatment of the halo region and this treatment is dependent on the amount of the iterations of the stencil. If we divide the stencil calculation using multiple tiles, each tile should have a number of rows equal to:

- $W + N$ for the first tile, where the additional $N$ rows are placed *on the top* of the tile,
- $W + N$ for the last tile, where the additional $N$ rows are placed *at the bottom* of the tile,
- $W + 2N$ for the other tiles, where $N$ additional rows are placed *at the bottom* of the tile, and the other $N$ rows are placed *on the top* of the tile,

where $N$ is the number of iterations and $W$, as already highlighted, is the size of the work unit.

Concerning the prototype setup, BarbequeRTRM is installed on CERICT Server 0 at `/opt/bosp` and relies on the *FPGA OpenCL Runtime* to retrieve information about the resources available on the system. The libraries of the *XDMA_Shell* also need to be installed. The application was successfully run and evaluated on the CERICT servers. As explained in the next subsections, we could fundamentally draw the following conclusions from these tests:

- The runtime infrastructure, particularly the support for C/R capabilities, does not intefere with the application operation from a functional point of view and has a limited impact in terms of added time overheads, both in absolute terms and compared to a pure-software

soluton. An extensive characterization is given in the following subsection.

- The Remote Direct accelerator Memory Access capabilities were verified and proved to be properly supported by both the hardware infrastructure and the software stack.
- The measured communication figures are aligned with the expected RDMA performance. A detailed quantitative evaluation for this aspect is given in Deliverable 4.3, where the underlying middleware support for remote direct accelerator memory access is presented, while the resulting communication/execution times for the MPI stencil application are given in Section 4.4.4.

### 4.4.3 Checkpoint Overhead Characterization

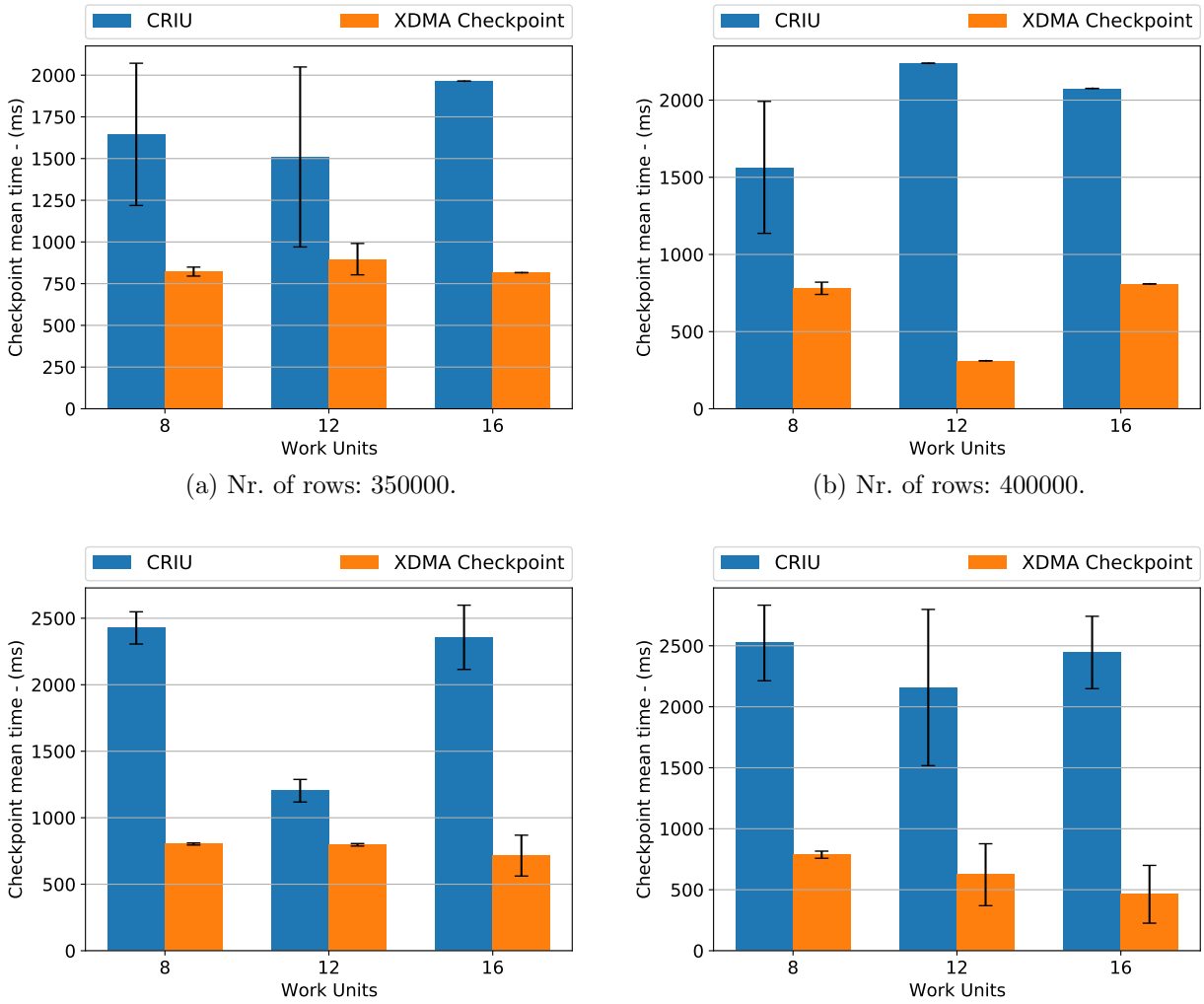
We ran some tests using the stencil application to characterize the overhead incurred by the checkpoint routine, using the services provided by *CRIU* and the *XDMA Checkpoint Library*. The stencil application was implemented according to the *Adaptive Execution Model* and able to execute both on CPU and FPGA. We collected the latencies observed performing periodic checkpoints with both checkpointing mechanisms, setting the period to 10 seconds. We tested the application ranging the number of rows of the stencil between 350000 and 500000, considering, for each workload, three different work units, i.e. 8, 12 and 16 rows of the stencil to be executed in the same `onRun()`.

Figures 14a-14d show the measurements of the checkpoint time mean value, classified by workload. The plots show that, for both technologies, the variation of workload and/or work unit does not influence the checkpoint time. In the majority of the cases, the average checkpoint overhead ranges between approximately 1.5 and 2.5 seconds in the case of *CRIU*, while it fluctuates between 0.7 and 0.9 seconds when the *XDMA Checkpoint* technology is used. Considering that, due to the short duration of the application code execution, we have been able to observe a maximum of five checkpoints per case, we exploited the above mentioned property to achieve a more meaningful result, given by the overall mean checkpoint time, shown in Figure 15.

The measurements highlight that the checkpoint mechanism provided by *CRIU* produces, on average, an overhead in the order of 2 seconds per dump, with a standard deviation of approximately 550 milliseconds, while the *XDMA Checkpoint* presents a mean latency of about 730 milliseconds with a standard deviation equal to 180 milliseconds. In conclusion, the experimental results show that the overhead produced by the latter is 2.8 times smaller than the one produced by the former, guaranteeing a minor performance loss when the accelerator is in charge of the execution.

### 4.4.4 RDMA Performance

This subsection reports on the final numbers evaluated in a full scenario of an MPI stencil application, exploiting tiling and remote FPGA-based stencil accelerators and remote direct accelerator memory access capabilities. Note that the internal structure and local performance of the dedicated accelerator are provided in Deliverable 4.3, while here we refer to a multi-node setting exploiting the full range of mechanisms introduced by RECIPE. Figure 16 shows the breakdown of the communication and execution time for various sizes of the matrix processed by the stencil code. The results directly reflect the findings in Deliverable 4.3. The levels of communication throughput and computation throughput are in the same order of magnitude.




(a) Nr. of rows: 350000.

(b) Nr. of rows: 400000.

(c) Nr. of rows: 450000.

(d) Nr. of rows: 500000.

Figure 14: Mean checkpoint time using CRIU (blue bars) and XDMA Checkpoint Library (orange bars) relying on the checkpointing hardware support deployed in the RECIPE rFPGA custom shell.

The absolute bandwidth values and the asymmetry exhibited by read operations depend purely on the particular physical configuration of the PCIe topology in the node which hosts the FPGA card, as explained in Deliverable 4.3. Overall, for a 4GB tile, the transfers to/from the remote node through RDMA and the FPGA stencil computation require a total time of around 4.57 s.

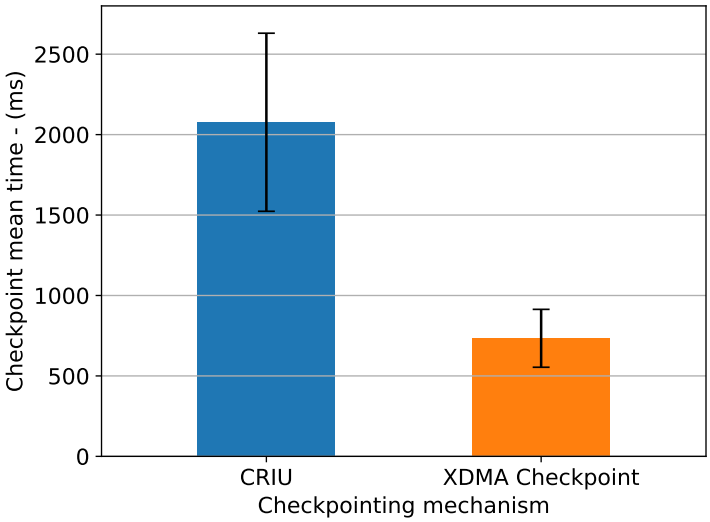



Figure 15: Overall checkpoint mean time.

# 5 Conclusions

In this deliverable, we described the integration effort of the RECIPE software stack. The integration has involved the developments of software components coming from different work packages.

We described how the local resource manager, which occupies a central role in the overall software stack, relies on a set of libraries or operating system interfaces for monitoring the status and perform the assignment of hardware resources. Such interfaces are in part already available and in part developed for the RECIPE targets (*FPGA Monitoring Library*, *HN Library*). For the reliability and the timing analysis part, we developed the *Hardware Relibility* and the *Timing Analysis* libraries and integrated them in the BarbequeRTRM management flow. Furthermore, a remarkable achievement of the last year of activity is represented by the support of the Checkpoint/Restore mechanisms on a heterogeneous configuration based on a computational node by a high-end multi-core CPU and FPGA hosting a custom accelerator. We provided some experimental results for this, showing how the checkpoint operation of FPGA-hosted accelerations is characterized by a lower overhead with respect to processes running on CPU.

We have also described the current status and recent advances on the Global Resource Manager, and the changes with respect to the setup described in previous deliverables (and more specifically D2.3) and how applications can be launched also using the underlying HELENNA infrastructure (which is also described to a greater detail as part of D1.8). The assessment of overall overheads when using the GRM for the three Use Cases of RECIPE, and more specifically for UC3, will be provided in D1.9.

Finally, on the programming model side, we shown in previous deliverables how we developed custom programming models targeting heterogeneous platform (the *MANGO Programming Library*) and enabling the possibility of adapting the application to the set of resources assigned at run-time and dynamically changed by the local resource manager policy (the *Adaptive Exe-*

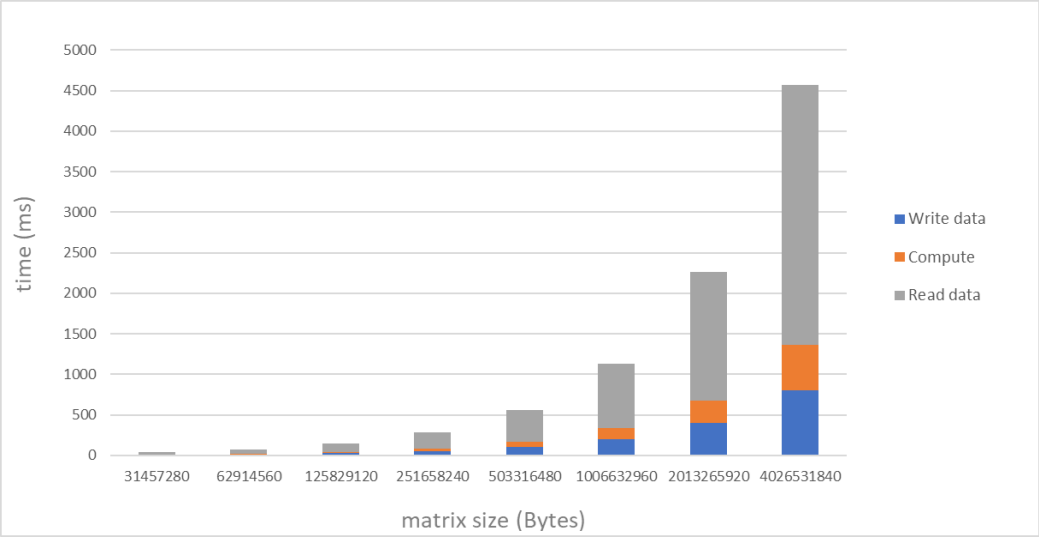



Figure 16: Breakdown of the communication and execution time for various sizes of the matrix processed by the stencil code.

*cution Model*). While the MANGO Programming Library has been refactored to improved the integration of multiple target platforms, the Adaptive Execution Model has been enriched by the introduction of different language wrapper layers. In D2.4 we described the Python support, while in the current one we introduced the Fortran wrapper. Fortran is in fact one of the languages characterizing the implementation of the UC2, other than being a language still in use in scientific applications running on HPC infrastructures. Finally, in D2.2 we have shown the addition of the support to legacy applications (no porting required), such that the local resource manage could control applications not exploiting any of the proposed programming models, even if this implies some limitations in terms of control capabilities.

As already stated, the actual exploitation of the effort described in this deliverable will be represented by the resource allocation policies described in the next D3.7.

# References


[1] CRIU. CRIU: Checkpoint/Restore in Userspace. https://criu.org.

[2] Dmtcp-Crac. Dmtcp-crac/crac-early-development.

[3] R. Garg, A. Mohan, M. Sullivan, and G. Cooperman. Crum: Checkpoint-restart support for cuda's unified memory. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 302–313, 2018.

[4] Twinkle Jain and Gene Cooperman. Crac: Checkpoint-restart architecture for cuda with streams and uvm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[5] Khronos OpenCL Working Group. The OpenCL Specification, Version 2.0. https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf, October 2014. Lee Howes and Aaftab Munshi eds.

[6] Paul Menage. Linux cgroups v1. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[7] Patrick Mochel and Mike Murphy. sysfs - The filesystem for exporting kernel objects. https://www.kernel.org/doc/html/latest/filesystems/sysfs.html.

[8] NVIDIA. NVIDIA Management Library (NVML). https://developer.nvidia.com/nvidia-management-library-nvml.

[9] Srinivas Pandruvada. Running Average Power Limit - RAPL. https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl.

[10] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. chronovise: Measurement-based probabilistic timing analysis framework. *Journal of Open Source Software*, 3(28):711, 2018.

[11] Taichiro Suzuki, Akira Nukada, and Satoshi Matsuoka. Transparent checkpoint and restart technology for cuda applications. In *GPU Technology Conference (GTC)*, volume 20156, 2016.