PHANTOM

Cross-Layer and Multi-Objective Programming Approach for Next Generation Heterogeneous Parallel Computing Systems

Project Number 688146

D4.4 – Final release of integrated monitoring platform, infrastructure integration and resource management software stack

Version 1.1
21 December 2018
Final

Public Distribution

University of Stuttgart, WINGS ICT Solutions, University of York, Unparallel Innovation, The Open Group

Project Partners: **Easy Global Market, GMV, Intecs, The Open Group, University of Stuttgart, University of York, Unparallel Innovation, WINGS ICT Solutions**

PROJECT PARTNER CONTACT INFORMATION

Easy Global Market Philippe Cousin 2000 Route des Lucioles Les Algorithmes Batiment A 06901 Sophia Antipolis France Tel : +33 6804 79513 E-mail : philippe.cousin@eglobalmark.com	**GMV** José Neves Av. D. João II, Nº 43 Torre Fernão de Magalhães, 7º 1998 – 025 Lisbon Portugal Tel. +351 21 382 93 66 E-mail: jose.neves@gmv.com
Intecs Silvia Mazzini Via Umberto Forti 5 Loc. Montacchiello 56121 Pisa Italy Tel: +39 050 9657 513 E-mail: silvia.mazzini@intecs.it	**The Open Group** Scott Hansen Rond Point Schuman 6 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of Stuttgart Bastian Koller Nobelstrasse 19 70569 Stuttgart Germany Tel: +49 711 68565891 E-mail: koller@hlrs.de	**University of York** Neil Audsley Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325571 E-mail: neil.audsley@cs.york.ac.uk
Unparallel Innovation Bruno Almeida Rua das Lendas Algarvias, Lote 123 8500-794 Portimão Portugal Tel: +351 282 485052 E-mail: bruno.almeida@unparallel.pt	**WINGS ICT Solutions** Panagiotis Vlacheas 336 Syggrou Avenue 17673 Athens Greece Tel: +30 211 012 5223 E-mail: panvlah@wings-ict-solutions.eu

DOCUMENT CONTROL

Version	Status	Date
0.1	Internal draft	27/09/2018
0.2	Partners' contributions	15/10/2018
0.3	Revised version by contributors	20/10/2018
0.4	Initial version for internal reviews	25/10/2018
0.8	Further updates and contributions from partners	04/12/2018
1.0	Full version	06/12/2018
1.1	Additional updates from integration	21/12/2018

TABLE OF CONTENTS

EXECUTIVE SUMMARY

This document describes the final release of the PHANTOM middleware which sets up, operates, and optimizes the distributed heterogeneous infrastructure (consisting of diverse CPU, GPU, FPGA, and mixed hardware). The WP3 middleware primarily supports the higher-level PHANTOM tools (such as the Multi-Objective Mapper or the Modelling Framework) but it also provides end-user interfaces for the management of the infrastructure.

The PHANTOM middleware includes the tools that were introduced in the M18 extended release (D4.3) – the Monitoring Client, the Resource Manager, the FPGA Linux Infrastructure, and the IP-Cores Marketplace. In addition, the final release includes a newly-developed tool – the Deployment Manager, which is introduced in this deliverable for the first time.

The final PHANTOM middleware is thus composed of the following tools:

- **Monitoring Client –** a light-weight software service that collects utilization data (including the application performance and energy consumption) of the heterogeneous infrastructure and makes it available to the PHANTOM Monitoring Framework (cf. D1.4).

  The final version of the PHANTOM Monitoring Client now supports accelerators (GPU and FPGA devices) and provides an easy way to obtain all data on an "as a service" basis, thus fostering the integration.

- **Resource Manager –** a light-weight software service that allows the end-users to easily set up a heterogeneous infrastructure for their applications. This service allows all other components of the PHANTOM platform to obtain the status of each included resource (either CPU-, GPU, or FPGA-based), based on the live-monitoring data from the Monitoring Client.

  The final version of the PHANTOM Resource Manager includes a registration front-end for adding new devices or managing the existing ones with several configuration options, as well as an analysis back-end, both functioning in a tight integration with the Monitoring Framework.

- **Deployment Manager –** responsible for the final adaptation of a PHANTOM application component to the selected hardware resource (e.g. generic or low-power CPU, GPU, FPGA, etc.). The Deployment Manager modifies user code as well as generates new source files to implement the integration of the application. Additionally, the corresponding scripts for the compilation and deployment of the application are generated, all available at the development machine for the user to execute when feasible. After the adaption, the Deployment Manager can also trigger execution of the component.

  The final version of the PHANTOM Deployment Manager provides an automatic framework, abstract enough to hide the complexity of a heterogeneous infrastructure, for the final integration and deployment of the application on the hardware platform, requiring minimal user interaction, but also allowing the users to execute their own configurations on the deployment if need be.

- **IP-Cores Marketplace and IP-Cores Generator –** a collection of the application-specific acceleration blocks to be executed on FPGA devices (the Marketplace) along with the tool to automatically generate them (the Generator). These blocks can be seamlessly integrated into the components of a PHANTOM application and boost the performance of the most computation-intensive parts of their workloads. The collection includes IP-cores for some common (compute-intensive) algorithms such as Fast Fourier transform (FFT) and Finite impulse response (FIR), or image processing filters such as Sobel Filter, Discrete Wavelet Transform (DWT), etc.

  The final version of the PHANTOM IP-Cores Marketplace and Generator include a software component, that undertakes the position of a bridge or driver between software and FPGA hardware, to provide to the users a straightforward approach, to easily integrate the IP core accelerated components with their application.

- **FPGA Linux Distribution and Infrastructure –** enables the automatic deployment of PHANTOM application components on to FPGA platforms. The infrastructure automatically constructs FPGA designs from IP cores, and autogenerates the associated software APIs and Linux device drivers to make those cores available to user software.

  The final version of the PHANTOM FPGA Linux Distribution and Infrastructure supports a range of FPGA development boards and target architectures, and includes support for security-hardened components that can execute in a fully-partitioned environment to protect their execution and data integrity.

The content of the deliverable is organized as follows. Section 1 gives a brief overview of the role and placement of the WP4 middleware in the PHANTOM architecture. Sections 2 – 6 give necessary details on each of the middleware, including the architecture design, operation, and implementation details. Section 7 provides conclusions and discusses major findings. The requirements and status of their achievement are listed in Appendixes.

1. INTRODUCTION

1.1 MOTIVATION

The goal of WP4's R&D activities is to provide solutions for operating a heterogeneous infrastructure that is composed of diverse CPU-, GPU-, and FPGA-resources/devices. The typical PHANTOM infrastructure can not only be set up ahead of time (Figure 1a) but also dynamically, allowing users to add and remove devices spontaneously (Figure 1b).

Figure 1: Organization of hardware resources as "heterogeneous cloud": a) HPC system with large server solutions and accelerators, b) low-power testbed with small and portable systems.

The heterogeneity of the included devices and especially the availability of accelerators introduces a big challenge for established resource management solutions like PBS[1]/Torque (for HPC) or OpenStack (for Cloud), which operate at the node-level only and do not take into account the rich granularity of the managed resource (e.g. the numerous CPU and GPU cores or the availability of FPGA accelerators) when executing the application in a "batch" (or "job-based") way. This leads to inefficient infrastructure usage by the less optimized applications. The limited application performance and infrastructure utilization is also a case for the above-mentioned (Figure 1a) HLRS cluster, managed by Torque.

The DreamCloud project [1] provided a technology to overcome this limitation by allowing several jobs to share common resources (only CPU-based), thus ensuring a better utilization. The resource heterogeneity breaks a new ground of optimization technology, narrowing the optimisation scope down to GPU kernels and FPGAs, as targeted by PHANTOM. The challenge of this PHANTOM approach raises a wide set of issues related to heterogeneous resource management, enabling of accelerators, and enforcement of security policies imposed by users and applications.

[1] PBS = Portable Batch System

1.2 SCOPE OF ADDRESSED TOOLS

The WP4 middleware (Figure 2) constitutes the hardware-specific part of the PHANTOM software architecture (cf. D1.4) and supports the high-level management components (such as the MOM, MBT, etc.) as well as the users (by providing intuitive management interfaces for the infrastructure).

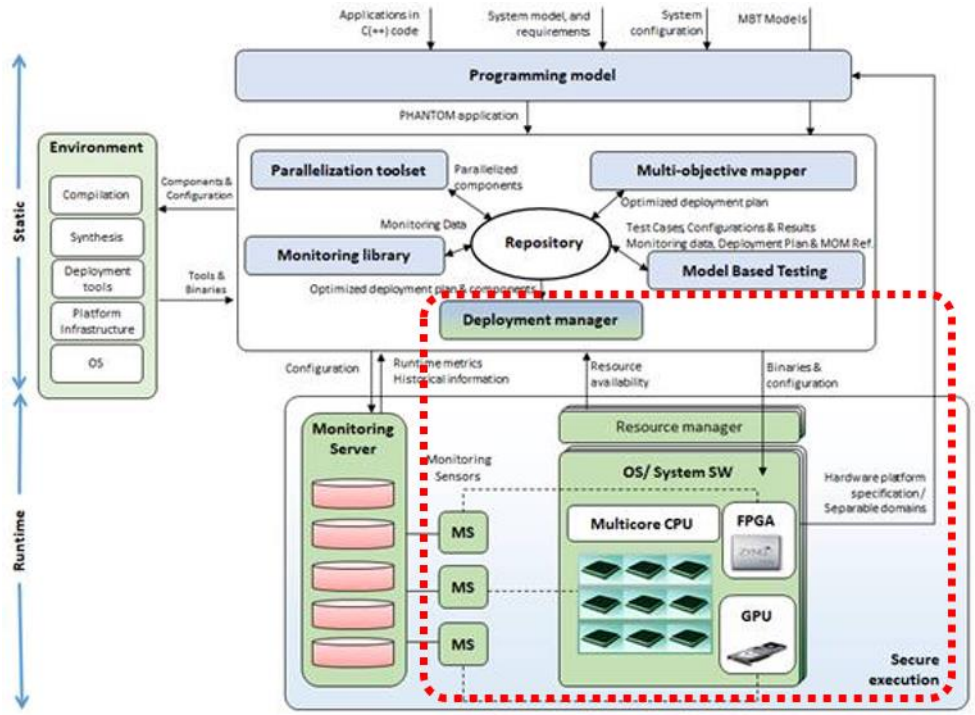


Figure 2: Scope of the released WP4 tools.

Management of a highly dynamic, heterogeneous infrastructure is a complex task which requires monitoring and status tracking (for both infrastructure hardware and applications software) as well as end-user tools to add/remove new devices to the infrastructure. For these actions the PHANTOM middleware offers a client application of the Monitoring Framework – the **Monitoring Client** – which is a light-weight software service that collects utilization data (including the application performance and energy consumption) of the heterogeneous infrastructure and make it available to the PHANTOM Monitoring Framework (cf. D1.4).

In order to ensure the proper collaborative operation of all heterogeneous devices that are included into the common infrastructure, a dedicated tool – the **Resource Manager** – has been developed. It allows the end-users to easily set up a heterogeneous infrastructure for their applications, by means of a rich set of configurable scripts. Also, this service allows all other components of the PHANTOM platform to obtain the status of every available resource (either CPU-, GPU, or FPGA-based), based on the live-monitoring data from the PHANTOM Monitoring Client (see above).

The execution of applications is accomplished by the **Deployment Manager –** which provides the final transformation of the application's code to the target resource and

performs all preparatory steps (like input data provisioning) that are required by the applications.

Execution on the FPGA accelerators is carried out by means of IP-Cores – hardware realisations of the application algorithms, the most typical of which have been implemented within the **IP-Cores Marketplace**, such as for Fast Fourier transform (FFT) and Finite impulse response (FIR), or image processing filters such as Sobel Filter, Discrete Wavelet Transform (DWT), etc.

The execution of IP-Cores on FPGA devices is facilitated by the **FPGA Linux Distribution and Infrastructure** which interfaces FPGAs in the standard Linux environment of the hosting machine.

1.3 MAJOR INNOVATIONS

This section highlights the major innovations implemented by the WP4 middleware.

Monitoring Client

The monitoring solutions that are currently available on the market like Torque, SLURM, etc. are very heavyweight and so only applicable to large and powerful hardware that is able to neglect the relatively high resource utilization overhead. Similarly, these systems often only support homogeneous hardware (like Zabbix and the majority of the other known tools).

The PHANTOM Monitoring Client, on the contrary, relies on the integrated information coming from the already available sources of performance data (like hardware counters, external energy measurement units, etc.) and thus provides minimal resource utilization overhead whilst maximizing the value of the obtained information. The data are passed to a consolidated storage (provided by the Monitoring Server, cf. D1.4) so that the users along with the high-level PHANTOM management components obtain recent utilization and health statistics for all online infrastructure devices. The Monitoring Client supports different types of resources, regardless of whether these are pure CPU-based devices, or GPU- or FPGA-enabled accelerators, and provides the functionality to control the data that are obtained from the monitored devices. The Monitoring Client is a unique solution on the market and has no analogues in the spectrum of its functionality.

More precisely, the main innovations are:

1. Platform-independence: seamless integration with CPU, GPU, and FPGA devices.

2. Exposing the collected monitoring data via standardized (RESTful) interfaces.

3. High measurement accuracy (in the range of milliseconds for MF Client, and typically in range of tens of microseconds for Apps instrumented with the MF Library).

4. Plug-able architecture supporting easy extension to new hardware architectures.

5. High customization of the monitored metrics.

Resource Manager

Originally, it was planned to build the Resource Manager based on existing solutions such as OpenStack (for virtualized Cloud environments) or SLURM (for the homogeneous cluster or HPC based infrastructures). However, it turned out the former (OpenStack) is of a little use for bare-metal resources (such as embedded low-power and reconfigurable FPGA-based devices) and the latter (SLURM) is more tailored to homogeneous systems with a static configuration of resources. Moreover, the monitoring functionality is already provided by the Monitoring Client and it is not necessary introduce an additional monitoring layer with a considerable overhead.

The PHANTOM Resource Manager allows the end-users to easily set up a heterogeneous infrastructure for their applications, adding new devices or managing the existing ones with several configuration options, by means of a rich set of configurable scripts or with a web-interface.

This service also allows all other components of the PHANTOM platform to obtain the status of each included resource (either CPU-, GPU, or FPGA-based), based on the live-monitoring data from the Monitoring Client.

The list of innovations includes:

1. Platform-independence: seamless integration with CPU, GPU, and FPGA devices.

2. Set up and access to the devices' configuration via standardized (RESTful) interfaces.

3. Report of the status of the available resources of the devices by RESTful interfaces, as well as an automated notification based on Web-Sockets.

4. Simplifies the configuration of the Monitoring Client and instrumented applications, by providing a default monitoring configuration independently for each device.

Deployment Manager

The main functionality of the Deployment Manager is to integrate the efforts of the preceding PHANTOM modules and activate the low-level hardware infrastructure according to the deployment plan and the final component source code. In particular, the orchestration of the data transfers between the application components as well as the execution of these components independently is the main purpose of the tool and is achieved with the generation of the necessary source files and of the corresponding deployment scripts.

The generated files are configured according to the deployment plan and provide the necessary generality to enable the execution of the deployment plan on any platform that is requested. The generated scripts interact with the Repository and the PHANTOM Managers in order to move the corresponding files and binaries to the corresponding locations across the target platform and the final deployment is configured in the deployment script to initiate the execution of the application.

The main innovations of the tool can be described in the following points:

1. Integrates the application components with the highly abstract Programming Interface functions through the usage of the corresponding structures that will implement the requested data transfers between the components.
2. Generates the necessary code that will enable statically-defined data transfers as well as dynamically-defined ones.
3. Implements the deployment of the application in a fine-grained fashion by providing wide support of different deployment requirements, such as required amount of threads/processes, specific software-hardware mappings/bindings etc.
4. The final deployment is guided by low-level requirements to remove unnecessary overheads, while using state-of-the-art libraries like OpenMPI and the POSIX standards to increase efficiency.

IP-Cores Marketplace and Generator

The IP Core Marketplace and IP Core Generator provides a way for users to exploit the benefits of FPGA hardware without the need for any extra effort or knowledge about reconfigurable hardware. The Marketplace provides optimized IP cores of common mathematical algorithms that can be reused in many different applications, while the IP Core Generator allows the user to run part of their specific application in FPGA hardware. The main innovations include:

1. Collection of curated IP cores for common mathematical algorithms optimized by hardware engineers that are easy to integrate in any application.
2. Ability to create IP cores and the software-hardware adapter from normal C/C++ code without developer intervention.

3. Allows a software developer to exploit hardware resources without need for reconfigurable hardware knowledge.

4. Simple and straightforward integration with the original application.

FPGA Linux Distribution and Infrastructure

The role of the PHANTOM Linux distribution is to implement a range of support features of the PHANTOM platform on embedded devices, with a particular focus on FPGA platforms. To enable rapid development and deployment, it is almost entirely automatically-generated without direct user interaction, based on instruction from the rest of the toolchain.

The main innovations are:

1. Support for automatic creation of FPGA hardware designs which encapsulate the PHANTOM IP cores developed as part of the project. When the Multi-Objective Mapper assigns a set of IP cores to a given FPGA platform, the generation tools assemble the design from the specified cores.

2. Automatic integration of communications APIs to allow the FPGA to be used as part of a heterogeneous compute platform. Communications between software and hardware are also handled seamlessly to support the PHANTOM programming model. The APIs are automatically updated based on the IP cores built into the system.

3. Automatic integration of security features when required to ensure execution integrity. Unlike a GPU system, when an application requires secure execution, memory can be fully partitioned and protected, from both hardware and software.

2. MONITORING CLIENT

2.1 OVERVIEW

The architecture of the Monitoring Framework (MF) is composed of the Monitoring Library, the MF-Server, and the MF-Client, interconnected as shown in **Figure 3**.

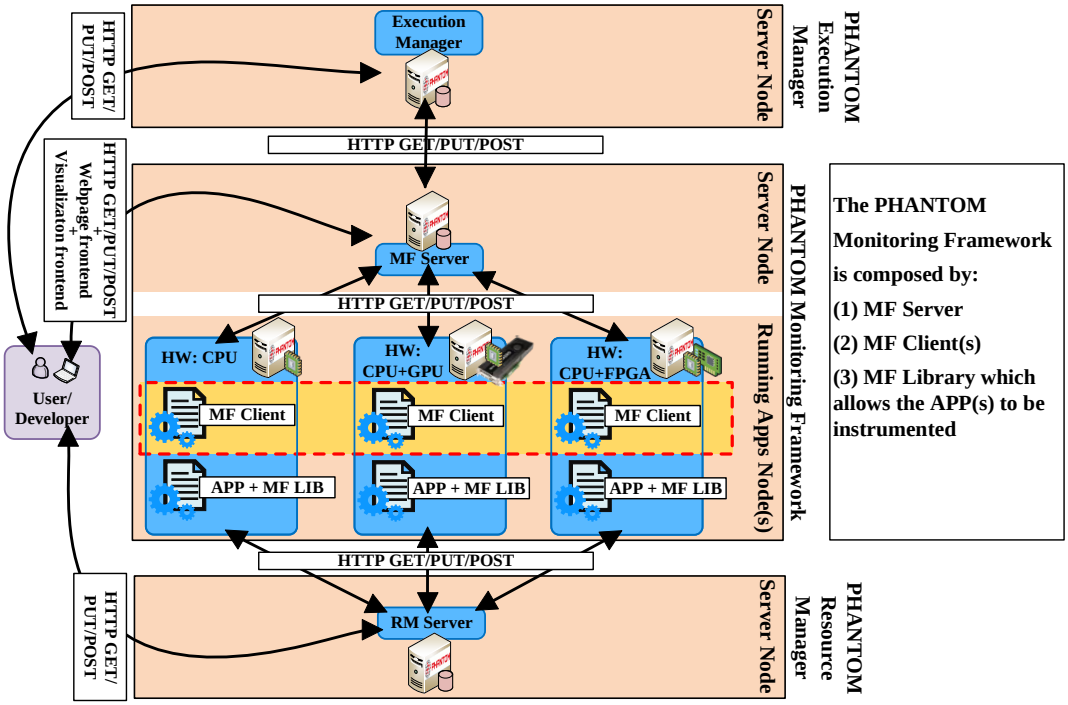


Figure 3. PHANTOM Monitoring Framework Architecture. In the figure is highlighted MF Client.

The MF-Client is a lightweight service that is installed on top of the available system software stack (it runs as a separate system process in the background of the OS). The MF-Client task is to collect the metrics on the system level, and forward them to the MF-Server.

2.2 DESIGN SPECIFICATIONS AND INTERFACES

The PHANTOM toolset requires information about the resource utilization and system load by the previously executed PHANTOM applications (application-level), and information about the state of the available resources (infrastructure-level) for the next execution of PHANTOM applications.

As clarified in D1.2, the PHANTOM Monitoring Framework architecture follows the design of ATOM – the monitoring solution elaborated by the EU EXCESS and DreamCloud projects. However, the initial design of ATOM was too infrastructure-oriented and application-level monitoring was not supported. Therefore, ATOM has

been considerably redesigned to enabling application-level monitoring, support heterogeneous hardware resources (including hardware accelerators as GPUs and FPGAs), and use a more modular component-based and service-oriented architecture, for enabling component-based organization of the PHANTOM applications. Usability and configurability on the user-side is also improved.

In order to achieve the requirements on monitoring of metrics, in both at System and Application Levels, the monitoring workflow is divided into two parts. As shown in Figure 4, devices can keep being monitored by the MF-Client (Infrastructure-level), and users can instrument their code and monitor at the application level using the MF Library (Application-level). In this deliverable is described the collection of metrics on the Infrastructure level. The details about the instrumentation of applications are described in D3.2.

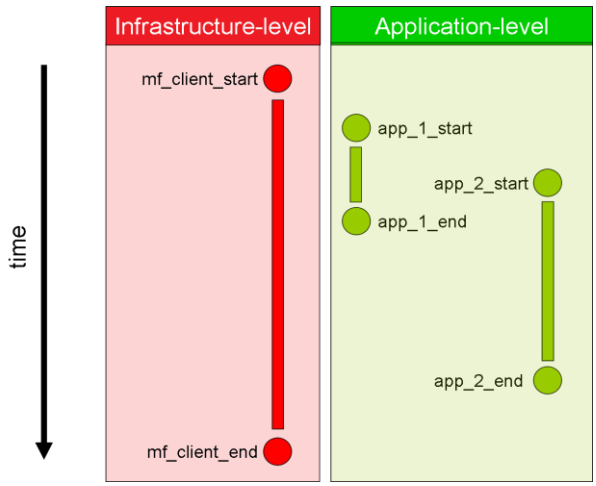


Figure 4: Infrastructure- and application-level monitoring with the MF-Client

The MF-client collects the infrastructure-specific metrics for the whole node/board, regardless of the applications running on it. These metrics are always collected after every time interval. A list of metrics to be acquired from the whole available set, and the frequency at which they should be uploaded to the MF-Server and to the Resource Manager, is provided by the configuration parameters (referred as Monitoring Configuration) stored by the Resource Manager (RM). The users do not need to do additional actions.

Additionally, the Monitoring Configuration has to be able to be updated during run-time. Thus, the MF-Client loads periodically the Monitoring Configuration, (the load frequency is also a parameter in such configuration). The modification of configuration at the Resource Manager is a simple task described in later in this deliverable in section 3.

The MF-Client design provides a flexible implementation of clients by importing plugins. Only required plugins are loaded, allowing for the minimization of resources used on the monitored device. Each one of these plugins is an independent thread that wakes up on its monitoring time interval. Figure 5 shows different configurations of MF-Client for different hardware platforms, and the transferred information between the

different PHANTOM servers. Notice that the MF-Server keeps all the monitored metrics received, while the RM only retains the most recent value of each one of the metrics.

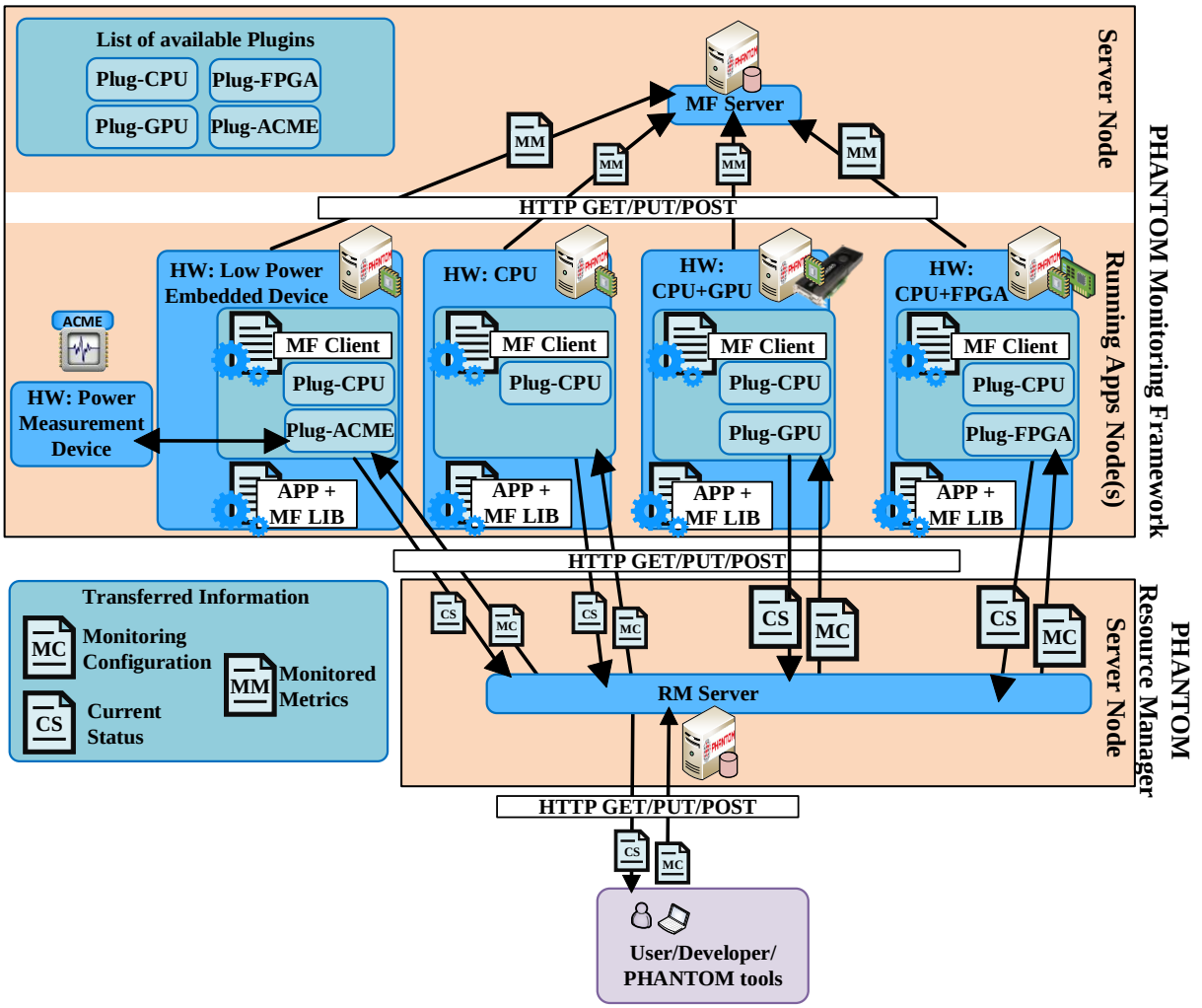


Figure 5: Monitoring Client architecture

2.3 IMPLEMENTATION AND INSTALLATION DETAILS

The MF-Client collects metrics at the system level by importing and starting the reauired set of Monitoring Plugins in the device to be monitored. All the available plugins are implemented in C to be run as independent threads. In order to decrease system utilization, the plugin threads keep sleeping most of the time, and periodically wake up on its monitoring time interval for collecting metrics and storing them in an

independent local buffer. There is a different time interval for transferring the content of each buffer to the MF-Server/RM, which is done in JSON format.

As an example, Listing 1 shows a JSON structure sent by a client to the server at a specific time point, which is composed of a simple key-value **pair metric** (metric name and its numeric value). The time stamp for each event is automatically generated. Following the characterization of metrics data used by time-series database, a metric is thus represented by its name and a series of numerical values collected over time. In the case of the PHANTOM distributed infrastructure, the timestamps sent are revised to the local timestamps (in milliseconds), since users are more interested in time duration rather than the exact real time[2].

Listing 1: Example a typical JSON metrics

```
{
    “local_timestamp”: 1490358666276.5
    “hostname”: “CPU:node01”
    “type”: “performance”
    “core01:MFLIPS”: 415279.555
}
```

The metrics that can be acquired by the Client cover a wide range of functions that target different aspects of the hardware, such as memory, IO, processor, GPU, and network utilization as well as energy consumption (requirements U72, U74-78). The detailed specification of all metrics is provided in Appendix 3. “List of Monitoring Metrics”.

Figure 6 shows also the application-specific metrics, which are related to a particular application execution. Further details on collecting application-specific metrics are described in D3.2.

[2] **CLOCK_REALTIME**: represents the machine's best-guess as to the current wall-clock, time-of-day time. It can jump forwards and backwards as the system time-of-day clock is changed, including by NTP.
CLOCK_MONOTONIC (referred as local-timestamp): represents the absolute elapsed wall-clock time since some arbitrary, fixed point in the past. It isn't affected by changes in the system time-of-day clock. It is the best option to compute the elapsed time between two events observed on the one machine without an intervening reboot.

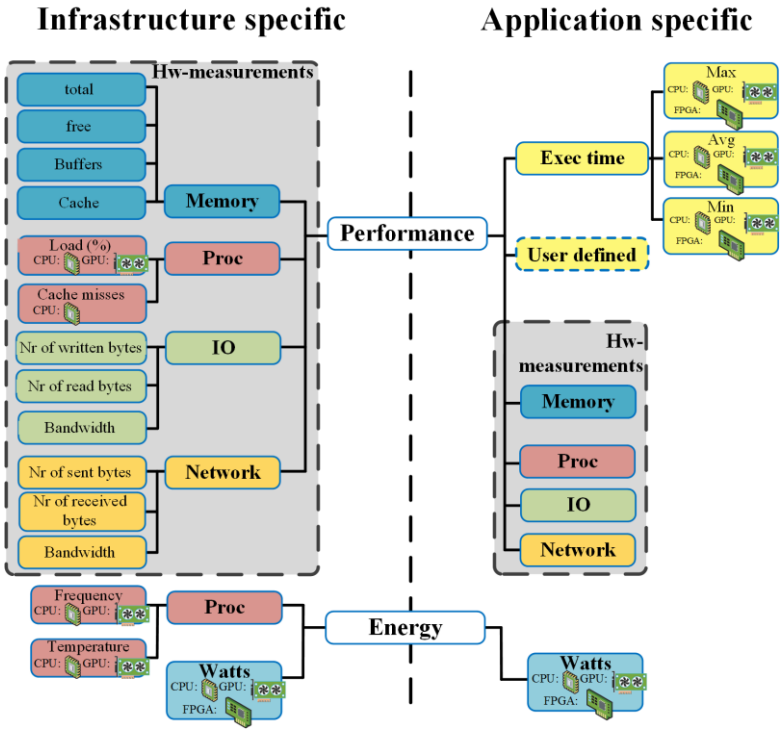


Figure 6: PHANTOM Monitoring Client metrics' taxonomy

Collection of the data from each of the above-listed sources is performed by a special plug-in, which can be built into a modular architecture of the Client software. Plug-ins for PAPI, RAPL, Linux-OS and other sources of information are provided with the current redistribution of the Client (see Appendix 2 "Monitoring-Client dependencies").

The MF-Client relies uses the following sources of information:

- Linux monitoring sensors

  The Linux system software provides some monitoring functionality, such as temperature, voltage, fans status, etc. which can be retrieved by means of special utilities like *lm_sensors*. The Client can interface those tools and consolidate their output.

- Linux OS counters

  The Linux core can track the resource consumption by individual applications, such as the load of each CPU core or memory utilization. This information is retrieved by the Monitoring Client to obtain application-specific characteristics.

- CPU Hardware counters

  Modern CPUs provide special hardware counters (like PAPI, RAPL, etc.) that are registering some important system metrics such as the total instructions executed, the amount of instructions per cycle, etc. This information can be retrieved by dedicated monitoring plug-ins.

Generally, the accuracy of the hardware counters is higher than of the software counters as they are not impacted by latencies introduced by the system software stack.

- External power measurement devices

  Enabled seamless support of external measurement systems along with counter-based hardware capabilities.

  Power consumption is obtained either from dedicated hardware counters (as provided by some server CPUs like Intel Haswell) or from an external power measurement system, such as ACME[3] (Figure 7).

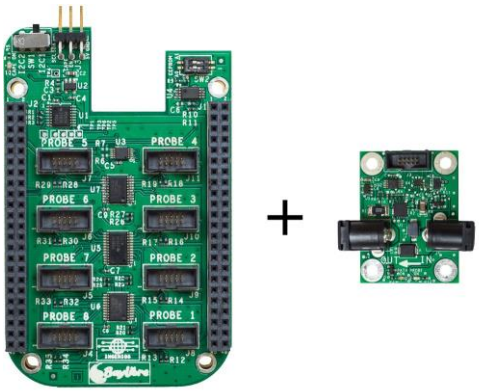


Figure 7: External power measurement board from ACME

- NVIDIA Management Library (NVML)

  NVML is a C-based API for monitoring and managing various states of the NVIDIA GPU devices. This information is retrieved by the Monitoring Client to obtain metrics such GPU utilization, power consumption, memory utilization, and temperature [3].

- Monitor Power Consumption on ZC702/706 Zynq-FPGA board.

  The Zynq (ZC702-ZC706) boards have power regulators [4] and a PMBus compliant system controller to supply core and auxiliary voltages. The MF-Client collects the values of Voltage and Current from the monitoring power controllers (UCD9248PFC) which are controlling those power regulators.

[3] The typical use of the ACME board provides 16-bit 7Ksamples/s with an accuracy of 2.5uV and 0.03uW, with the limitation of 36V and 6A on the power supply on the embedded device.
[4] The voltage output of the regulators (5 switching +1 linear regulators PTD08D210W) provide the power required for the Zynq APSoC as well as the on-board components present on the ZC702/706 board.

For the **installation**, the Client is supplied with the necessary configuration and installation script, so that the installation procedure is straightforward for the users (Listing 2). During the installation, the user can choose a unique name for the monitored hardware platform that will be used for its identification in the scope of all infrastructure devices, which is organized by the Resource Manager (see Section 3).

Listing 2: Monitoring Client setup

```
$ svn export https://github.com/PHANTOM-Platform/Monitoring.git/trunk/Monitoring_client Monitoring_client
$ cd  phantom_monitoring_client

# Executing the next script if your device has a 32 bits Operating System (Intel or ARM CPU):
$./setup-client-32.sh

# Executing the next script if your device has a 64 bits Operating System (Intel or ARM CPU):
$./setup-client-64.sh

$ make clean-all
$ make all
$ make install
```

For each newly-installed device, it is recommended to perform a configuration by means of the PHANTOM Resource Manager (see Section 3). The Resource Manager contains settings for each specific type of supported hardware resources (e.g. x86 CPU, ARM-based CPUs, etc.), which are used to tailor the algorithms of monitoring thus improving its quality.

3. RESOURCE MANAGER

3.1 OVERVIEW

The Resource Manager (RM) is a lightweight service which provides information related to the available heterogeneous infrastructure. It provides for the PHANTOM tools and users a list of the available devices and resources, their most recent load status (CPU, MEM, IO, Net,), their access route (IP address), and in particular the monitoring parameters (Monitoring Configuration) for the MF-Client.

In order to achieve this task, the RM requires an initial configuration (for each registered device), and a running MF-Client on each of the available computing nodes.

3.2 DESIGN SPECIFICATIONS AND INTERFACES

The PHANTOM toolset requires information about the state of the available resources (infrastructure-level) for the next execution of PHANTOM applications. In order to achieve this goal, the Resource Manager (RM) has to store the monitoring configuration (MC) and supply it to the MF-Clients when they request it. The MC specifies what to measure on each device, and how often to measure it.

The RM must also keep a register of the current status (CS) which consists of the most recent value of each metric and their timestamps. It provides a web service where users and the PHANTOM tools can retrieve these values in JSON format. Figure 8 shows the transferred information between the Resource Manager, users, and the PHANTOM tools.

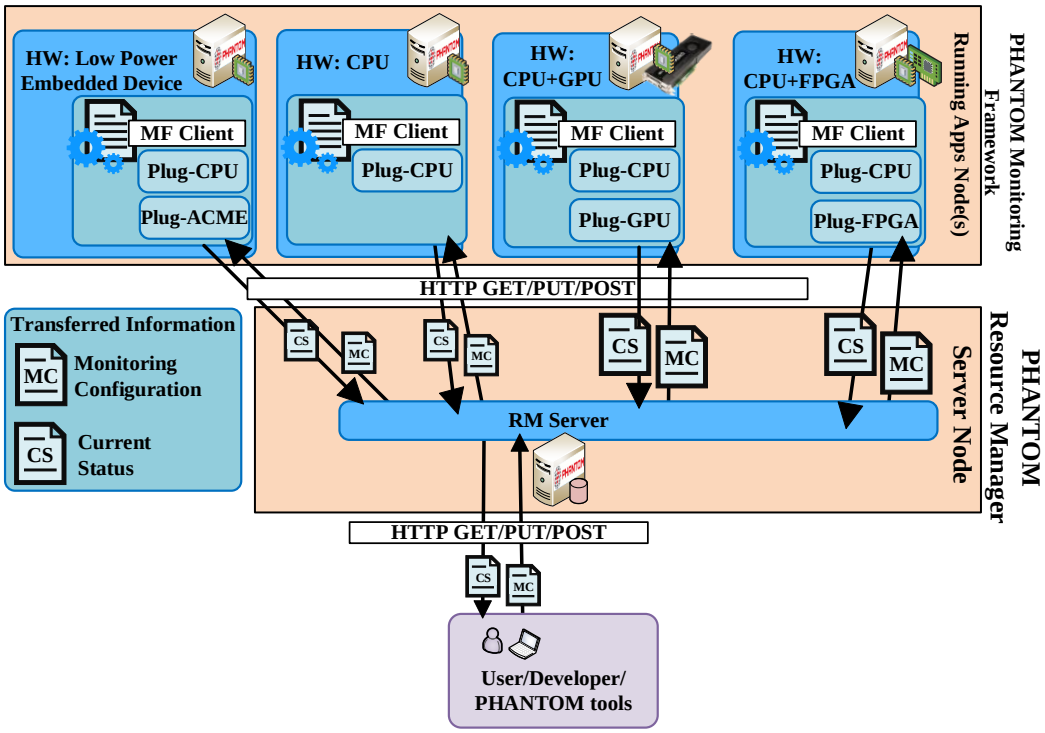


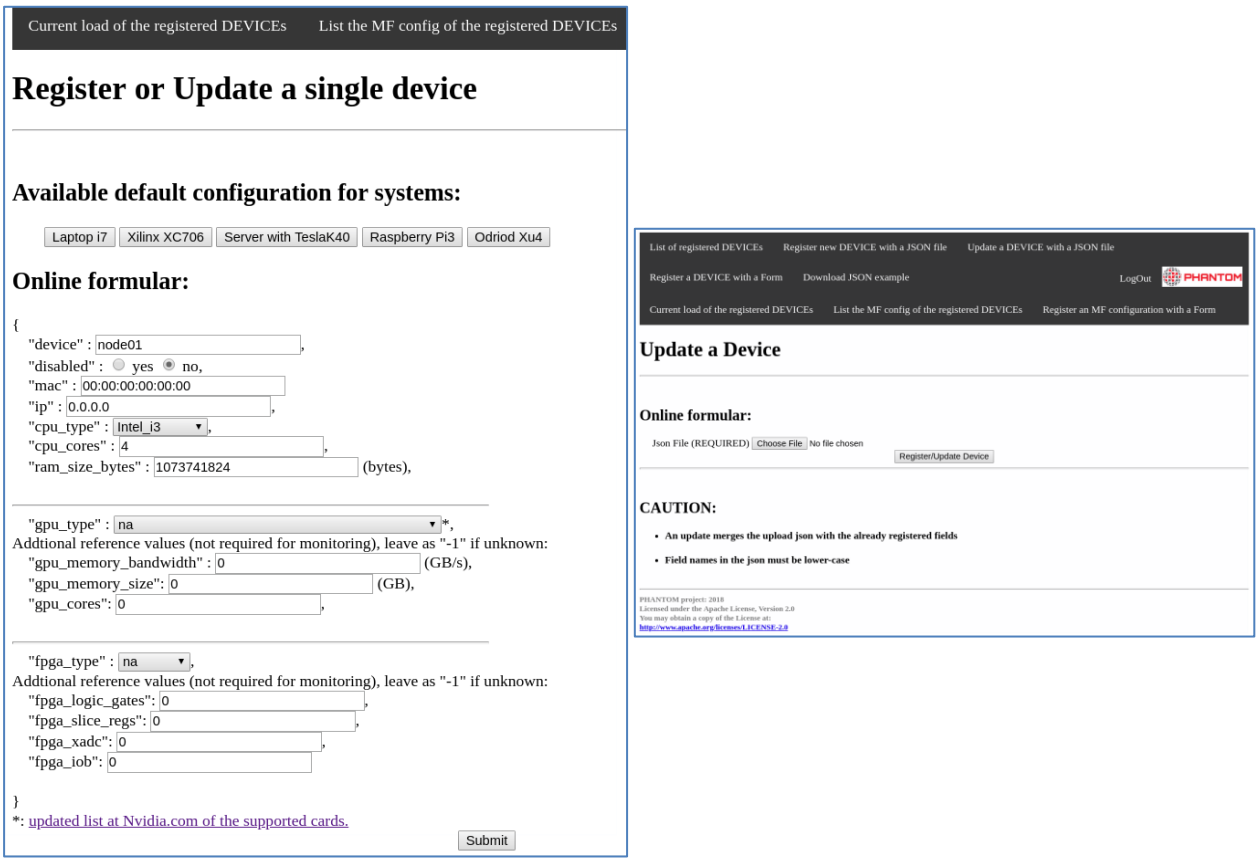
Figure 8: Interaction between Resource Manager and the users and PHANTOM tools

The RM manager provides a simple web interface for registering or modifying the Registered Devices, their Monitoring Configuration, manually, as well as a RESTful interface for updating it when new devices are added to the system. In addition, the RM manager provides also a web interface and a RESTful interface to query the last registered status or load of the monitored devices.

Interface for Registering Devices

The description of the registered devices consists of a list of values which helps to identify the device (such a name or IP address) and a description of the hardware resources such processors, memory, and installed accelerators. This information will be accessible to the users and the PHANTOM toolset.

Figure 9(a) shows the web interface for registering the devices. The form supports the users by automatically filling the details when the user selects some standard devices, for example the form provides the characteristics of the most recent NVidia cards. Figure 9(b) shows an alternative registration process based on providing the device description with a JSON file.



(a) **(b)**

Figure 9. Screenshots of the web interface for register or update a device, (a) using a form, and (b) providing a JSON file with the description.

The RM manager also provides a RESTful interface for registering devices. Listing 3 shows the query for registering/updating devices, which requires the user or PHANTOM tool to provide the hardware description in a JSON file. As an example, Listing 4 shows a simple hardware description in JSON format.

Listing 3: Example of register/update a device with a JSON file throw the RESTful interface.

```
curl -s -H "Authorization: OAuth ${mytoken}" -H "Content-Type: multipart/form-data" -XPOST -F "UploadJSON=@../web/device.json" http://${server}:${resourcemanager_port}/register_new_device;
```

Listing 4: Example of a Device Description (JSON).

```
{
 "device": "Raspberry pi3",
 "cpu_type": "armv7",
 "cpu_cores": 4,
 "ram_size_bytes": 1073741824,
 "mac":"00:11:22:33:44:55",
 "ip":"192.168.0.12",
 "gpu_type":"NA",
 "fpga_type":"NA",
 "disabled": "false"
}
```

The web interface also provides access to already registered devices, as shown in Figure 10 and in Listing 5.

Listing 5: Example of query for the description of a registered device.

```
curl -s -H "Authorization: OAuth ${mytoken}" -XGET http://localhost:8600/query_device?device="rpi3"
```

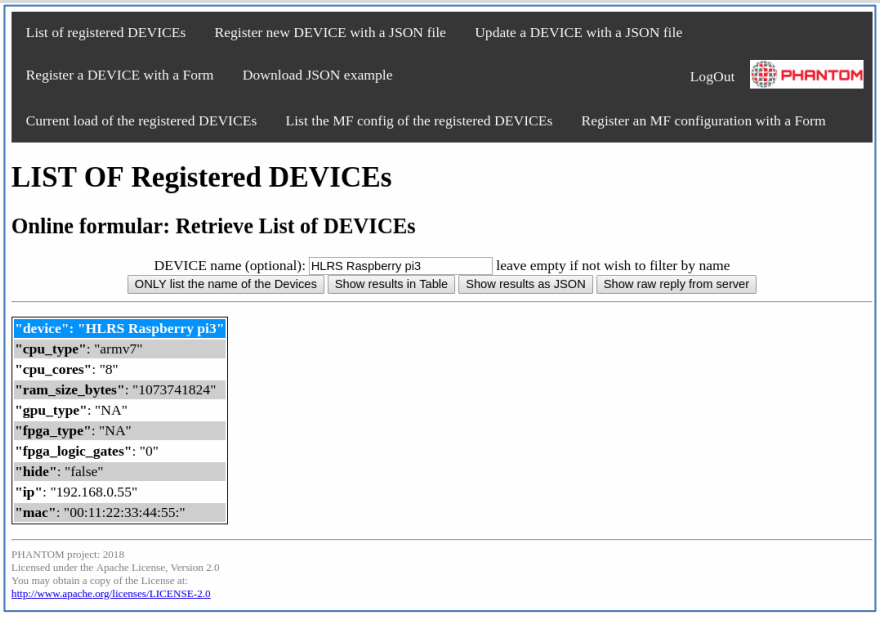


Figure 10. Screenshots of the web interface which shows the description of a registered device.

Interface for Registering the Monitoring Configuration (MC)

The configuration consists of a list of plugins and the parameters for each of them. Available plugins are defined by the available hardware resources (e.g. x86 CPU, ARM-based CPUs, etc.), and the parameters allow a trade-off between more frequent measurements and reduced the impact on the measured devices.

Figure 11 shows the web form for the configuration of the Monitoring parameters for a single device.

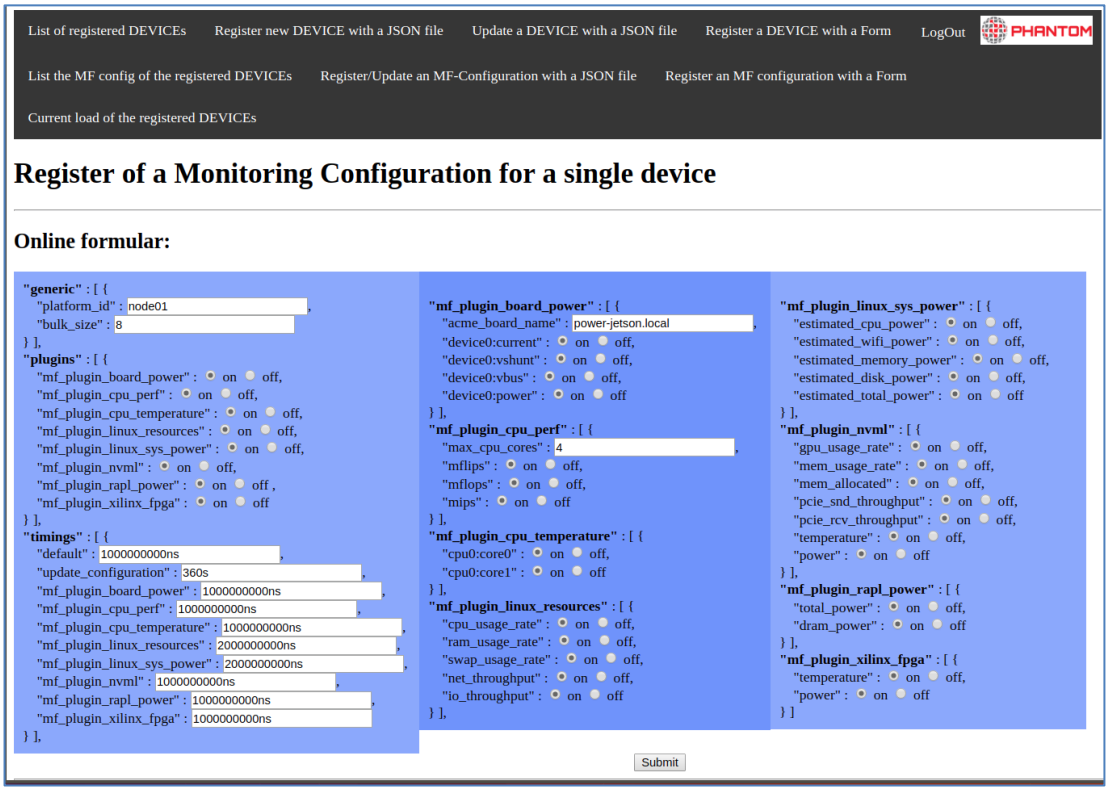


Figure 11. Screenshot of the web interface form for the configuration of the Monitoring parameters.

Listing 6 shows updating a Monitoring Configuration from the command line, and
Listing 8 shows the content of JSON file used on that query.

Listing 6: Example of register/update a device with a JSON file throw the RESTful interface.

```
curl -s -H "Authorization: OAuth ${mytoken}" -H "Content-Type: multipart/form-data" -XPOST -F "UploadJSON=@../web/device.json" http://${server}:${resourcemanager_port}/ register_mf_config;
```

Listing 8 shows retrieval of the registered Monitoring configuration in JSON format.

Listing 7: Example of query a Monitoring Configuration of a single device throw the RESTful interface.

```
curl -s -H "Authorization: OAuth ${mytoken}" -H "Content-Type: multipart/form-data" -XGET http://server:port/query_device_mf_config?pretty=true\&device="devicename";
```

Listing 8: Example of a shortened Monitoring Configuration (JSON).


```
{"hits" :[
   "registered_id": "AWRbSGcjwHjCws13Gi1h",
   "generic":  {
        "platform_id": "node01",
        "host": "node01",
        "host_length": 6,
        "bulk_size": 8
     } ,
   "plugins":  {
        "mf_plugin_Board_power": "on",
        "mf_plugin_CPU_perf": "on",
        "mf_plugin_CPU_temperature": "on",
        "mf_plugin_Linux_resources": "on",
        "mf_plugin_Linux_sys_power": "on",
        "mf_plugin_NVML": "on",
        "mf_plugin_RAPL_power": "on"
     } ,
   "timings":  {
        "default": "1000000000ns",
        "update_configuration": "360s",
        "mf_plugin_Board_power": "1000000000ns",
        "mf_plugin_CPU_perf": "1000000000ns",
        "mf_plugin_CPU_temperature": "1000000000ns",
        "mf_plugin_Linux_resources": "2000000000ns",
        "mf_plugin_Linux_sys_power": "2000000000ns",
        "mf_plugin_NVML": "1000000000ns",
        "mf_plugin_RAPL_power": "1000000000ns"
     }
]}
```


Interface for retrieving the registered info and status of the infrastructure

Listing 9 shows performing a query from the command line, and Listing 10 shows the reply.

Listing 9: Example of a query of a registered status of a device throw the RESTful interface.

```
curl -s -H "Authorization: OAuth ${mytoken}" -H "Content-Type: multipart/form-data" -XGET http://server:port/query_device_status?pretty=true\&device="devicename"
```

Listing 10: Example of a registered status of a device (JSON).


```
{"hits" :[ {
   "host": "node01",
   "host_length": 6,
   "type": "CPU_perf",
   "type_length": 8,
   "local_timestamp": "1530089901912.6",
```

```
   "core00:MIPS": 66.13,
   "core03:MIPS": 40.076
}, {
   "host": "node01",
   "host_length": 6,
   "type": "Linux_sys_power",
   "type_length": 15,
   "local_timestamp": "1530089910254.1",
   "estimated_total_power": 105.139,
   "estimated_CPU_power": 0,
   "estimated_wifi_power": 0,
   "estimated_memory_power": 104.386,
   "estimated_disk_power": 0.753
}, {
   "host": "node01",
   "host_length": 6,
   "type": "Linux_resources",
   "type_length": 15,
   "local_timestamp": "1530089910254.3",
   "CPU_usage_rate": 16.22,
   "RAM_usage_rate": 65.593,
   "swap_usage_rate": 0,
   "net_throughput": 0,
   "io_throughput": 17638.43
}]}
```


The RM also provides a simple web interface and a RESTful interface to query the status of the registered devices. The retrieved information shows the current load status (CPU, MEM, IO, Net,), and some additional fields helpful for the identification of the devices (ID, name, MAC-address, or IP address). As an example, Figure 12 shows the load of a device measured by three different plugins. Each plugin may collect data at different time intervals.

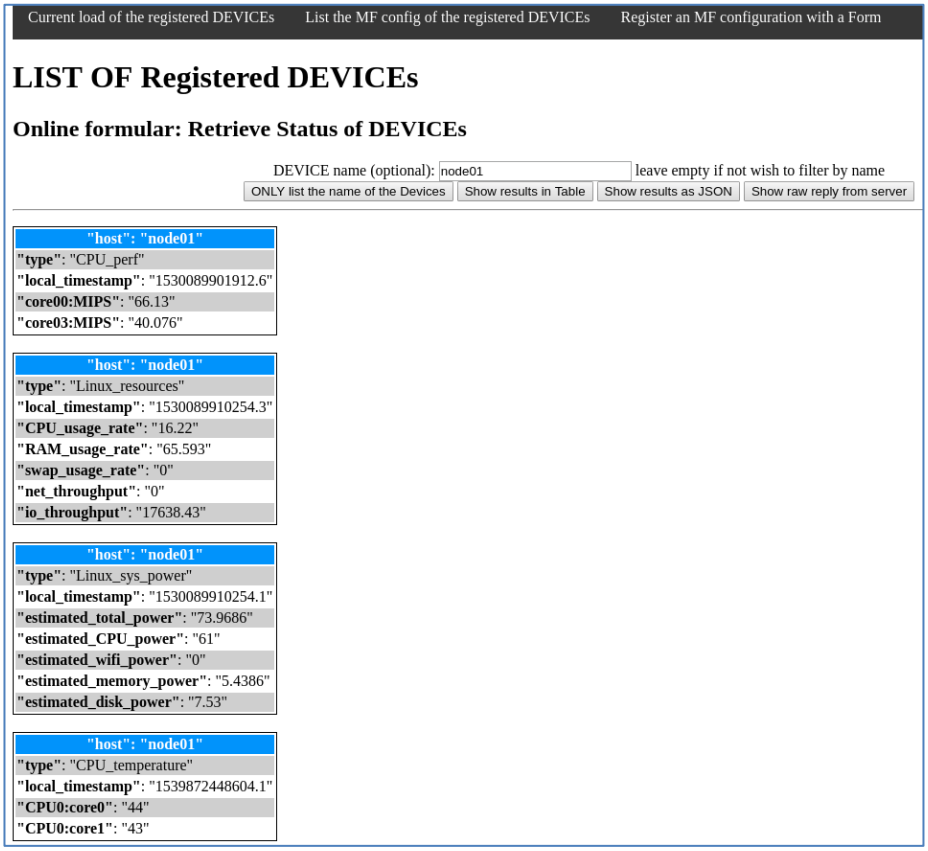


Figure 12. Screenshot of the web interface which shows the status (collected from 3 types of plugins) of a registered device.

The information collected in the previous query can combined to provide the Table 1.

Table 1. Example of registered status of heterogeneous infrastructure in the Resource Manager

Id	Name	Type	Load CPU	Load MEM	io_throughput (Bytes/s)	Load GPU	Load GPU-MEM	Energy	…
1	Node01	CPU	16.22%	65.59%	17638	NA	NA	73W[5]	

[5] Note: The instant energy consumption registered in the Resource Manager in Watts (W). While, the monitored energy consumption registered in the Execution Manager of the applications, which run for period of time, is registered on Joules (Energy W × time s).

3.3 IMPLEMENTATION AND INSTALLATION DETAILS

The RM server is composed of a data storage layer, used to persistently store the monitoring information and configuration parameters, and a web-service for the data transmission from the agents to the data storage layer (cf. Figure 13).

The Web service is written in JavaScript under the Node.js runtime environment. Node.js uses an event-driven architecture and is suitable to be used for data-intensive real-time applications that run across distributed devices. A free and open-source framework for Node.js – Express.js is used to build the RESTful APIs.

For the data storage component, ElasticSearch is used. It is a flexible and powerful open-source, real-time search and analytics engine. As a distributed, multi-tenant full-text search engine, it is preferred as supporting RESTful web interface and using schema-free JSON documents.

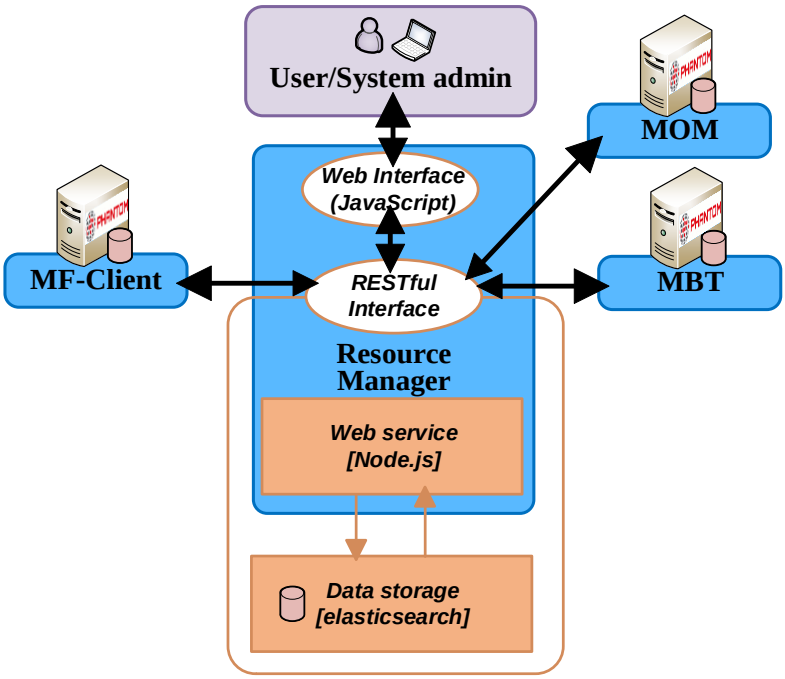


Figure 13. Software frameworks and interfaces of the Resource Manager

Installation scripts are supplied, so that the installation procedure is straightforward for the users (Listing 11).

Listing 11: Resource Manager setup

```
$ svn export https://github.com/PHANTOM-Platform/Resource-Manager phantom_rm
$ cd  phantom_rm

# Execute one of the following commands based on the architecture of the server
```

```
# 32-bit Intel
$./setup-server-X86-32.sh

# 64-bit Intel:
$./setup-server-X86-64.sh

# 32-bit ARM
$./setup-server-armv7l.sh

# 64-bit ARM
$./setup-server-armv7-64.sh
```

The Software dependences of the Resource Manager are listed in the Appendix 4 “Resource Manager dependencies”.

4. FPGA LINUX DISTRIBUTION AND FPGA INFRASTRUCTURE

4.1 OVERVIEW

The PHANTOM Linux distribution is a standardised Linux distribution which contains the required libraries and support software for the PHANTOM targets. It serves as a fixed target for partner collaboration, but its main contribution is for targeting FPGA boards. When running on a supported FPGA board, it can automatically integrate PHANTOM IP cores into the platform. In the PHANTOM toolchain, once the Multi-Objective Mapper has placed components onto an FPGA target, the tools developed in this work automatically generate the necessary hardware designs and associated support software. A PHANTOM hardware design encapsulates a set of IP cores, makes them available to the application components running in the Linux distribution, and includes the various security and monitoring requirements of the PHANTOM platform.

This release contains facilities to construct the bootloaders, kernel, root filesystem, hardware design, and the various drivers and interfaces. It also implements monitoring requirements by providing access to power and bandwidth monitoring.

Certain areas of the Linux distribution and related infrastructure (for example, Open MPI) can be built using Docker (https://www.docker.com), in order to target nonstandard platforms without requiring virtual machines or installing full toolchains for the target device on the build host.

A short list of the major features is as follows:

- The distribution includes support for PHANTOM monitoring actions, for example to read current power use.
- Support for communications between a Linux user space process and PHANTOM IP cores. IP cores are mapped to the User-space I/O subsystem so processes can map the address space of the IP core into that of the user space process.
- Support for automatic creation of FPGA hardware designs which encapsulate the PHANTOM IP cores developed as part of the project. A hardware design consists of a set of PHANTOM IP cores, wired up appropriately, and a further set of supplementary cores for tasks such as clock management, monitoring and debugging, and bus arbitration. When the Multi-Objective Mapper assigns a set of IP cores to a given FPGA platform, the generation tools assemble the design from the specified PHANTOM IP cores and the necessary supplementary cores, and wires everything appropriately.
- Automatic integration of communications APIs to allow the FPGA to be used as part of a heterogeneous compute platform.
- Automatic integration of security features when required to ensure execution integrity.

4.2 DESIGN SPECIFICATIONS AND INTERFACES

4.2.1 Usage

Full usage instructions are included with the software, which detail how to customise the environment to more specific needs. This section only provides a brief quick start overview.

First, set the TARGET environment variable to refer to the desired target FPGA board.

```
export TARGET=zc706
```

Copy the included pre-built kernel and boot images to be used on the board:

```
./make.sh prebuilt
```

If necessary, download updated packages and rebuild a new root filesystem:

```
./make.sh rootfs
```

To integrate PHANTOM IP cores into the system:

```
./make.sh hwproject ipcore1 ipcore2
./make.sh implement
```

And finally to copy the system images onto a boot SD card:

```
./make.sh sdcard
```

4.2.2 Memory Partitioning

Unless otherwise specified, each IP core will be assigned an equal partition of device memory. To specify IP cores with differing memory sizes, a configuration file can be created where each line is the name of the IP core to integrate, followed by the desired memory aperture size. For example:

```
ipcore1 128kb
ipcore2 4mb
ipcore3 256kb
```

And run using:

```
./make.sh hwproject configfile.conf
```

Each size must be a power of two and in a format supported by the Xilinx tools. Accepted postfixes are `kb`, `mb`, `gb`.

If more memory is specified than is available, the build will fail.

4.2.3 Security Considerations

If a component mapped to the FPGA infrastructure requires security isolation then additional capabilities are added to the infrastructure to ensure that malicious IP cores cannot affect system integrity, and that cores handling sensitive data are protected from observation. A PHANTOM component consists of hardware and software parts, and both of these need to be considered.

Software

The isolation of software components running on the FPGA platform relies on standard Linux kernel process isolation. Each process is run as a standard user with limited privileges to avoid any potential interference. A superuser account is then used to manage the processes, including their hardware access permissions.

Hardware components are controlled from the ARM cores using memory-mapped registers in the CPU's standard address space, as well as including a portion of shared memory for each component that both the hardware and software can access. Therefore, the Linux user-space process for each component requires access to both of these areas to communicate with the hardware. This is implemented using UIO.

The kernel's UIO system allows predefined areas of physical memory defined in the Linux device tree to be mapped into user-space processes, with a device node created for each individual component in the system. Each UIO device node has mappings for the control and status registers of a component, and its associated shared memory area. The read/write permissions of each UIO device node can be set by a superuser account or using `udev` rules, allowing the mapping of each memory area to be restricted to a single component's user account. Each component has UIO mappings for its control and status registers, along with a dedicated area of system memory shared between the software and hardware. This ensures that PHANTOM cores only have access to the memory spaces that they are allowed to use. A fragment of these `udev` rules are shown in Figure 14.

```
SUBSYSTEM=="uio",KERNELS=="40000000.phantom_axi",
  SYMLINK+="phantom/component0", OWNER="phantom0",
  GROUP="phantom0", MODE="0600"
SUBSYSTEM=="uio", KERNELS=="41000000.phantom_axi",
  SYMLINK+="phantom/component1", OWNER="phantom1",
  GROUP="phantom1", MODE="0600"
```

Figure 14: Example udev rules to enforce software access

Hardware

The master AXI interface on each FPGA component for accessing shared memory can read and write any address available on the bus by requesting the transfer. This means that even if the software side of a user's component was isolated, a malicious hardware core could compromise system integrity.

In order to limit accesses only to valid addresses and to guarantee memory isolation between components, system memory is statically partitioned according to the configuration provided when the system was built. Access is controlled using a memory management unit (MMU) core on each component's AXI master bus. Using this mechanism, bus requests from a component are restricted to only its pre-allocated memory area, which is the same area mapped through to the corresponding UIO device in Linux. Therefore, no hardware component can access the shared memory of any other component, or the memory of any software running on the ARM cores (including the Linux kernel), either intentionally or accidentally. This is shown in Figure 15 and Figure 16.

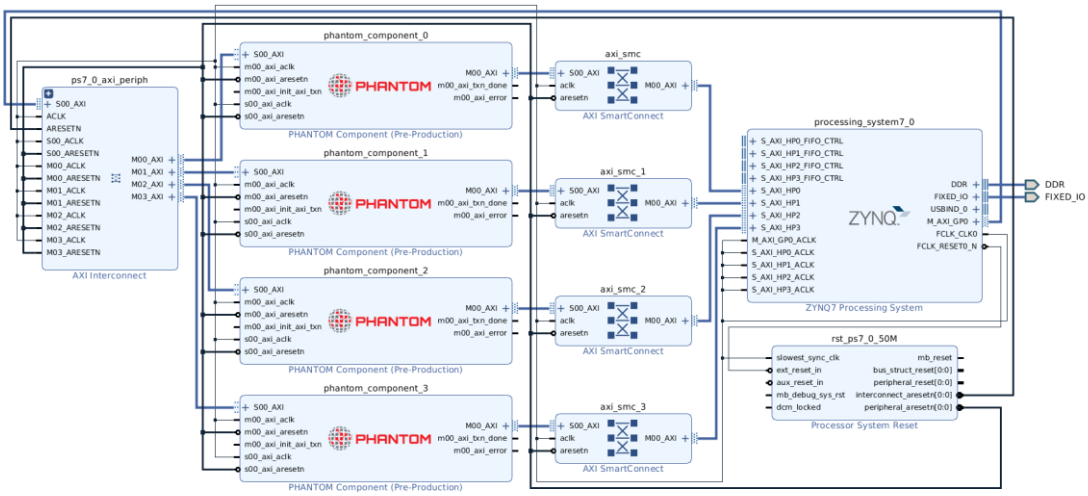


Figure 15: AXI SmartConnect MMU used to restrict hardware access to memory on new FPGA devices.

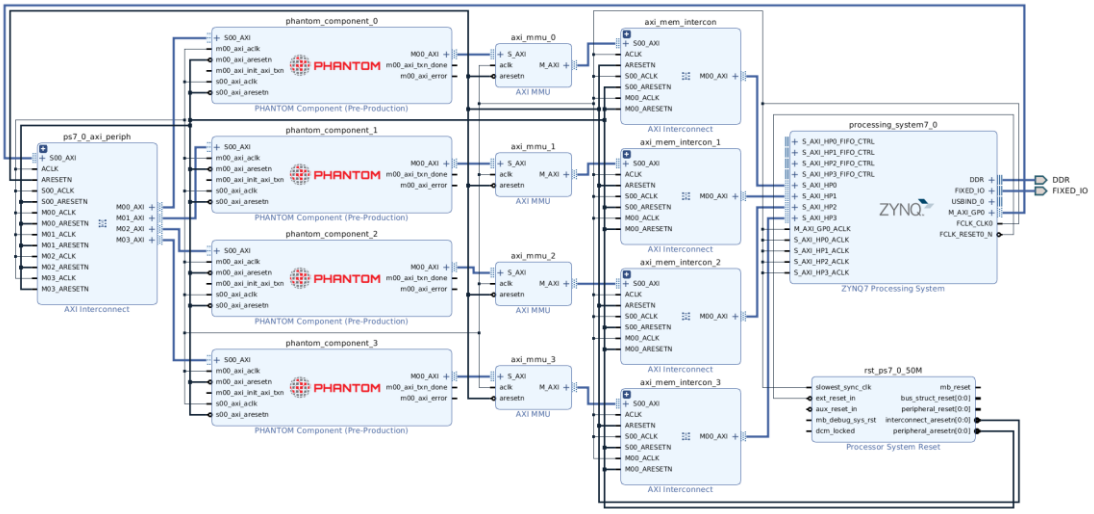


Figure 16: Memory protection is supported on older FPGA devices with the AXI MMU and AXI Interconnect cores.

4.3 IMPLEMENTATION DETAILS

4.3.1 Installation

Full installation and building instructions are in the source repository. The README file explains how to build the kernel, device tree, bootloaders, and filesystem for a given target FPGA board. It also explains the process of automating the bitfile generation for your target device. Begin by checking out the repository:

```
git clone https://github.com/PHANTOM-Platform/PHANTOM-FPGA-Linux.git
```

Note that building any hardware designs requires the Vivado Design Suite from Xilinx. The Linux distribution is optionally built using Multistrap (https://wiki.debian.org/Multistrap) which must be installed, along with the necessary cross-compilers for ARM and related tools.

4.3.2 Filesystem Configuration

In order to accommodate the entire range of use cases, the PHANTOM Linux filesystem is configurable to a range of sizes. Debian and Ubuntu-based systems are available for applications that can be run from an SD card or similar. These builds use Multistrap and vary between 200-400MB in size. For smaller applications, a BusyBox-based system can be selected. This is designed to be run as an ephemeral ramdisk, loading from a storage device (potentially on-board flash storage) into system memory each time the device is reset. Because of this, the size is far smaller than the full Debian system, typically under 10MB.

Example configurations for a range of applications are available in the PHANTOM repository.

5. IP-CORES MARKETPLACE AND GENERATOR

5.1 OVERVIEW

In order to achieve the required support for FPGAs in PHANTOM two different modules were developed to tackle different approaches. First there is the IP Core Marketplace, which can be seen as a curated repository for IP Cores that implement a specific mathematical algorithm and are optimized for that purpose. Secondly, there is the IP Core Generator which generates IP Cores on demand for user algorithms that cannot be found in the Marketplace, but the developer still desires to run them on FPGA logic. These modules are described in more detail bellow.

The IP Core Marketplace is a part of the PHANTOM Repository specifically for storing all the dedicated FPGA logic blocks that were manually optimised for distinct algorithms. The IP Cores in the Marketplace will serve as accelerators for specific functions, commonly used mathematical algorithms (e.g. Fast Fourier transform (FFT), Finite impulse response (FIR), etc.), or image processing filters (e.g. Sobel Filter, Discrete wavelet transform (DWT), etc.) implemented in FPGA logic fabric. Being handcrafted, this IP Cores can be deeply optimized by a specialized hardware engineer to assure they will have the best performance and resource usage optimization.

The IP Core Generator serves the purpose of taking over where the Marketplace can't cover. When the user wants to exploit FPGAs but there isn't an IP core available in the Marketplace, that covers the user specific needs, the IP Core Generator can provide the desired functionalities to the user. If the user desires to run a certain component of the application in FPGA Logic, the code can be annotated to indicate that a certain function should be transformed into an IP Core. The IP Core Generator will take care of transforming the user source code and inserting the appropriate interfaces in accordance to the PHANTOM platform. Although the generated IP cores won't have the same level of efficiency and performance as custom-made IP cores this tool allows for a developer without any hardware knowledge to still be able to exploit FPGA logic and its benefits.

5.2 DESIGN SPECIFICATIONS AND INTERFACES

5.2.1 IP Core Marketplace

The IP cores available in the Marketplace are provided with an accompanying software component that will control the IP core for the user. This software component can be seen as a driver or bridge between software and FPGA hardware. The provided driver deals with all the needed initializations, synchronisms and memory transfers between software and IP core.

The IP cores and respective software components, available in the Marketplace, are compliant with the interfaces defined in the PHANTOM FPGA Linux Infrastructure and are intended to be deployed in an FPGA running the PHANTOM FPGA Linux Distribution.

To deploy one of the IP cores available in the Marketplace, the developer must create a new component implementation that includes and calls the functions from the IP core accompanying software driver instead of the standard software version. The developer must also edit the component network and add a new implementation with the paths to the new component files that use the IP core. The Deployment Manager will then read this new implementation from the component network and call the respective tools to compile and synthesize all the application components.

5.2.2 IP Core Generator

The IP Core Generator is integrated with the repository, subscribing to project updates to receive notifications when new deployment plans are available. When a notification, that new deployment plans are available, is received, the IP Core Generator searches for components mapped to FPGAs in the deployment plans, to find if there are any components that do not have a corresponding IP core in the Marketplace.

When a component which needs to have an IP Core generated is found, the respective source and header files are pulled from the repository to proceed with the code analysis and transformation. If the IP Core generation succeeds, a new version of the component is created with the needed interfaces and software to control the IP core.

The generated files, for the IP core and software component, are then uploaded to the repository to be available for the other PHANTOM modules to use when needed. The Deployment Manager can then use this new component implementation, when appropriate, the same way as the original, expect its target will be a Zynq device instead of a CPU.

5.3 IMPLEMENTATION DETAILS

5.3.1 IP Core Marketplace

The developer can directly replace the original function with the provided software-FPGA adapter as it should have the same input and output parameters. This way the user application can call the Zynq specific methods instead of the standard methods without the need for any major code restructuring.

The new software component, that was created by the developer for Zynq devices using the provided software-FPGA adapter, should be selected by the Deployment Manager when the MOM maps the corresponding component to run on a Zynq device. Though, this requires that the developer adds the new component as an alternative implementation in the component network.

The components will work in the same manner except they can exploit the FPGA resources to do the same job in parallel hardware instead of software. This ensures that the IP cores available in the Marketplace will function correctly while greatly simplifying the integration process.

5.3.2 IP Core Generator

An instance of the Clang Compiler is used to manage the various objects needed to run the source code rewriter. Clang is used to create an abstract syntax tree (AST), a tree representation of the abstract syntactic structure of source code that is used to regenerate the two new versions of the component.

One version is the original source code implementation with the addition of some pragmas that specify the IP core interfaces. These and are required by Vivado HLS to generate the proper interfaces in accordance to what is supported by the PHANTOM Platform. The IP Core Generator needs to have access to the Xilinx tools including Vivado HLS and Zynq board support files installed locally in the machine where it will run, since it uses these toolkits to generate the IP core files.

The second version of the component that is produced by the IP Core Generator is the adapter that will serve as a bridge between the software component and the IP core that is instantiated in FPGA logic. This adapter will map the memory used by the IP core and copy the software functions inputs to the respective memory areas. Then, it instructs the IP core to run and takes care of the synchronism between software and hardware. When the IP core has finished all the work, the outputs are copied back from the IP core memory to the function outputs and return variables. The outputs, inputs and respective sizes must be specified by the developer with pragmas to allow the generator to map input and output variables to the proper IP core memory regions.

All the produced files, IP core and modified software component, are uploaded to the repository after being generated for an easier integration with the rest of the PHANTOM modules.

6. DEPLOYMENT MANAGER

6.1 OVERVIEW

The Deployment Manager automates the deployment procedure due to the different modifications that need to occur for the components' execution on complex, heterogeneous environments. The different adjustments and code refinements are executed as a final stage of the application's development following the analysis and results of the Parallelization Toolset and the decisions of the Multi-Objective Mapper. Additionally, a set of scripts corresponding to the aforementioned procedure, is generated implementing the actual deployment of the components on the available hardware.

6.2 DESIGN SPECIFICATIONS AND INTERFACES

As already mentioned, the Deployment Manager is responsible for the source code modifications and compilation/deployment scripts in order to facilitate deployment on CPU and GPU targets. In this section, these steps are described in detail, while in the following section some lower-level implementation details are included.

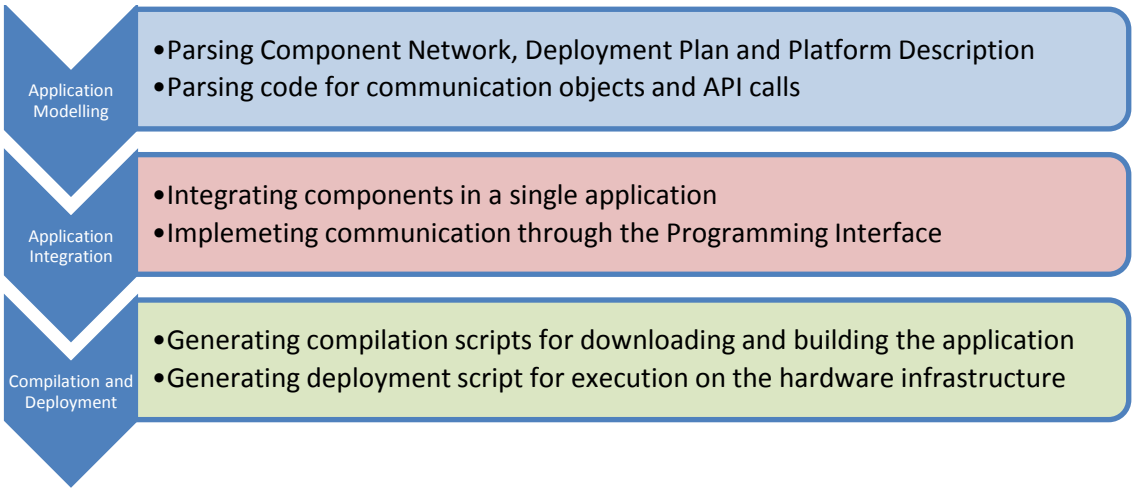


Figure 17: Deployment Manager functionality

As shown in Figure 17, the tool flow of the Deployment Manager can be split into three stages. First, there is the modelling of the application where a set of Java classes are used to describe the components, the way they communicate with each other, their mapping on the hardware platform and information about the data or signals they need to exchange. Second, the generation of the necessary files, which enable the integration of the components with each other, will take place implementing in this way the communication between the different parts of the application. Finally, a set of scripts will be generated for building and placing the binaries on the corresponding machines.

6.2.1 Application Modelling

In order to orchestrate the deployment as instructed by the MOM and the Parallelization Toolset's Technique Selection, the Deployment Manager needs to collect all the necessary information to model the application per component accompanied by the

corresponding communication objects. This information is suitably encapsulated in the Component Network and Deployment Plan as well as inside the source code in the form of pragma annotations.

Analyzing modelling-XML files

First, the necessary information is extracted from the Component Network in order for an application image to be constructed using a set of Java classes. Information like the components' names, source and paths in the Repository is obtained, along with the components' positioning inside the component network, meaning their interactions with other components and the corresponding communication objects.

The Platform Description is also analysed for data concerning the available hardware components. The design of the system architecture is included in the application model as well as information concerning each individual machine or device. So, attributes like frequency, bandwidth, cache memory, cores, IP address are gathered for the coordination of the deployment on the hardware infrastructure. The correlation between the software components and their communication objects, and the hardware components of the platform is achieved with the analysis of the Deployment Plan. Each component and communication object is mapped on a specific hardware resource, thus this connection is also included in the application model completing its structure.

Communication object identification

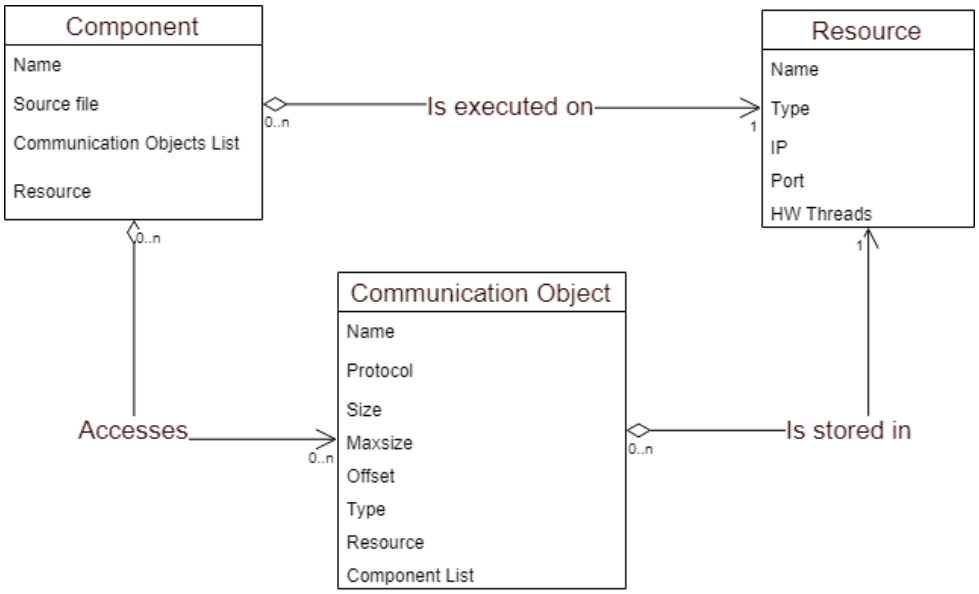
The PHANTOM framework creates a common environment for the components to be executed in, at a higher logical level than the one that the components work in. In order to identify the communication data and link the local variables that refer to the same communication objects, a pre-compilation pass of the code will initiate the analysis. Specifically, pragma annotations provided by the programming model will be exploited, so that the communication objects can be declared in a higher-level environment.

The annotations used are the following and they are described in detail in the "Implementation" section.

```
#pragma <queue | shared | signal | mutex> <in | out | inout> name
```

When a communication object is declared, the corresponding information is declared as well, like its size, the name of the local variable that the data is stored in, the protocol that is used for its transfer etc. According to the protocol that's defined for a communication object, the way that this object is accessed changes, while the consistency requirements of the data is defined as well. Some ensure sequential consistency or stronger, whilst others define no consistency and the programmer must manually force synchronization.

The application model is completed with the information extracted from the source code, thus the Deployment Manager can proceed with the implementation of the actual deployment. The architecture of the model that has been created follows the structure of the following diagram:



6.2.2 Constructing a multi-process application

There are multiple libraries that can be used for the parallel execution of the components. Guided by the library selection of the Technique Selection module, the Deployment Manager implements the coordinated execution of the components, along with the communication that is needed between them. The PHANTOM Programming Model, guarantees the independence of the components so that the communication methods that are used can be chosen among a variety of options including different libraries, technologies and protocols. Therefore, the components can run as independent processes that can communicate over the network, via socket mechanisms or even using local or cloud storage for communication through the PHANTOM Programming Interface. The different implementations can differ according to the different requirements that are encountered during development.

The key innovation is that the Deployment implementation is automatically chosen based on the application's architecture and the target platform. For example, if the Multi-Objective Mapper has decided that all the components should be executed on the same machine/SMP then deployment implements the top-right case from Figure 12, in which the program spawns multiple threads to initiate and execute the components. Communications are implemented using shared memory.

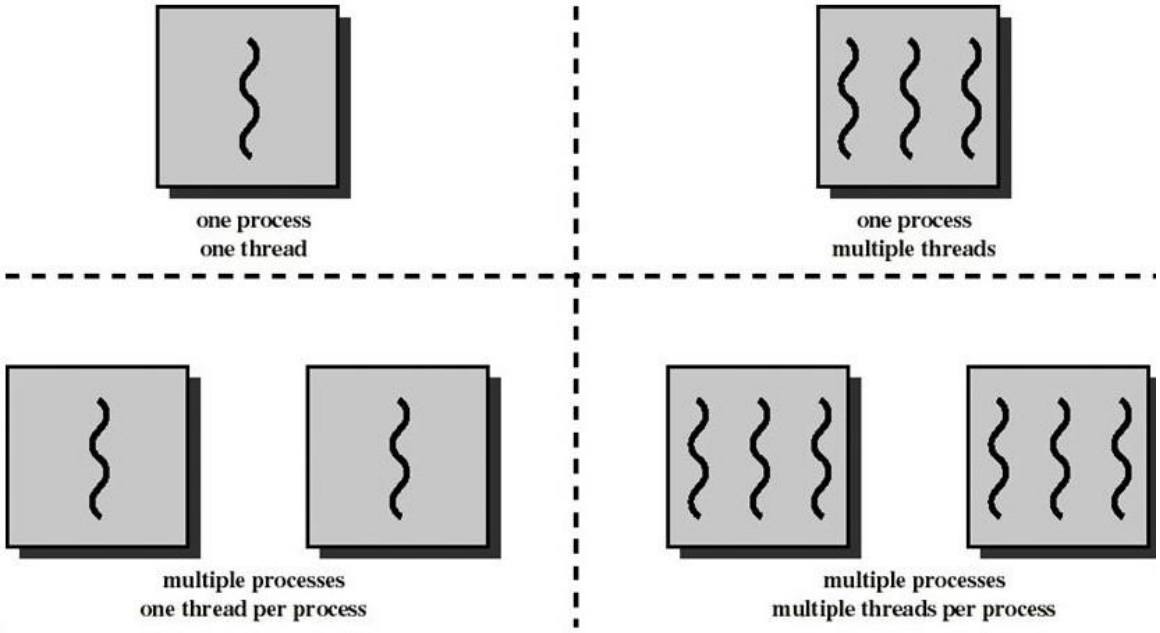


Figure 18: Different implementations of an SPMD program using threads and processes

Similarly, when the MOM has instructed the application to be executed on a more complex environment than a single machine (e.g. to use different nodes of a cluster system), a message-passing protocol is needed for exchanging data in a distributed memory space, so the MPI protocol is adopted. This is the bottom-left case in Figure 12. In this case, the different components are executed each one as a separate MPI process, according to the deployment plan, all running in parallel. The Programming Interface implementation uses the corresponding MPI functions to transfer data and synchronization signals between the components. The Deployment Manager is responsible to generate the necessary files to bind the components into an MPI application and declare the structures that are needed for implementing the communication. These files are described in detail in the 'Implementation' section.

A more complex implementation will try to reduce the overhead caused by the creation of different MPI processes, while still exploiting the multi-core capabilities of the available machines. For this purpose, the execution of components running on the same memory space is implemented as different threads of the same process. The result is similar to the bottom-right case in Figure 12. A set of MPI processes is initiated (one for each different machine) each one of which spawns a number of threads for every component that is mapped on its memory space. This is the most generic case of the above, where the parallelization of the different components is achieved in a local-memory level from a thread-based perspective, but also in a distributed-memory level from a multi-process one. In the "Implementation" section, the state-of-the-art approaches for combing MPI with thread spawning technologies are discussed.

All approaches have their own benefits and drawbacks concerning including additional time overhead, power consumption or memory footprint. The Deployment Manager uses some simple rules to decide the best approach, as it is only the tool to implement the

deployment plan. In specific, the tool tries to avoid the overhead added by the usage of MPI calls as much as possible, so it creates a single process for every different address space, while at the same time it spawns different threads for components that are executed in the same environment, exploiting the shared memory of the system for intra-node communications. The integration of the application described above is concentrated in the generation of an additional, 'main' component that is responsible to handle the necessary memory allocation and objects' initialization. This component acts as the initialization point of the application and takes up the actual implementation of the decisions described above. The implementation of this component is described in detail in the "Implementation" section.

6.2.3 Integration with the Programming Interface

The implementation of the Programming Interface is highly dependent on the implementation model that is selected for the application like explained in the previous paragraph. That is why during the integration of the application components, the necessary structures are generated as well, following the patterns suggested by Technique Selection in order to implement the communication between the components in the hardware environment that is decided by the Multi-Objective Mapper. Details about the aforementioned structures that are used can be found at the "Implementation" section.

The initial design of the Programming Interface was characterized by the independence it would provide to the components to access the necessary data whenever they needed to implementing one-sided communication channels using RMA (Remote Memory Access) operations provided by the MPI protocol. However, during development, the need for dynamic data transfers arose, demanding the allocation of memory space at run-time, which is infeasible when one-sided communication is used. That's why a new approach has been followed allowing the target machines to actively communicate with the source of the transfer. In specific, each machine will use a different thread as a *listener* agent that will act as the receiver of incoming messages, receiving data and allocating the corresponding memory space that is necessary for the storage of the incoming data at run-time. With this approach, data transfers include an active target and thus create a bidirectional channel, providing flexibility to the communication patterns requested by the user. The implementation of this functionality is handled by the main component of the application that was described in the previous section. The implementation of the Programming Interface has been modified accordingly and the updated functions can be found at the Appendix section.

6.2.4 Executable generation and deployment

The deployment of the components on the given hardware infrastructure is the core function of the Deployment Manager. While this is true, the actual deployment is an action that is assigned to the user via a set of scripts that are generated by the tool. Guided by the Deployment Plan, the DM creates the compilation scripts in order to build the application executables on the corresponding machines/devices when the user selects to execute them. The following steps show the actions that are completed by the scripts' execution.

1. *Check for the necessary* executables *on the target machines.*
2. *If not found on machines*
    a. *Check on Repository.*
    b. *If found on Repository*
        i. *Download* executable *from Repository*
    c. *Else*
        i. *Download project source folder from Repository*
        ii. *Compile source files and generate corresponding* executable
        iii. *Upload on Repository*
3. *Download necessary input files (check if they already exist on the machines)*

Unnecessary downloads are avoided by checking locally for the files when needed. This is for both executables and input files. Checksums are used to determine whether a remote file has changed.

A deployment script is also generated for the initiation of the components' execution on the heterogeneous hardware architecture. This contains the deployment directives about where to execute each component. Due to the locality of the application that is promoted by the programming model, the Deployment Manager makes all possible efforts to guarantee the software components are bound to the appropriate hardware components in the hardware platform, as determined by the user's deployment constraints. For this purpose, the OpenMPI library provides specific functionalities during execution, which are fully exploited by PHANTOM to bind specific processes to specific processor cores, sockets etc.

The aforementioned scripts are available to the user for execution on the remote machine where the Deployment Manager is executed.

6.2.5 Interaction with other PHANTOM modules

As with all PHANTOM tools, the Deployment Manager is part of the overall architecture and needs to communicate with the other modules to complete its functionality. Its positioning in the platform is shown below:

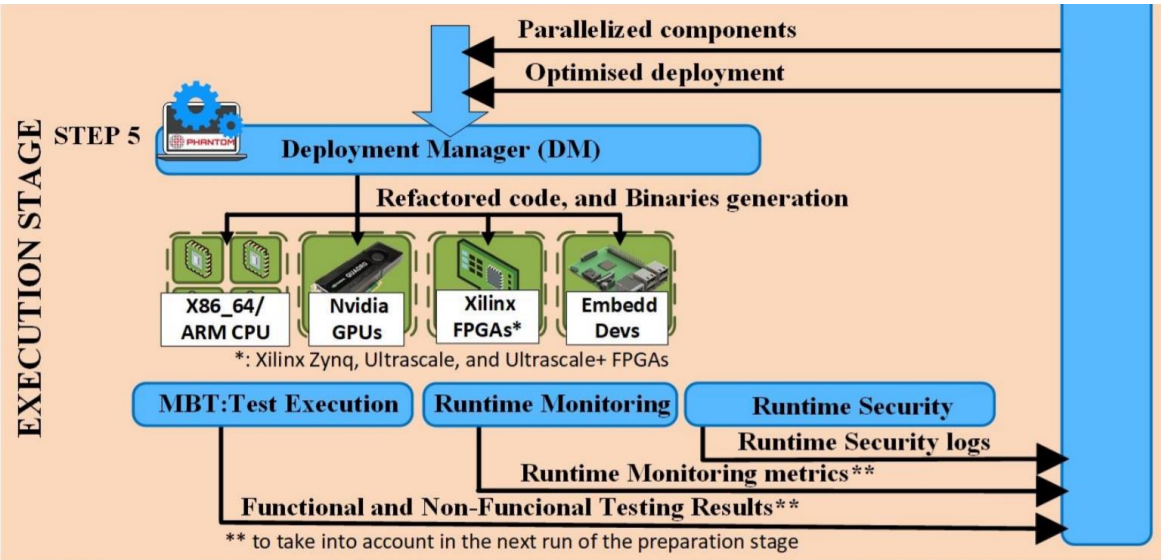


Figure 2: Deployment Manager positioning in the PHANTOM platform

It only needs to interact with the Repository, obtaining the files needed for the deployment. These include:

- The Platform Description, where the hardware infrastructure is defined
- The Component Network, where the structure of the application is described
- The Deployment Plan, where the mapping designed by the Multi-Objective Mapper is described
- The source files residing on the Repository after the PT's analysis

The Deployment Manager is notified to begin work after the Parallelisation Toolset has completed and uploaded its results into the Repository.

6.3 IMPLEMENTATION DETAILS

6.3.1 Communication Object Identification

The communication objects that are used by the application have to be declared like any other variable. For a communication object declaration, a pragma annotation is used defining the protocol of the object, its name etc.

The pragma annotations are described here in detail:

```
#pragma <queue | shared | signal | mutex> <in | out | inout> name
```

By declaring an object using this pragma, the local variable following the annotation is linked with the object. The size and type of the object is defined by the corresponding attributes of the local variable.

Info included in the local variable declaration:

- Local variable name
- Object size

- Object type

In the pragma annotation, the following attributes are also defined:

- Communication object name
- Protocol (queue or shared)
- Direction of the object (used as input, output or both)

The modelling of the application is completed with information extracted by the Component Network:

- Maximum amount of objects that can be inserted in the queue (only for queue protocol objects) - optional
- Offset for accessing specific data of an array stored in the shared memory. Helpful for avoiding moving big chunks of data between components for no real reason. (only for shared protocol objects) - optional

In the following example, some of the protocol functions that were defined in the PHANTOM Programming Interface are used between the components *CA* and *CB* to assist the understanding of the aforementioned pragma annotations. Let's examine the object declared as a shared-memory item using the identifier 'the_data'. This means that the name 'the_data' refers to the same array of 1024 items and of type *uint8_t*, declared to exist in the memory, shared between the two components. The local variables using the name 'data' are two different variables existing in different ranges of the application, but defined to hold the same chunk of data using the shared protocol.

```
// Component A

#pragma phantom signal out ready
bool startb;

#pragma phantom shared out the_data
uint8_t data[1024];

#pragma phantom queue in sum
uint32_t input_sum;

void CB() {
   phantom_shared *shared = phantom_shared_init(the_data);
   phantom_signal *signal = phantom_signal_init(ready);
   phantom_queue *queue = phantom_queue_init(sum);
   construct_input_data(data);
   phantom_synchronize(shared,data,1);  //  1  is  for  updating  the
shared memory
   phantom_signal(signal);
   input_sum = phantom_queue_get(queue);
}

// Component B

#pragma phantom signal in ready
bool startme;
```

```
// Getting the last 128 items from the 'the_data' comm. Object –
offset is defined at the Component Network as '896'
#pragma phantom shared in the_data
uint8_t data[128];

#pragma phantom queue out sum
uint32_t output_sum;

void CB() {
   phantom_shared *shared = phantom_shared_init(the_data);
   phantom_signal *signal = phantom_signal_init(ready);
   phantom_queue *queue = phantom_queue_init(sum);
   while(true) {
      phantom_wait(signal);
      phantom_synchronize(shared,data,0); // 0 is for updating the
local memory
      for(int i=0; i<128; i++)
         output_sum += data[i];
      phantom_queue_put(queue,&output_sum);
   }
}
```

In the example above, the application uses the shared protocol to transfer data from component CA to component CB. The call to the *phantom_synchronize* function with the direction attribute set to the value '1' updates the shared memory between the two components with the values of the data variable in component CA, while on the other side component CB calls *phantom_synchronize* with the direction attribute set to '0' to update the local memory with the values in the shared memory. In order to guarantee for the concurrency of the data, the programmer has used the signal protocol to coordinate the transactions between the components. The queue protocol is also used for moving the result of the calculations made in CB back to CA. No synchronization actions are needed here since the queue protocol can guarantee the consistency of the data.

6.3.2 Integration with the Programming Interface

The PHANTOM Programming Interface uses a set of structures to manipulate and transfer the communication objects. These structures represent the communication objects at a lower level to facilitate the implementation of the data transfers between the components. They store the necessary information in specific fields and are accessed by the APIs in order to complete their functionalities.

The following structures are used:

```
# SHARED PROTOCOL
typedef struct {
        void *data;
        int caller;
        int owner;
        int ID;
        int size;
        int offset;
        MPI_Datatype type;
        MPI_Comm comm;
} phantom_shared;
```

```
# QUEUE PROTOCOL
typedef struct phnode {
        void* item;
        uint32_t size;
        struct phnode *next;
} phantom_queue_node;

typedef struct {
        phantom_queue_node *front;
        phantom_queue_node *rear;
        pthread_mutex_t lock;
        volatile uint32_t count;
        uint32_t owner;
        uint32_t caller;
        uint32_t ID;
} phantom_queue;

# SIGNAL PROTOCOL
typedef struct {
        int ID;
        int owner;
        int caller;
        int value;
        pthread_mutex_t lock;
        pthread_cond_t cond;
        MPI_Comm comm;
} phantom signal;

# MUTEX PROTOCOL
typedef struct {
        int ID;
        int owner;
        int caller;
        pthread_mutex_t lock;
        MPI_Comm comm;
} phantom_mutex;
```

In order to provide transparency to the user as well as configurability to the implementation of the application, a simple interface (see D3.2 – the final interface will be included in D1.4) is used by the programmer to invoke the corresponding protocol functions. According to the decisions made by PHANTOM, the API calls invoke different implementations depending on the relation between the components' location. These implementations can be found in the Appendix section.

6.3.3 Constructing a multi-process application

The MPI protocol was selected for its efficiency in both UMA and NUMA architectures due to the memory locality strategies that it follows, as well as for its application on heterogeneous environments with fine-grain deployment options. The implementation of the produced application will follow the Single-Program-Multiple-Data (SPMD) model using the OpenMPI library. This means a single program is built and run on the hardware infrastructure in multiple executable instances, all following the same path initiating the corresponding components at the location they are executed.

This approach allows the easier representation of the execution flow as well as the independence between the different executables, so that repetitive compilations due to modified deployments can be avoided. This is achieved with a configuration file that is dynamically modified by the DM without any change occurring in the source code, so that the deployment directives can be provided to the program at run time. This means that the binaries of an application are only created and placed on the corresponding locations the first time that the Deployment Manager is executed for the specific implementation of the components. If different implementations of the components (meant for GPUs or FPGAs) are selected, new binaries are created if necessary, and stored on the Repository for future redeployments.

Specifying some implementation details, a main-function file stands as the starting point of the application, where all the necessary structures are allocated, including the communication objects, accompanied by a finalization function that deallocates any used space and ends execution. A common header file is used for creating the PHANTOM environment, holding the higher-level information that is needed for the implementation. For the generation of these source files, the application model that has been created in the previous stage is used for providing the corresponding information (e.g. local variables, communication object info etc).

A simplified version of the main file is shown in the following figure:

```
#include "phantom_init.h"

void invoke_component(int id) {
        if(id == 0) {
                CA();
        }
        else if(id == 1) {
                CB();
        }
        return;
}

void initialize(int *argc, char*** argv) {
        MPI_Init_thread(&argc,&argv,MPI_THREAD_MULTIPLE,NULL);
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

        Signal0 = new_phantom_signal(… args …);
        Mutex0 = new_phantom_mutex(… args …);
}

void finalize() {
         phantom_mutex_destroy(Mutex0);
         phantom_signal_destroy(Signal0);
         MPI Finalize();
}

int main(int argc, char** argv) {
        pthread_t phantom_pthread[LOCALCOMPS];
        int phantom cmpid[LOCALCOMPS];
        int i;
        for(id=0; id<LOCALCOMPS; id++) {
```

```
                phantom_cmpid[id] = id;
                pthread_create(&phantom_pthread[id], NULL,
phantom_call_components, (void *)&(phantom_cmpid[id]));
        }

        phantom_communication_manager();  // Handle communications

        for(id=0; id<LOCALCOMPS; id++)
                pthread_join(phantom_pthread[id],NULL);

        finalize();
        return 0;
}
```

As illustrated above, a set of functions, generated by the Deployment Manager, sets up the PHANTOM environment by allocating space for the necessary structures and initializing the corresponding values/objects, coordinates the execution and finally clears the environment space afterwards.

The variables that are used for the implementation of the communication objects in the above segment are declared in the 'phantom_init.h' file, which is displayed below (also in a simplified version):

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "SignalProtocol.h"
#include "MutexProtocol.h"
#include "SharedProtocol.h"
#include "QueueProtocol.h"

phantom_mutex *Mutex0;
phantom_signal *Signal0;
int world_rank, world_size;

void CA();
void CB();
```

The example above shows the main component that is executed on a single node. More complex situations can arise, where the implementation can include a combination of the two approaches, deploying the components on both MPI processes and separate threads, optimizing deployment depending on the design of the application as well as the requirements that where defined for its execution. In this case, multiple executables will be created – one for each different node in the deployment plan, each including a different set of components that were selected by the MOM to run on the specified node.

6.3.4 Executable generation and deployment

The compilation scripts that are generated prepare the local directories of the hardware for deployment. The executables or the source files are downloaded, compiled and placed on the corresponding locations of the target machines. If new versions of the executables are generated, they are uploaded on the Repository for future reuse. The Makefile, residing in the 'src' directory on the Repository, is also modified to include the corresponding

compiler wrappers (nvcc, mpicc …), necessary libraries (OpenMP, pthreads) and the modified component source files.

The deployment script is responsible for the actual execution of the components on the corresponding locations. This script is designed according to the mapping that is described in the Deployment Plan and the Platform Description that provides the network location of each machine or device. The machines are accessed via SSH by the development machine, where the Deployment Manager is executed. Thus, the necessary configurations on the target machines are assumed for the successful deployment of the application. Examples of the compilation/deployment scripts can be found in the Appendix section.

OpenMPI provides a list of options that enable affinity configurations at execution. By binding specific resources to specific processes, the deployment suggested by the Multi-Objective Mapper is implemented, while additional locality optimizations allow further improvements in the exploitation of the memory resources.

Some examples of these options are noted here:

To map processes:

--map-by <foo>
Map to the specified object, defaults to *socket*. Supported options include slot, hwthread, core, L1cache, L2cache, L3cache, socket, numa, board, node, sequential, distance, and ppr. Any object can include modifiers by adding a : and any combination of PE=n (bind n processing elements to each proc), SPAN (load balance the processes across the allocation), OVERSUBSCRIBE (allow more processes on a node than processing elements), and NOOVERSUBSCRIBE. This includes PPR, where the pattern would be terminated by another colon to separate it from the modifiers.

-nolocal, --nolocal
Do not run any copies of the launched application on the same node as orterun is running. This option will override listing the localhost with **--host** or any other host-specifying mechanism.

-nooversubscribe, --nooversubscribe
Do not oversubscribe any nodes; error (without starting any processes) if the requested number of processes would cause oversubscription. This option implicitly sets "max_slots" equal to the "slots" value for each node.

-bynode, --bynode
Launch processes one per node, cycling by node in a round-robin fashion. This spreads processes evenly among nodes and assigns MPI_COMM_WORLD ranks in a round-robin, "by node" manner.

To order processes' ranks in MPI_COMM_WORLD:

--rank-by <foo>

Rank in round-robin fashion according to the specified object, defaults to *slot*. Supported options include slot, hwthread, core, L1cache, L2cache, L3cache, socket, numa, board, and node.

For process binding:

--bind-to <foo>
Bind processes to the specified object, defaults to *core*. Supported options include slot, hwthread, core, l1cache, l2cache, l3cache, socket, numa, board, and none.

7. CONCLUSIONS

The PHANTOM middleware, developed during WP4, offers a variety of tools that allow smooth setup and also ensure availability and seamless operation of the heterogeneous infrastructure. Whereas the traditional CPU- and GPU-based resources are relatively well established, there are still issues presented in ensuring their consolidated functioning, such as the fine-grain management at the level of individual compute units – cores. Furthermore, the FPGA accelerators constitute the major challenge for the operation of the infrastructures that incorporates them. The challenges also include the proper monitoring, resource management, security enforcement, deployment of applications, etc.

All tools have been developed by the PHANTOM consortium to address the users' requirements. The final evaluation has revealed, that requirements are met. All tools are in operation and used successfully by all three demonstration use cases.

The future directions of work concentrate on the dissemination among the potential user groups, including academic, industrial and SME representatives. In particular, the roadmap for individual tools includes:

- Monitoring Client

  The Monitoring Client as well as the Monitoring Server and the MF-Library are available on open source at GitHub. Its capabilities in addition to its low impact on the monitored systems and its low requirements on resources, makes it suitable for a wide range of possible uses and applications, from embedded devices to large computing systems. Automated installation scripts, and examples of use uploaded on the GitHub will easy its future use.

- Resource Manager

  The PHANTOM Resource manager open source code is available on the same GitHub project folder among other PHANTOM tools as the Monitoring Framework. There can be found contains examples of use, scripts as well as a web-interface for simplify its installation and use. The Resource manager can support future hardware architectures, characteristics and monitoring configurations without need be modified. It is because the information stored on it is based on flexible JSON documents.

- FPGA Linux Distribution and FPGA Infrastructure

  As a support framework, the Linux distribution is primarily useful to aim dissemination of the programming model. In addition to this, it is already deployed within the University to assist with easy access to FPGA development boards for both teaching and research, but the aim is to extend its functionality to handle a wider array of target boards. Another area of interest is to integrate support for newer networking standards such as 10Gbps and 40Gbps. These are supported by high-end FPGA boards, but due to their incredibly challenging domain requirements are currently inaccessible to all but the most experienced

hardware designers. This could allow the platform to support FPGA-based networking, software-defined networks, and other growth areas.

- IP-Cores Marketplace and Generator

  The IP Core Marketplace can be assessed as a part of the PHANTOM Repository specific for storing IP Cores that were manually optimised, to have the best performance and optimal resource utilization, by a specialized hardware engineer and will serve as accelerators, for commonly used mathematical algorithms or image processing filters, implemented specifically for FPGA reconfigurable hardware. For other application specific algorithms, the IP Core Generator can be used and allows the user to run part of their application in FPGA hardware, taking care of transforming the user source code and creating the appropriate interfaces between software and hardware, without any effort from the developer. These two modules together with the rest of the PHANTOM platform deliver a certain degree of flexibility to the user for exploiting FPGA hardware with almost zero effort and without the need for specialized knowledge in reconfigurable hardware.

- Deployment Manager

  The Deployment Manager will be part of the WINGS multi-vertical platform - along with the Programming Interface – and will be used to orchestrate cloud management low-level services, optimizing their deployment on available infrastructures. To this end, the deployment strategies used by the tool will be further developed and integrated extending their support for the needs of the platform.

8. REFERENCES

[1] DreamCloud EU project website – http://www.dreamcloud-project.org/

[2] INCITS 499-2013, Information technology – Next Generation Access Control – Functional Architecture, InterNational Committee for Information Technology Standards, Cyber Security technical committee 1, 2013.

[3] "NVIDIA Management Library (NVML).," NVIDIA, [Online]. Available: https://developer.nvidia.com/nvidia-management-library-nvml. [Accessed 10 2018].

APPENDIX 1. FINAL LIST OF REQUIREMENTS AND THEIR FULFILMENT – MONITORING CLIENT

The requirements consist of two sets:

- The initial set of requirements (indexed in the following as Uxx), imposed by the use case providers at the beginning of the project and specified in Deliverable D1.1. The requirements in the Monitoring Framework can be related to one or more components of the MF architecture.

- Additional requirements that were obtained from the feedback of use case providers (indexed in the following as Nx), done after the release of a preliminary version of PHANTOM tools by M18, as introduced in D4.3.

Use Case requirements

Heterogeneous target platform (U73)

The PHANTOM monitoring framework should support all mandatory target platforms of the users' interest. To be specific, the PHANTOM infrastructure should be heterogeneous, including multi-core CPUs, GPUs, FPGAs, and the targeted embedded systems (e.g. Movidius). Subject to the hardware facilities and availabilities, sometimes a hosting hardware is required in order to monitor the connected accelerator. For instance, collecting the run-time metrics of a GPU is done by the monitoring framework deployed on the associated hosting CPU. The reconfigurable (FPGA) and hybrid (CPU+FPGA) device monitoring should happen via industry-standard FPGA Mezzanine Connectors (FMC), e.g. Xilinx Zynq platform.

Table 2. Heterogeneous target platform use case requirements addressed by MF-Client

Req. No.	Requirement	Priority	Status Achieved
U73	The run-time monitor shall be capable of acquiring monitoring data in all mandatory target platforms (e.g. CPU, FPGA, etc) subject to available hardware capabilities	SHALL	yes

- **Metrics (U72, U74-78)**

Generally, the PHANTOM run-time monitoring should support metrics covering both hardware (infrastructure-level) and software (application-level) properties. Some metrics are predefined, like the execution time, memory properties, power consumption, communication bandwidth, and I/O usage, while the others are application-specific metrics, which are user-defined and application-distinctive. Some examples of the user-defined metrics are the number of processed frames for the surveillance use case or the number of the numerical integration steps for the HPC application. These predefined

metrics are also different based on different hardware and sensors availabilities. Appendix 3. "List of Monitoring Metrics" lists the predefined metrics for the major platforms (cf. D4.2) according to the users' requirements and hardware availabilities. As Linux constitutes the major OS of the targeted hardware architectures (cf. D4.2), its monitoring services are largely leveraged by the PHANTOM monitoring services.

Table 3. Metrics use case requirements addressed by MF-Client

Req. No.	Requirement	Priority	Status Achieved
U72	The PHANTOM run-time monitor shall be able to monitor non-functional properties of an application	SHALL	YES
U74	The PHANTOM framework should be capable of monitoring execution time properties	SHOULD	YES
U75	The PHANTOM framework should be capable of monitoring memory properties	SHOULD	YES
U76	The PHANTOM framework should be capable of monitoring power consumption properties	SHOULD	YES
U77	The PHANTOM framework should be capable of monitoring communications bandwidth properties	SHOULD	YES
U78	The PHANTOM framework should be capable of monitoring I/O properties	SHOULD	YES

- **Accessibility (U35, U82)**

The run-time monitoring accessibility requirements are mainly the following:

- Data obtained by the run-time monitoring framework shall be accessible to the users by some means based on the users' interest.

- Data obtained by the users should be structured in a standard format, which enables further integration requirements.

- Users should be able to control the metrics sampling frequency and to select which metrics are to be monitored.

- Data storage and historical metrics analysis shall also be provided in the monitoring framework.

Table 4. Accessibility use case requirements addressed by MF-Client

Req. No.	Requirement	Overall Priority	Status
U35	The PHANTOM framework shall be capable of interfacing with local target platforms (deploying the application, monitoring the	SHALL	YES

	execution and the state of the target platform resources).		
U82	For non-periodic non-functional properties, the user should be able to select the frequency of data acquisition	SHOULD	YES

Feedback from M18 results

The requirement found from the questionnaires filled by the Use Case Partners (D1.3), appears in the next table.

Table 5. Additional use case requirements addressed by Monitoring Client

Req. No.	Requirement	Overall Priority	Status
N5	Strategies to minimise memory space required for gathering data and network bandwidth to send data to MF-server.	MAY	YES

The requirement is solved as the MF-Client is a lightweight service. The evaluation results published in D4.2 show a CPU overhead of less than 0.1% and memory consumption of less than 4.5 MB, thus making the client an affordable option even for embedded devices.

The users can reduce the memory space required for gathering data by reducing the time interval of the MF-Client for transferring data. On the other hand. The use of the network can be eliminated during the execution of a particular application by increasing the time interval, which will require storing the collected metrics in local memory.

A solution for reducing the network bandwidth and reduce the required memory space simultaneously consists of reducing the frequency of sampling metrics, which can be defined by the users.

APPENDIX 2. FINAL LIST OF REQUIREMENTS AND THEIR FULFILMENT – RESOURCE MANAGER

- **Integration with tools/processes (U35)**

Table 6. Integration use case requirements addressed by the Resource Manager

Req. No.	Requirement	Overall Priority	Status
U35	The PHANTOM framework shall be capable of interfacing with local target platforms (deploying the application, monitoring the execution and the **state of the target platform resources).**	SHALL	Support provided

- **Multi-dimensional optimization (U48)**

Table 7. Multi-Dimensional Optimization use case requirements addressed by the Resource Manager

Req. No.	Requirement	Overall Priority	Status
U48	The PHANTOM framework shall propose a mapping of the parallel code blocks, composing the application, to the **available resources/target platforms**	SHALL	Support provided

- **System and data security (U65, U67)**

Table 8. System and Data Security use case requirements addressed by the Resource Manager

Req. No.	Requirement	Overall Priority	Status
U65	Data obtained through the run-time monitoring (stored in the MF-Server) should be able to be secured against eavesdropping / unwanted access → **System status from the RM which is an independent server from MF-Server allows to reduce access to the MF-Server, even place them in a different network, which can help to increase the security on the MF-Server**	SHOULD	YES
U67	PHANTOM should be able to support HPC/Cloud security mechanisms to protect data and control data access	SHOULD	Support provided

- **Dependability (U71)**

Table 9. Dependability requirements addressed by the Resource Manager

Req. No.	Requirement	Overall Priority	Status
U71	PHANTOM may support survivability mechanisms to detect and recover from faults → **RM can provide an alarm when a processing node stop updating their status.**	MAY	Support provided

- **Run-time monitoring (U78, U82)**

Table 10. Run-Time monitoring use case requirements addressed by Monitoring Client

Req. No.	Requirement	Priority	Status
U79	The **data obtained** by the run-time monitor shall be **accessible** and exposed to the user for their own tasks	SHALL	Support provided
U82	For non-periodic non-functional properties, the user should be able to **select the frequency of data acquisition**	SHOULD	Support provided

APPENDIX 3. FINAL LIST OF REQUIREMENTS AND THEIR FULFILMENT – FPGA LINUX DISTRIBUTION AND FPGA INFRASTRUCTURE

Use Case requirements

Given its place as low-level infrastructure, distribution and associated tooling affects a wide range of requirements. The following table covers only the major requirements specifically targeted by this work.

Table 11. Use case requirements addressed by Monitoring Client

Req. No.	Requirement	Priority	Status
U11	The PHANTOM framework shall support FPGAs within target platforms	SHALL	Achieved.
U15	The PHANTOM framework should exploit hardware accelerators present in the target hardware platforms	SHOULD	Achieved.
U18	The PHANTOM framework shall hide the target platform heterogeneity, abstracting the underlying platform technologies' details from the user, provided the user is not trying to exploit specific platform capabilities.	SHALL	Achieved from the perspective of software. Hardware heterogeneity is covered by the FPGA work in the Parallelisation Toolkit.
U64	Remote target platforms should be able to be secured against eavesdropping through interfaces with external infrastructures for trust/authentication	SHOULD	Achieved.
U66	PHANTOM should support means for tasks isolation and information flow control policy	SHOULD	Achieved.
U68	PHANTOM shall be able to guarantee data integrity when applications are mapped onto heterogeneous targets	SHOULD	Achieved.
U73	The run-time monitor shall be capable of acquiring monitoring data in all	SHALL	Achieved.

Req. No.	Requirement	Priority	Status
	mandatory target platforms (e.g. CPU, FPGA, etc) subject to available hardware capabilities		

Feedback from M18 results

The M18 results highlighted two areas of required improvement:

1) 64-bit support was lacking. 64-bit IP cores were supported, but only 32-bit CPUs were targeted which meant that they could not be used efficiently.

2) The security mechanisms are implemented using memory space partitioning. Previously this partitioning was of a fixed size, which restricted the flexibility of user IP cores.

These issues have been addressed in this release.

APPENDIX 4. FINAL LIST OF REQUIREMENTS AND THEIR FULFILMENT – FPGA MARKETPLACE AND FPGA GENERATOR

Use case requirements

Table 12. Use case requirements addressed by the IP Core Marketplace and/or IP Core Generator

Req. No.	Requirement	Priority	Status
U11	The PHANTOM framework shall support FPGAs within target platforms	SHALL	**Achieved**

The support for FPGAs is provided by the IP Core Marketplace, IP Core Generator and the PHANTOM FPGA Linux Platform, that integrates the IP cores from the two previously mentioned modules into functional FPGA architectures that are in accordance with the interfaces defined in PHANTOM and can be exploited by the PHANTOM Platform.

Req. No.	Requirement	Priority	Status
U12	The PHANTOM framework may support DSPs within target platforms	MAY	**Achieved**

Both IP cores in the Marketplace and those produced by the IP Core Generator can use the DSPs present in the FPGAs. Vivado HLS tools will decide, based on the source code, whether it is beneficial or not to use the DSPs in the generated IP cores.

Req. No.	Requirement	Priority	Status
U15	The PHANTOM framework should exploit hardware accelerators present in the target hardware platforms	SHOULD	**Achieved**

PHANTOM can take advantage of FPGAs available in the target platform as well as existing DSPs in the FPGAs.

APPENDIX 5. FINAL LIST OF REQUIREMENTS AND THEIR FULFILMENT - DEPLOYMENT MANAGER

A lot of the use case requirements were requested without the inclusion of the Deployment Manager in the PHANTOM architecture. Its design was specifically done with the purpose of satisfying a lot of the needs that are concluded by these requirements.

The following requirements can be split into two main categories:

- Integration of the components, including the communication between them
- Compilation and deployment of the components on the target machines

Req. No.	Requirement	Priority	Status
U1	PHANTOM shall have a defined execution, memory and communications model	SHALL	**Achieved**
U2	PHANTOM shall support uniform and non-uniform memory access models	SHALL	**Achieved**
U5	The PHANTOM framework shall support multi-threaded concurrent tasks, including communication and synchronization	SHALL	**Achieved**
U8	PHANTOM shall support component-based application design	SHALL	**Achieved**
U9	The PHANTOM framework shall support multi-core CPUs within target platforms	SHALL	**Achieved**
U10	The PHANTOM framework shall support GPUs within target platforms	SHALL	**Achieved**
U12	The PHANTOM framework may support DSPs within target platforms	MAY	**Not Achieved**
U13	The PHANTOM framework shall support computing clusters as target platforms	SHALL	**Achieved**
U16	The user shall be able to configure the target platform for a given project, which is composed by a set of supported target platforms	SHALL	**Achieved**
U18	The PHANTOM framework shall hide the target platform heterogeneity, abstracting the underlying platform technologies' details from the user, provided the user is not trying to exploit specific platform capabilities	SHALL	**Achieved**
U20	The PHANTOM framework shall provide constructs or abstractions to deal with non-uniform and uniform memory, hiding the underlying data transfer details	SHALL	**Achieved**
U21	The PHANTOM framework shall automate the process of transferring data to/from different memories according to the	SHALL	**Achieved**

Req. No.	Requirement	Priority	Status
	component data model		
U32	The PHANTOM framework shall accept application source code developed in C.	SHALL	**Achieved**
U33	The PHANTOM framework may support higher level language such as Java and C++	MAY	**Achieved**
U34	The PHANTOM development framework shall execute locally on a Linux workstation	SHALL	**Achieved**
U35	The PHANTOM framework shall be capable of interfacing with local target platforms (deploying the application, monitoring the execution and the state of the target platform resources).	SHALL	**Achieved**
U36	The PHANTOM framework should be capable of interfacing with remote target platforms	SHOULD	**Achieved**
U39	The PHANTOM framework shall be able to automatically compile or synthesize the parallelized source code for any supported target platform	SHALL	**Achieved**
U40	The user should be able to configure/customize the compilation/synthesis options used during the compilation/synthesis process	SHOULD	**Achieved**
U41	The PHANTOM framework shall be able to automatically deploy the binary code over the target platform	SHALL	**Achieved**
U42	The automatic compilation/synthesis and deployment processes should be optional, as the user might want to do this manually or externally	SHOULD	**Achieved**
U43	The deployment process may be encapsulated under a separate component with a well specified interface, as to decrease the effort of adding new deployments means or new target platforms	MAY	**Achieved**
U44	The compilation process may be encapsulated under a separate component with a well specified interface, as to decrease the effort of adding new compilation tool chains or new target platforms	MAY	**Achieved**
U47	PHANTOM shall support Telecom specific application classes where domain-specific libraries are commonly utilized	SHALL	**Achieved**
U69	The PHANTOM development framework shall be able to execute on a typical "mid-end"	SHALL	**Achieved**

Req. No.	Requirement	Priority	Status
	workstation (Core I5, 4GB RAM)		
U86	PHANTOM may support application specific communication bus/protocols	MAY	**Not Achieved**

Due to the fact that the Deployment Manager was not considered in the DoA and was introduced later in development than the rest of the PHANTOM tools, there was not sufficient feedback from M18 results to evaluate its performance. However, there were some issues that were identified during the development procedure, which are introduced in the table below:

Req. No.	Requirement	Priority	Status
AU1	Compilation process should be optimized in such a way that unnecessary binary generation will be avoided minimizing compilation time	SHOULD	**Achieved**
AU2	Input/output file transfers should be reduced to a minimum to avoid network congestion due to excessively large data files	SHOULD	**Achieved**
AU3	Input files shall be checked for validity before deployment, ensuring the uncorrupted status of the data	SHALL	**Achieved**
AU4	Transfers of both static and dynamic-sized data between components should be supported	SHOULD	**Achieved**

APPENDIX 6. PROGRAMMING INTERFACE IMPLEMENTATION

In this appendix, the final implementation of the Programming Interface is displayed:

```
QUEUE PROTOCOL

void *phantom_queue_get(phantom_queue *queue) {
        void *item;
        int32_t size;
        void *tmp = NULL;
        if(queue->caller == queue->owner) {
                while(tmp == NULL) {
                        phantom_dequeue(queue,&tmp,&size);
                }
                item = malloc(size+sizeof(uint32_t));

memcpy(item,uint32_t_to_uint8_t(size),sizeof(uint32_t));
                memcpy(item+sizeof(uint32_t),tmp,size);
        }
        else {
                srand(time(0)+seed);
                seed++;
                int r = rand();
                int tag = (((r & 0x0003FFFF) << 13 ) | (0xFF & queue-
>ID));
                printf("QUEUE GET WITH TAG: %d = ",tag);
                print_binary(tag);
                int rtag = tag | 0x1FFF;
                MPI_Status status;
                MPI_Send(&rtag,1,MPI_INT,queue-
>owner,tag,MPI_COMM_WORLD);
                MPI_Probe(queue->owner,rtag,MPI_COMM_WORLD,&status);
                int count;
                MPI_Get_count(&status,MPI_CHAR,&count);
                item = malloc(count);
                MPI_Recv(item,count,MPI_CHAR,queue-
>owner,rtag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        }
        return item;
}

bool phantom_queue_put(phantom_queue *queue,void *item) {
        if(queue->caller == queue->owner)

phantom_enqueue(queue,item+sizeof(uint32_t),uint8_t_to_uint32_t(item))
;
        else {
                srand(time(0)+seed);
                seed++;
                int r = rand();
                int tag = ((((r & 0x0003FFFF) << 13) | 0x100) | (0xFF
& queue->ID));
                printf("QUEUE PUT WITH TAG: %d = ",tag);
                print_binary(tag);
                MPI_Send(item,uint8_t_to_uint32_t((uint8_t
*)item)+sizeof(uint32_t),MPI_CHAR,queue->owner,tag,MPI_COMM_WORLD);
                char dum=0;
```

```
                int rtag = tag | 0x1FFF;
                MPI_Recv(&dum,1,MPI_CHAR,queue-
>owner,rtag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        }
        return true;
}

void *phantom_queue_peek(phantom_queue *queue) {
        void *item;
        if(queue->caller == queue->owner) {
                int32_t size;
                void *tmp;
                phantom_peek_at_queue(queue,tmp,&size);
                item = malloc(size);

memcpy(item,uint32_t_to_uint8_t(size),sizeof(uint32_t));
                memcpy(item+sizeof(uint32_t),tmp,size);
        }
        else {
                srand(time(0)+seed);
                seed++;
                int r = rand();
                int tag = ((((r & 0x0003FFFF) << 13) | 0x200) | (0xFF
& queue->ID));
                printf("QUEUE PEEK WITH TAG: %d = ",tag);
                print_binary(tag);
                int rtag = tag | 0x1FFF;
                MPI_Status status;
                MPI_Send(&rtag,1,MPI_INT,queue-
>owner,tag,MPI_COMM_WORLD);
                MPI_Probe(queue->owner,rtag,MPI_COMM_WORLD,&status);
                int count;
                MPI_Get_count(&status,MPI_CHAR,&count);
                item = malloc(count);
                MPI_Recv(item,count,MPI_CHAR,queue-
>owner,rtag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        }
        return item;
}

uint32_t phantom_queue_count(phantom_queue *queue) {
        uint32_t count;
        if(queue->caller == queue->owner) {
                pthread_mutex_lock(&queue->lock);
                count = queue->count;
                pthread_mutex_unlock(&queue->lock);
        }
        else {
                srand(time(0)+seed);
                seed++;
                int r = rand();
                int tag = ((((r & 0x0003FFFF) << 13) | 0x300) | (0xFF
& queue->ID));
                printf("QUEUE COUNT WITH TAG: %d = ",tag);
                print_binary(tag);
                int rtag = tag | 0x1FFF;
                MPI_Status status;
                MPI_Send(&rtag,1,MPI_INT,queue-
```

```
>owner,tag,MPI_COMM_WORLD);
                MPI_Recv(&count, 1, MPI_INT,queue->owner, rtag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        return count;
}
```

SHARED PROTOCOL

```
void phantom_synchronize(phantom_shared *item, void *local_data, int
dir) {
        if(dir == 0) {
                if(item->caller != item->owner) {
                        srand(time(0)+seed);
                        seed++;
                        int r = rand();
                        int tag = ((((r & 0x0003FFFF) << 13 ) | 0x400)
| (0xFF & item->ID));
                        printf("SYNC0 WITH TAG: %d = ",tag);
                        print_binary(tag);
                        int rtag = tag | 0x1FFF;
                        int info[2];
                        info[0] = item->offset;
                        info[1] = item->size;
                        MPI_Send(info,2,MPI_INT,item-
>owner,tag,MPI_COMM_WORLD);
                        MPI_Recv(local_data,item->size,item-
>type,item->owner,rtag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
                }
                else {
                        memcpy(local_data,item->data+item-
>offset,item->size);
                }
        }
        else if(dir == 1) {
                if(item->caller != item->owner) {
                        srand(time(0)+seed);
                        seed++;
                        int r = rand();
                        int tag = ((((r & 0x0003FFFF) << 13 ) | 0x500)
| (0xFF & item->ID));
                        printf("SYNC1 WITH TAG: %d = ",tag);
                        print_binary(tag);
                        int info[2];
                        info[0] = item->offset;
                        info[1] = item->size;
                        printf("Sending OFFSET: %d, SIZE: %d\n",item-
>offset,item->size);
                        MPI_Send(info, 2, MPI_INT, item->owner, tag,
MPI_COMM_WORLD);
                        MPI_Send(local_data, item->size, item->type,
item->owner, tag, MPI_COMM_WORLD);
                }
                else {
                        memcpy(item->data+item->offset, local_data,
item->size);
                }
        }
        else {
```

```
                fprintf(stderr,"Invalid direction value!
Terminating...\n");
                exit(1);
        }
}
```

SIGNAL PROTOCOL

```
int phantom_notify(phantom_signal *signal) {
        if(signal->caller == signal->owner) {
                pthread_mutex_lock(&signal->lock);
                if(pthread_cond_signal(&signal->cond) != 0) {
                        fprintf(stderr,"Failed to send signal\n");
                        exit(0);
                }
                signal->value++;
                pthread_mutex_unlock(&signal->lock);
        }
        else {
                srand(time(0)+seed);
                seed++;
                int r = rand();
                int tag = ((((r & 0x0003FFFF) << 13 ) | 0x600) | (0xFF
& signal->ID));
                printf("SIGNAL NOTIFY WITH TAG: %d = ",tag);
                print_binary(tag);
                char dum=0;
                MPI_Send(&dum,1,MPI_CHAR,signal->owner,tag,signal-
>comm);
        }
        return 0;
}

int phantom_wait(phantom_signal *signal) {
        if(signal->caller == signal->owner) {
                pthread_mutex_lock(&signal->lock);
                while(signal->value == 0) {
                        if(pthread_cond_wait(&signal->cond,&signal-
>lock) != 0) {
                                fprintf(stderr, "Failed to wait the
condition variable\n");
                                exit(0);
                        }
                }
                signal->value--;
                pthread_mutex_unlock(&signal->lock);
                return 0;
        }
        else {
                MPI_Status status;
                srand(time(0)+seed);
                seed++;
                int r = rand();
                int tag = ((((r & 0x0003FFFF) << 13 ) | 0x700) | (0xFF
& signal->ID));
                printf("SIGNAL WAIT WITH TAG: %d = ",tag);
                print_binary(tag);
                char dum=0;
                MPI_Send(&dum,1,MPI_CHAR,signal->owner,tag,signal-
```

```
>comm);
                int rtag = tag | 0x1FFF;
                MPI_Recv(&dum,1,MPI_CHAR,signal->owner,rtag,signal-
>comm,&status);
                return status.MPI_ERROR;
        }
}

int phantom_notifyall(phantom_signal *signal) {
        if(signal->caller == signal->owner) {
                pthread_mutex_lock(&signal->lock);
                if(pthread_cond_broadcast(&signal->cond) != 0) {
                        fprintf(stderr,"Failed to broadcast\n");
                        exit(0);
                }
                signal->value += (signal->num-1);
                pthread_mutex_unlock(&signal->lock);
        }
        else {
                srand(time(0)+seed);
                seed++;
                int r = rand();
                int tag = ((((r & 0x0003FFFF) << 13 ) | 0x800) | (0xFF
& signal->ID));
                printf("SIGNAL BROADCAST WITH TAG: %d = ",tag);
                print_binary(tag);
                char dum=0;
                MPI_Send(&dum,1,MPI_CHAR,signal->owner,tag,signal-
>comm);
        }
        return 0;
}

void phantom_barrier(phantom_signal *signal) {
        if(signal->caller == signal->owner) {
                pthread_mutex_lock(&signal->lock);
                if(signal->value < signal->num-1) {
                        signal->value++;
                        if(pthread_cond_wait(&signal->cond,&signal-
>lock) != 0) {
                                fprintf(stderr,"Failed to wait at
barrier\n");
                                exit(0);
                        }
                }
                else {
                        if(pthread_cond_broadcast(&signal->cond) != 0)
{
                                fprintf(stderr,"Failed to
broadcast\n");
                                exit(0);
                        }
                        signal->value = 0;
                }
                pthread_mutex_unlock(&signal->lock);
        }
        else {
                srand(time(0)+seed);
```

```
                seed++;
                int r = rand();
                int tag = ((((r & 0x0003FFFF) << 13 ) | 0x900) | (0xFF
& signal->ID));
                printf("SIGNAL BROADCAST WITH TAG: %d = ",tag);
                print_binary(tag);
                char dum=0;
                MPI_Send(&dum,1,MPI_CHAR,signal->owner,tag,signal-
>comm);
        }
        return;
}
```

MUTEX PROTOCOL

```
int phantom_lock(phantom_mutex *mutex) {
        if(mutex->caller == mutex->owner) {
                pthread_mutex_lock(&mutex->lock);
        }
        else {
                srand(time(0)+seed);
                seed++;
                int r = rand();
                int tag = ((((r & 0x0003FFFF) << 13 ) | 0xA00) | (0xFF
& mutex->ID));
                printf("MUTEX LOCK WITH TAG: %d = ",tag);
                print_binary(tag);
                char dum=0;
                MPI_Send(&dum,1,MPI_CHAR,mutex->owner,tag,mutex-
>comm);
        }
        return 0;
}

int phantom_unlock(phantom_mutex *mutex) {
        if(mutex->caller == mutex->owner) {
                pthread_mutex_unlock(&mutex->lock);
        }
        else {
                MPI_Status status;
                srand(time(0)+seed);
                seed++;
                int r = rand();
                int tag = ((((r & 0x0003FFFF) << 13 ) | 0xB00) | (0xFF
& mutex->ID));
                printf("MUTEX UNLOCK WITH TAG: %d = ",tag);
                print_binary(tag);
                char dum=0;
                MPI_Send(&dum,1,MPI_CHAR,mutex->owner,tag,mutex-
>comm);
        }
        return 0;
}

int phantom_trylock(phantom_mutex *mutex) {
        if(mutex->caller == mutex->owner) {
                pthread_mutex_trylock(&mutex->lock);
        }
        else {
```

```
                srand(time(0)+seed);
                seed++;
                int r = rand();
                int tag = ((((r & 0x0003FFFF) << 13 ) | 0xC00) | (0xFF
& mutex->ID));
                printf("MUTEX TRYLOCK WITH TAG: %d = ",tag);
                print_binary(tag);
                char dum=0;
                MPI_Send(&dum,1,MPI_CHAR,mutex->owner,tag,mutex-
>comm);
        }
        return 0;
}
```

According to the latest architecture of the application integrated by the Deployment Manager, there is a listener thread for every process that takes up to implement any incoming requests from external processes/devices. The implementation of the listener is also displayed here:

```
LISTENER FOR INCOMING REQUESTS

void *object_handler(void *st) {
        char dum=0;
        MPI_Status status = *((MPI_Status *)st);
        int tag = status.MPI TAG;
        int ID = tag & 0xFF;
        int method = (tag >> 8) & 0x0F;
        int source = status.MPI_SOURCE;
        int done = (tag >> 12) & 0x01;
        int count;
        MPI_Get_count(&status,MPI_CHAR,&count);

        if(done == 1) {
                MPI_Recv(&dum, 1, MPI_CHAR, source, tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                // request
                if(world_rank == 0) {
                        __sync_fetch_and_add(&process_join,1);
                }
                // response
                else {
                        app_running = false;
                }
                return NULL;
        }
        if(object_protocols[ID] == SHARED) {
                // Sharing local data
                if(method == 4) {
                        int rtag = tag | 0x1FFF;
                        int info[2];
                        MPI_Recv(info, 2, MPI_INT, source, tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        MPI_Send(shared[ID]->data+info[0], info[1],
shared[ID]->type, source, rtag, MPI_COMM_WORLD);
                }
                // Obtaining external data
                if(method == 5) {
```

```
                        int info[2];
                        MPI_Recv(info, 2, MPI_INT, source, tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        MPI_Recv(shared[ID]->data+info[0], info[1],
shared[ID]->type, source, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                }
        }
        else if(object_protocols[ID] == QUEUE) {
                // External item pull
                if(method == 0) {
                        int rtag;

MPI_Recv(&rtag,1,MPI_INT,source,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
                        void *item;
                        uint32_t size;
                        item = phantom_queue_get(queue[ID]);
                        size = uint8_t_to_uint32_t((uint8_t
*)item)+sizeof(uint32_t);
                        MPI_Send(item, size, MPI_CHAR, source, rtag,
MPI_COMM_WORLD);
                }
                // External item push
                else if(method == 1) {
                        void *buf;
                        buf = malloc(count);
                        MPI_Recv(buf, count, MPI_CHAR, source, tag,
MPI COMM WORLD, MPI STATUS IGNORE);
                        phantom_queue_put(queue[ID],buf);
                        int rtag = tag | 0x1FFF;

MPI_Send(&dum,1,MPI_CHAR,source,rtag,MPI_COMM_WORLD);
                }
                // Peeking at queue
                else if(method == 2) {
                        int rtag;
                        MPI_Recv(&rtag, 1, MPI_INT, source, tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        void *item;
                        uint32_t size;
                        item = phantom_queue_peek(queue[ID]);
                        size = uint8_t_to_uint32_t((uint8_t
*)item)+sizeof(uint32_t);
                        MPI_Send(item, size, MPI_CHAR, source, rtag,
MPI_COMM_WORLD);
                }
                // Getting queue count
                else if(method == 3) {
                        int rtag;

MPI_Recv(&rtag,1,MPI_INT,source,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
                        uint32_t count;
                        count = phantom_queue_count(queue[ID]);

MPI_Send(&count,1,MPI_INT,source,rtag,MPI_COMM_WORLD);
                }
        }
        else if(object_protocols[ID] == SIGNAL) {
                // External signal notify
```

```
                if(method == 6) {
                        phantom_notify(signal[ID]);

MPI_Recv(&dum,1,MPI_CHAR,source,tag,signal[ID]-
>comm,MPI_STATUS_IGNORE);
                }
                // External signal wait
                else if(method == 7) {

MPI_Recv(&dum,1,MPI_CHAR,source,tag,signal[ID]-
>comm,MPI_STATUS_IGNORE);
                        phantom_wait(signal[ID]);
                        int rtag = tag | 0x1FFF;

MPI_Send(&dum,1,MPI_CHAR,source,rtag,signal[ID]->comm);
                }
                // Notify all
                else if(method == 8) {
                        phantom_notifyall(signal[ID]);

MPI_Recv(&dum,1,MPI_CHAR,source,tag,signal[ID]-
>comm,MPI STATUS IGNORE);
                }
                // Barrier
                else if(method == 9) {
                        phantom_barrier(signal[ID]);

MPI_Recv(&dum,1,MPI_CHAR,source,tag,signal[ID]-
>comm,MPI_STATUS_IGNORE);
                }
        }
        else if(object_protocols[ID] == MUTEX) {
                // Mutex lock
                if(method == 10) {
                        phantom_lock(mutex[ID]);
                        MPI_Recv(&dum,1,MPI_CHAR,source,tag,mutex[ID]-
>comm,MPI_STATUS_IGNORE);
                }
                // Mutex unlock
                else if(method == 11) {
                        phantom_unlock(mutex[ID]);
                        MPI_Recv(&dum,1,MPI_CHAR,source,tag,mutex[ID]-
>comm,MPI_STATUS_IGNORE);
                }
                // Mutex trylock
                else if(method == 12) {
                        phantom_trylock(mutex[ID]);
                        MPI_Recv(&dum,1,MPI_CHAR,source,tag,mutex[ID]-
>comm,MPI_STATUS_IGNORE);
                }
        }
        return NULL;
}
```