PHANTOM

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

Project Number 688146

D4.3 – Enhanced release of integrated monitoring platform, infrastructure integration and resource management software stack

Version 1.1
19 May 2018
Final

Public Distribution

University of Stuttgart, University of York, Unparallel Innovation

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 Phone: +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	21/02/2018
0.2	Integrate HLRS/UI/YORK initial contributions	02/03/2018
1.0	Editing pass	04/03/2018
1.1	Added further information on development progress	19/05/2018

TABLE OF CONTENTS

EXECUTIVE SUMMARY

The purpose of this document is to describe the enhanced release of the system software stack of the PHANTOM platform. The components that are included in the release are the Monitoring Framework for heterogeneous hardware platforms and applications, the Resource Management service as well as a set of tools that enable the integration of reconfigurable hardware resources into a common infrastructure – the FPGA Linux system and the IP Cores Marketplace. The released tools constitute an important central part of the PHANTOM platform framework and are used to support the core platform components such as the Multi-Objective Mapper and the Deployment Manager.

Most notably, the release includes:

- **The PHANTOM Monitoring Client –** a tiny client software running on the device and collecting performance- and energy-specific metrics. The Client was designed in the EXCESS and DreamCloud project. PHANTOM makes significant extensions to support a separation of infrastructure- and application-specific metrics, simultaneous monitoring of heterogeneous hardware platforms, custom analytics and feedback, and the possibility to specify user-specific metrics.

- **The PHANTOM Monitoring Server –** a storage and database in which the metrics from all clients are aggregated and centrally analysed. As with the Client, the Server is an outcome of the DreamCloud project and had a substantial improvement in PHANTOM to improve the configurability, and extend the analytics functionality with regard to the gathered metrics values.

- **Resource management service –** a service that defines JSON schemata that describes the status of each hardware device of the PHANTOM architecture as well as providing general scripts to manage the resource status. The schema is hosted on the web-server and database of the Monitoring Server. This is a pure PHANTOM development.

- **The PHANTOM Linux system** which manages the software components mapped to the processors of a reconfigurable device. This is responsible for passing data to and from components and for coordinating their execution. The Linux system is partially based on the outcomes of the JUNIPER project but goes far beyond their core functionality, among others – the PHANTOM FPGA architecture is now included, which is responsible for entirely automatically hosting hardware components in supported FPGAs.

- **The PHANTOM IP Cores Marketplace** which contains several logic blocks specially designed to be used as application-specific accelerators. The IP cores implement specific, commonly used, mathematical algorithms such as Fast Fourier transform (FFT) and Finite impulse response (FIR), and image processing filters such as Sobel Filter, Discrete Wavelet Transform (DWT). The PHANTOM IP cores are a pure PHANTOM development. The cores are automatically integrated into the platform at the request of the MOM, by the PHANTOM Linux System.

The release documentation gives necessary details on the technological background and PHANTOM advances for all major tools and provides necessary installation and usage guides for them.

1. INTRODUCTION

1.1 RATIONALE

Modern software applications have to struggle with the great variety of available hardware platforms; ranging from commodity and low-power CPUs, to GPUs, reconfigurable-logic systems like FPGAs, or dedicated systems like Movidius' Myriad2. The selection of the most proper platform for the specific application, which has to fulfil the user-imposed functional and non-functional requirements, is a very challenging task. Moreover, the challenge is even more complex when this hardware is part of a common shared infrastructure.

The component-based PHANTOM programming model (see D1.2 and D2.1) allows the PHANTOM application to be deployed and executed on several hardware devices, which might be of different types. PHANTOM provides a platform that allows the components constituting the application to be executed in heterogeneous, parallel, and distributed hardware environments (see Figure 1) with all mapping and communications being automatically adapted to the changing platform without the requirement for developer input. PHANTOM also supports automatic parallelisation, to automate targeting of devices such as GPUs and FPGAs; this is discussed in deliverable D3.2.

Operation of such a heterogeneous infrastructure imposes several challenges. In particular:

- Monitoring of the infrastructure components status (e.g. utilization of the CPU and memory, cache misses, etc.)
- Management of the infrastructure components (availability for allocation of new application components, operational power mode, etc.)
- Availability of a run-time environment for the application execution (with the major challenge lying on reconfiguration-enabled hardware)

In order to solve these challenges, the PHANTOM platform requires middleware for monitoring and management, the integration into a common infrastructure of the heterogeneous hardware resources, and the tracking and managing of their statuses. The corresponding middleware has been developed in the frame of PHANTOM by work package WP4.

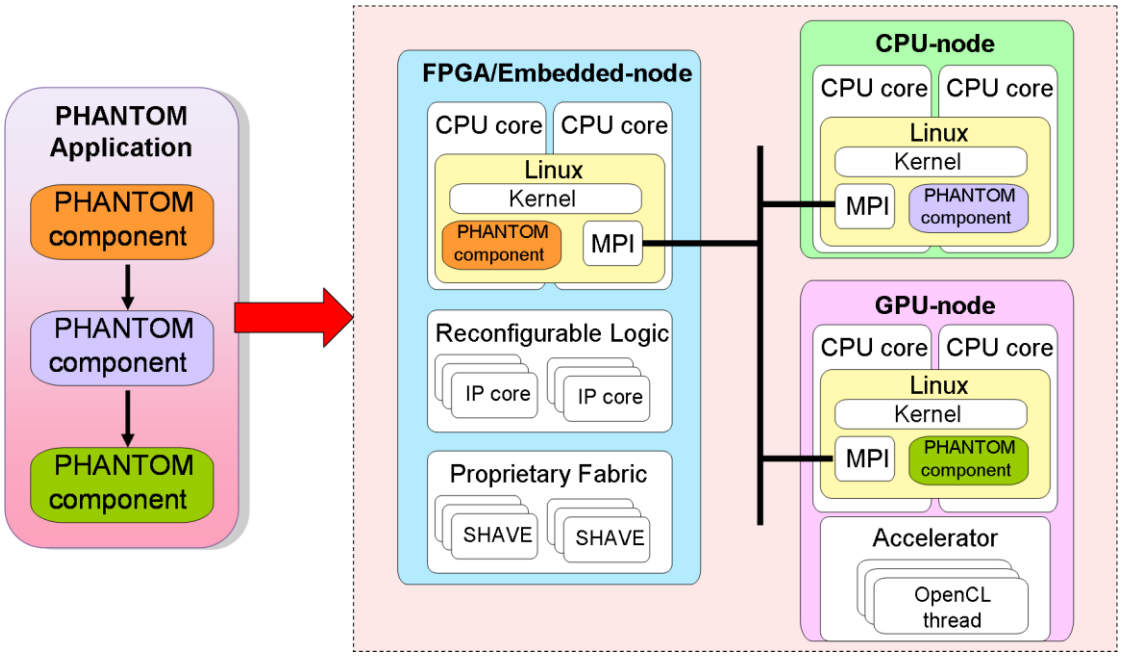


Figure 1: PHANTOM application execution in heterogeneous hardware environment

1.2 CHANGES FROM PREVIOUS RELEASE

Many of the improvements in this development phase are usability improvements driven by feedback from the application providers. Feedback has been taken and incorporated into the tools, and documentation has been added to.

More specifically, the monitoring framework has been extended to fully support user-defined metrics. The monitoring client is now also able to determine the subset of an overall resource utilisation caused by a specific application. Additionally, feedback from the Telecom use case has led to a reduction in the network traffic used by the monitoring infrastructure.

The Linux subsystem has focused on supporting a wider range of platforms, including more powerful 64-bit systems. The difficulties of cross-compilation has led to use of a Docker-based build system which makes installation simpler for the end user. Based on feedback, the size of the generated images can now be customised in order to support a wider range of smaller embedded platforms with limited storage.

1.3 RELEASE INFORMATION AND SCOPE

The current release is comprised of the following components:

- Monitoring and resource management framework

  The framework offers a set of tools and services for collection and centralized analysis of a broad scope of metrics that characterize the state of the hardware resources (utilization, overall energy consumption, etc.) as well as of the

applications (performance, application-specific energy consumption, etc.) over the time. The framework is used by the PHANTOM Multi-Objective Scheduler (MOM, for details please refer to deliverable D2.1) for elaborating the optimal resource allocation strategies for the applications as well as by the end-users of the applications in order to evaluate the performance characteristics and optimize the infrastructure utilization. The framework also hosts a resource management service that fosters the collection, publishing, and setting the status of the controlled infrastructure devices.

The following components are released as part of the monitoring and resource management framework:

- **Monitoring client –** a tiny client software running on the device and collecting performance- and energy-specific metrics
- **Monitoring server –** a storage and database in which the metrics from all clients are aggregated and centrally analysed
- **Resource management service –** a service, which is currently hosted by the monitoring server, that aims to track and control the status of all hardware devices included in the infrastructure

- Reconfigurable applications run-time environment

This environment provides technologies for executing applications on the hardware constituted by reconfigurable (FPGA) logic devices. The environment includes following components:

- **The PHANTOM Linux system** which manages the software components mapped to the processors of a reconfigurable device. This is responsible for passing data to and from components and for coordinating their execution. It also implements the process isolation and monitoring requirements of the platform.

  The system also includes **the PHANTOM FPGA architecture** which is responsible for hosting the hardware parts of components mapped to the FPGA. This architecture also implements further isolation and monitoring features. The architecture is automatically assembled and built based on the input from the PHANTOM Multi-Objective Mapper.

- **The PHANTOM IP Cores Marketplace** which contains several logic blocks specially designed to be used as application-specific accelerators. These IP cores will have specific functions, commonly used mathematical algorithms (e.g. Fast Fourier transform – FFT, Finite impulse response – FIR, etc.), and image processing filters (e.g. Sobel Filter, Discrete wavelet transform – DWT), etc.) implemented in the FPGA fabric.

The release includes tarballs with the source code and installation scripts as well as all necessary documentation and user guides. All released components are provided under open source licenses (see Table 1) and are available for free downloading on the PHANTOM's public hosting platform GitHub (https://github.com/phantom-platform).

Table 1: Released PHANTOM components

Component	Version	License	Link
Monitoring and Resource Management Framework	0.2	Apache 2.0	https://github.com/PHANTOM-Platform/Monitoring
PHANTOM Linux System	0.2	GPLv3	https://github.com/PHANTOM-Platform/PHANTOM-FPGA-Linux
PHANTOM IP core marketplace	0.2	Apache 2.0	https://github.com/PHANTOM-Platform/PHANTOM-IP-Core-Marketplace

2. PHANTOM HETEROGENEOUS PARALLEL INFRASTRUCTURE TESTBED

In order to evaluate the functionality of the PHANTOM system software components as well as to provide a heterogeneous parallel infrastructure testbed for the deployment of the developed PHANTOM application components, the PHANTOM project partners have granted access to hardware resources that are available on their sites and also that are required by the pilot applications (see D1.1). The resource access is only granted for the PHANTOM project consortium under the policies of the respective partners. Below an overview of the most typical resources is given.

2.1 STANDARD CPU-BASED DEVICES

The devices of this category constitute the most common general-purpose x86, AMD64 and ARM hardware.

2.1.1 Server-Solutions

USTUTT-HLRS provides access to its cluster testbed, used as a testbed for evaluating energy-efficiency of High Performance Computing applications.

The cluster (see Figure 2) consists of 3 compute nodes and a common networking file system, interconnected with the Infiniband network. Each node is built with NUMA (Non-Uniform Memory Access) technology with the characteristics summarized in Table 2.

Table 2: HPC cluster hardware specification

Amount	Hardware components (node01 and node02)
2	Intel Xeon CPU: E5-2690 v2 (Ivy Bridge) - 10 cores; 25MB L3 Smart Cache; 4 Mem. Ch. DDR3 1866 MHz = 59.7 GB/s
8	HP Memory 4 GB DDR3 MFG 708633-B21 - 1866 MHz PC3-14900
1	GPU: Tesla K40c - GPU Clock: 745 MHz; Shading Units: 2880; GDDR5 12288 MB; 288 GB/s; PCIe3.0 8GT/s x16
1	Hard disk: 500GB WD5003AZEX Black
1	SSD: 128GB Vertex450
	Hardware components (node03)
2	Intel Xeon CPU: E5-2680 v3 (Haswell) - 12 cores; 30MB L3 Smart Cache; 4 Mem. Ch. DDR4 2133 MHz = 68 GB/s
8	SAMSUNG DRAM 16GB Samsung DDR4-M393A2G40DB0-CPB- 2133 MHz
1	Hard disk: 500GB WD5003AZEX Black
1	SSD: 240GB Vertex460A

All 3 nodes are interconnected by the Infiniband network. A shared file system (NFS over Infiniband) is available. There is a common front-end for accessing the cluster and management of jobs on Nodes 01-03.

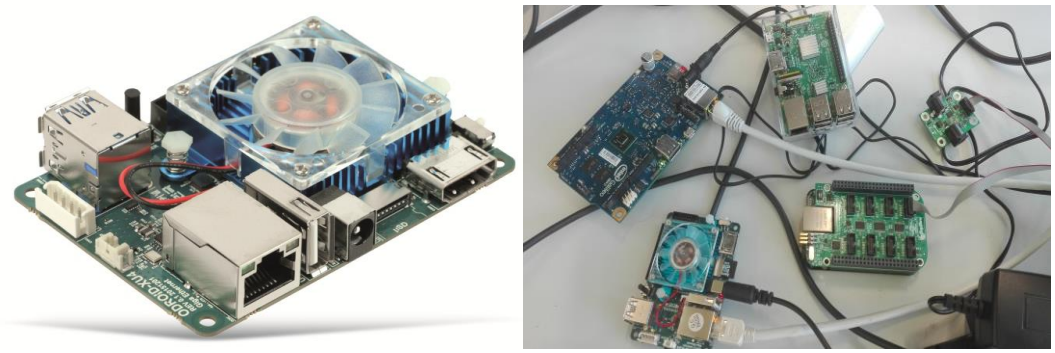
A dedicated power measurement system is installed on each of the nodes in order to collect power and energy infrastructure profiles.

The cluster will be used for applications demanding high performance without hard real-time requirements, such as the simulation application of the USTUTT-HLRS partner or the surveillance one of GMV.

Figure 2: EXCESS cluster of USTUTT-HLRS

2.1.2 Low-Power and Mobile Devices

For evaluation purposes, the consortium partners will use a range of portable ARM-based platforms such as ODROID-XU4 (Figure 3). ODROID is a one-board SAMSUNG Exynos5 Octa CPU with 2 GB DDR3. The CPU consists of 4 more powerful Cortex-A15 and 4 less powerful Cortex-A7 cores. A dynamic load balancing system of the board decides on which core every thread should be executed. This board is planned to be used for all 3 pilot PHANTOM use cases. The MOM (Multi-Objective Scheduler) will be able to instruct the native scheduler of the board about the better allocation properties and the impact of the static vs. the native dynamic scheduling will be evaluated. There are a variety of other boards that will be evaluated, including Raspberry PI-3 (1,2GHz, 4x Cortex-A53v8), Intel Galileo Gen 2, and some other.



a) b)

Figure 3: Low-power hardware testbed: a) ODROID-XU4 board of USTUTT-HLRS, b) mini-embedded cluster made of Raspberry Pi-3, Galileo, and Odroid

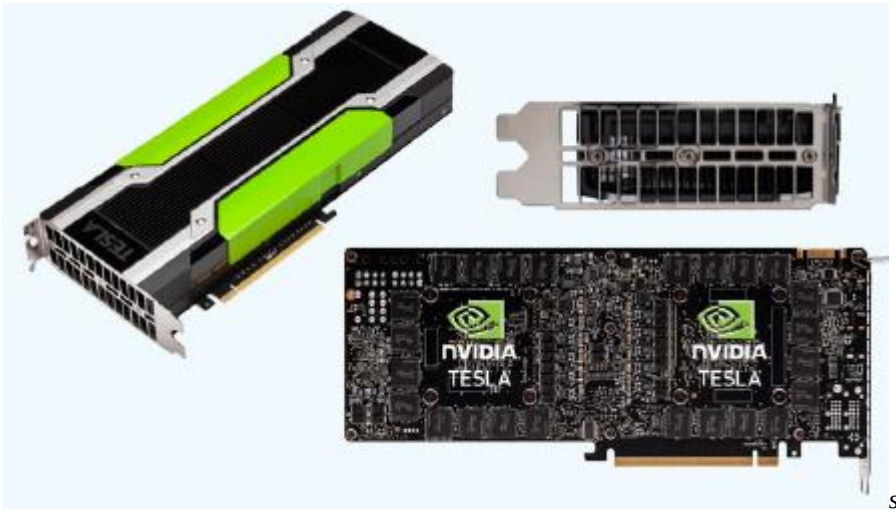
2.2 ACCELERATION DEVICES

2.2.1 GPU-Based Accelerators

The GPU-based accelerated compute resources are enjoying an increasing popularity in many infrastructure communities from embedded systems to high performance computing. Piz Daint – the Europe's fastest supercomputer[1], located in Switzerland, is made up of Tesla P100 GPUs from NVDIA.

In the project, we are going to use the GPU facilities that are provided by the EXCESS cluster, described in Section 2.1. Its Node01 includes an NVIDIA Tesla K40 graphic card, and node03 – an NVIDIA Tesla K80 one. Tesla K40 can reach the performance of up to 1.43 TFLOPS on 2.880 CUDA cores and provide 12 GB of the local memory (GDDR5). Tesla K80 (Figure 4) consists of 2 Tesla K40 and offers the nearly double performance (Figure 4).

[1] https://www.top500.org/lists/2016/11/



source: NVIDIA

Figure 4: NVIDIA Tesla K80 GPU of USTUTT-HLRS

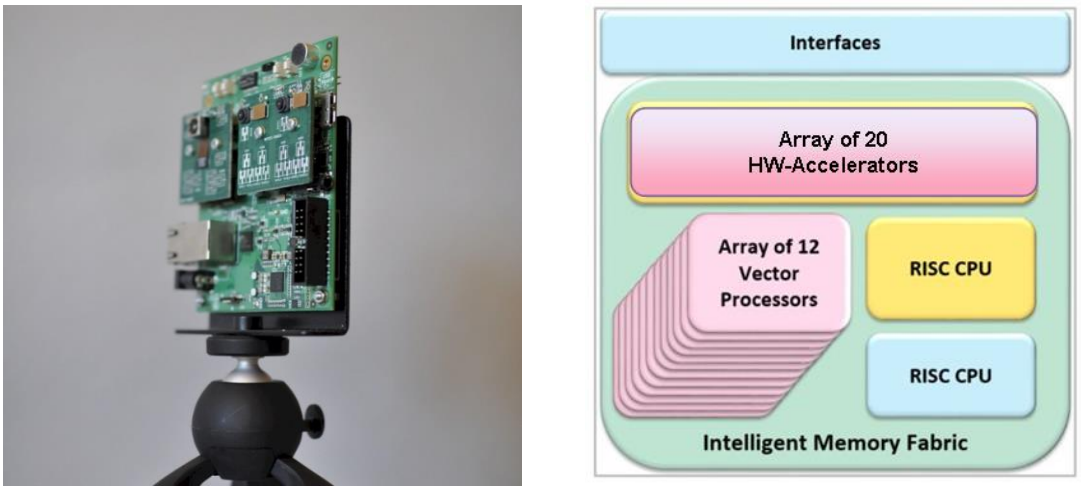
The NVIDIA GPUs will be used for the massively-parallel and data-centric (thousands of independent threads working on shared data sets) parts of the computationally-intensive parts of the USTUTT-HLRS and GMV workflows.

2.2.2 Specialised Acceleration Devices

Myriad 2[2] (Figure 5a) is a specialised SoC (system-on-chip) hardware platform of a company called Movidius (currently owned by Intel) that provides solutions for mobile image processing, for which the CPU- and GPU-based solutions prove ineffective (due to the mixed image rendering and processing workflows as well as the very high requirements on portability and energy efficiency).

The Myriad 2's SoC architecture (Figure 5b) provides on the same silicon alongside with low-power general-purpose SPARC Leon cores (one for scheduling within the SoC and the other for running the user code within a real-time operating system) a set of 20 hard-coded (ASICS) for HW-acceleration of some typical image processing operations as well as 12 SHAVE 128-bit SIMD vector units. A common memory for all SoC resources of 2 MB is provided by the board that functions as a hybrid L3 cache. Additionally, up to 1 GB of external DRAM memory can be supported by the board.

[2] http://www.tomshardware.com/news/movidiud-myriad2-vpu-vision-processing-vr,30850.html



a) b)

Figure 5: Movidius' Myriad 2 platform of USTUTT-HLRS: a) board view, b) architecture

This board is planned to be used by all use cases. The advantages of the heterogeneous SoC components, including the acceleration parts, will be leveraged by the PHANTOM platform components to optimise the application performance and power consumption characteristics.

More details on the Myriad2 board, including the software stack information, are provided in Appendix 2.

2.3 RECONFIGURABLE DEVICES

These devices include the reprogrammable-logic hardware that allows the hardware-based application development.

2.3.1 FPGAs

The FPGA boards targeted by the PHANTOM platform are all Zynq and Zynq Ultrascale-class boards from Xilinx (Figure 6). Zynq FPGAs are comprised of an ARM system-on-chip with attached reconfigurable logic. The ARM SoCs are 32-bit or 64-bit multiprocessor systems, clocked from 800Mhz upwards, and consisting of currently 2 or 4 cores. They contain full MMUs and are capable of running a standard Linux mainline kernel. The FPGA logic is tightly coupled to the CPUs, allowing for very high-speed data transfer. Therefore, the CPUs can be used for varied software tasks, and to invoke custom hardware designs on the reconfigurable logic for high-volume, low-latency data processing.

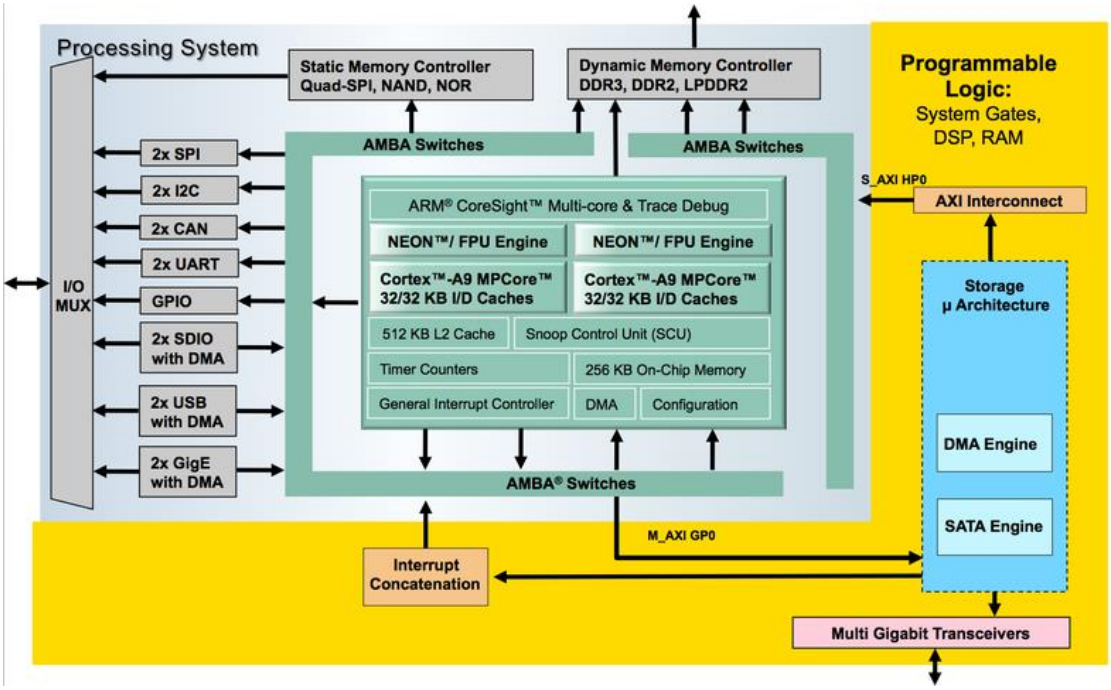


Figure 6: The Zynq reference block diagram (from www.xilinx.com)

In the context of the PHANTOM project, this means that FPGA targets are actually an FPGA with tightly-coupled CPU SoC. PHANTOM does not target a "raw" FPGA but requires a CPU-host instead. This allows FPGA targets to host both the software and hardware parts of IP cores.

3. RELEASED COMPONENT: MONITORING CLIENT

3.1 FUNCTIONALITY

3.1.1 Overview

The PHANTOM Monitoring Client (henceforth, the Client) is a part of the PHANTOM monitoring framework which is in charge of acquiring performance- and energy-relevant data from the hardware and applications running on this hardware (Figure 7).

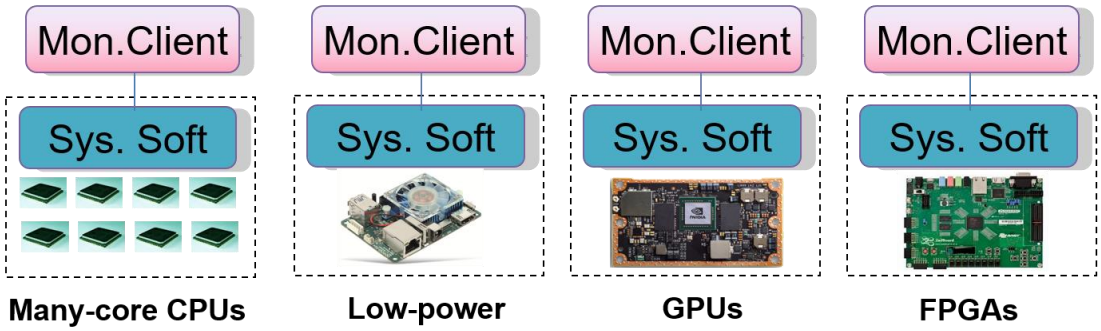


Figure 7: PHANTOM Monitoring Client design

The Client is a lightweight service that is installed on top of the available system software stack. 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 Client is running as a separate system process in the background of the OS (Linux). At every time interval, which can be customized by the user (in the range from milliseconds to hours, depending on the application), the Client collects the data from the controlled device and transmits them to the Monitoring Server (see Section 4).

The major sources of information, acquired by the Client, are:

- 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.

- 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

  The power consumption is obtained either from the dedicated hardware counters (as provided by some server CPUs like Intel Haswell) or from an external power measurement system, such as ACME[3] (Figure 8).

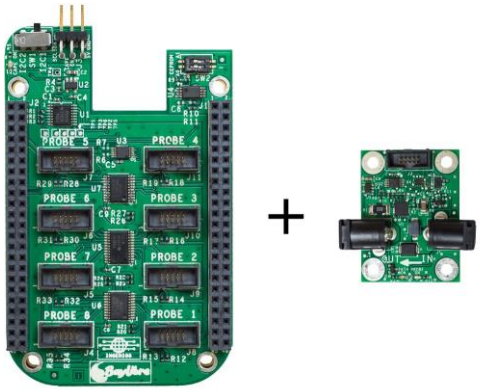


Figure 8: External power measurement board from ACME

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 utilization, etc. (Figure 9). The detailed specification of all metrics is provided in Appendix 1.

Collection of the data from each of the above-listed sources is performed by a special plug-in, which can be easily 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 for the specific list).

[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.

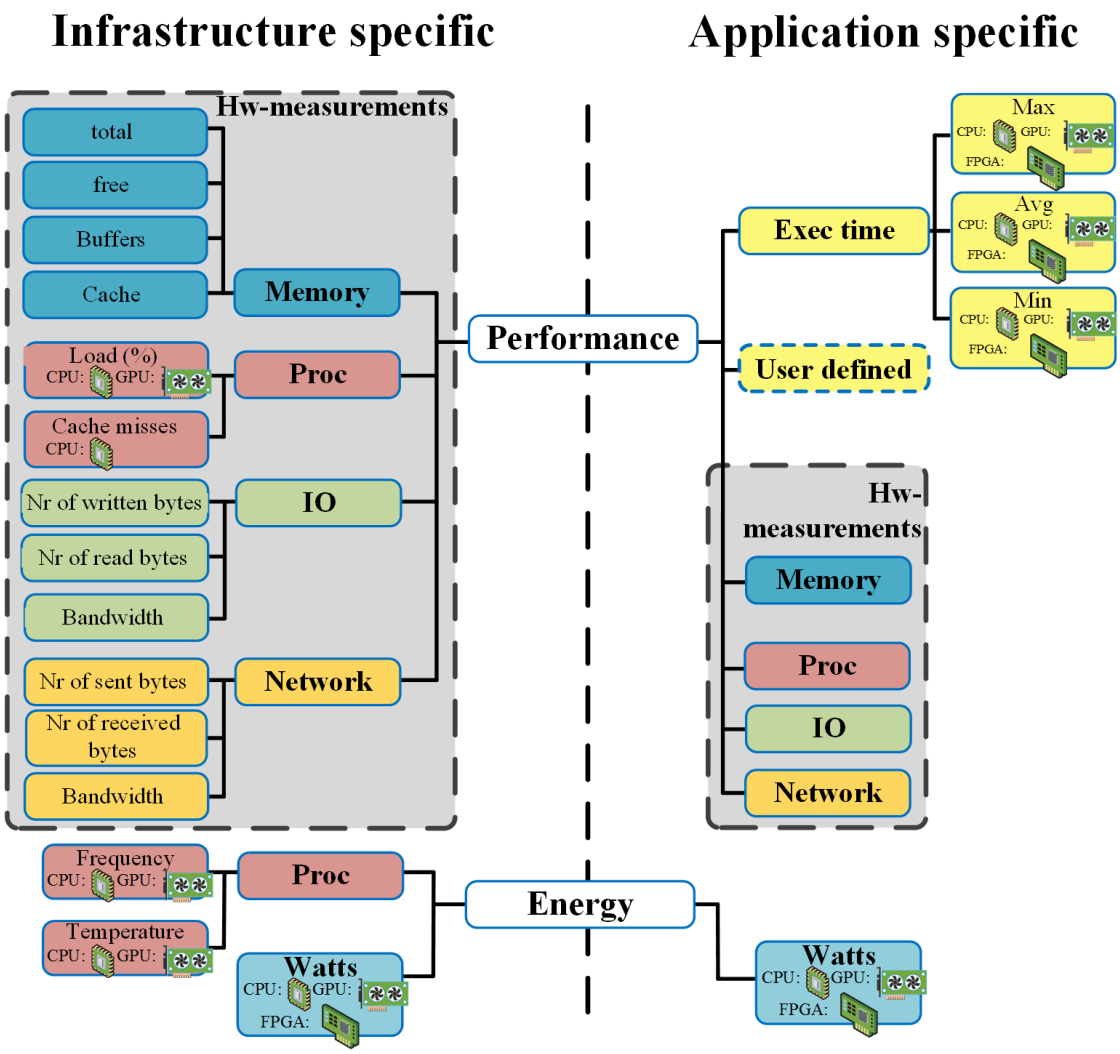


Figure 9: PHANTOM Monitoring Client metrics' taxonomy

The metrics that are collected by the Client fall into one of two following general categories (Figure 10):

- Infrastructure-specific metrics
- Application-specific metrics

The infrastructure-specific metrics are collected for the whole node/board, regardless of the applications running on it. These metrics are always collected after every time interval, which can be defined (along with a list of metrics to be acquired from the whole available set) by the user following a simple configuration procedure.

The application-specific metrics can be collected on the per-application basis. They contain a subset/fraction of the hardware-specific metrics, which are related to a

particular application execution. This functionality can be enabled for the applications by a PHANTOM Monitoring Library (see in deliverable D1.3).

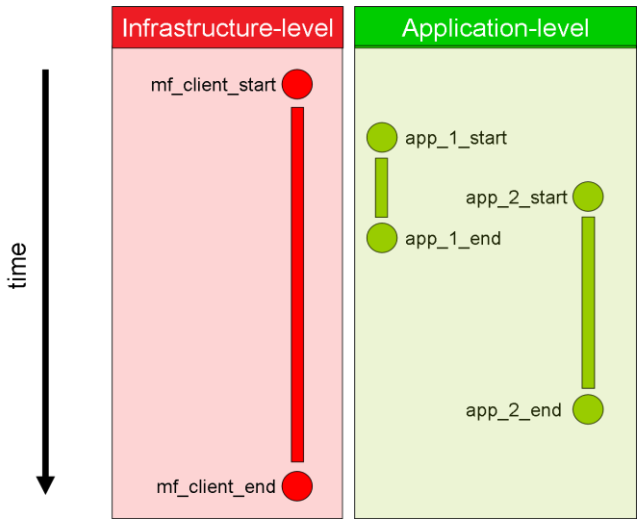


Figure 10: Infrastructure- and application-level monitoring with the Monitoring Client

3.1.2 PHANTOM Advances to State-of-the-Art

The Monitoring Client was initially provided as a part of ATOM – a monitoring framework solution developed in the frame of EXCESS and DreamCloud projects. In PHANTOM, the Client was majorly refactored with the aim to support the heterogeneity of the hardware resources (in particular the inclusion of the reconfigurable ones), allow the organization of the resources within the distributed infrastructure, enable the component-based organization of the PHANTOM applications, improve the usability and configurability on the user-side.

A short list of the major extensions is as follows:

- Support of buffering in order to decrease the network interception that is introduced by transmitting the monitoring data from the Clients to the Server.
- Introduced application-specific metrics along with the hardware-specific ones.
- Enabled seamless support of external measurement systems along with the counter-based hardware capabilities.
- Introduced dynamic configuration of metrics to be monitored, which implies the user can define the metrics at the application run-time via an intuitive user interface.

3.1.3 Enhancements to Previous Release

The enhanced release aims to improve the usability of the initially provided components (as a part of release 0.1) and incorporate the feedback from the first evaluation

(performed by the application providers). The metrics specification was extended and documented, so that the users now have a clearer overview of all metrics and better ways of using them for the applications.

The major functional advance of the new release is the support of the application-level metrics. The Client is now able to determine the subset of the overall resource utilization caused by a specific application. This is of a special importance for the embedded (low-power) applications as well as for the HPC architectures (which host many applications simultaneously).

Additionally, a buffering mechanism was implemented in order to i) fit the requirement of the embedded applications (based on the Telecom use case's feedback) and ii) minimize the network traffic during the transmission of the metric values, collected by the Client, to the Server.

3.2 DEPENDENCIES AND INSTALLATION

The PHANTOM Monitoring Client relies on various external libraries, as summarized in Table 3.

Table 3: Monitoring Client dependencies

Component	Version	Link
General		
PAPI-C	5.4.0	http://icl.cs.utk.edu/papi/
CURL	7.37.0	http://curl.haxx.se/download/
Apache APR	1.5.1	https://apr.apache.org/
Apache APR Utils	1.5.3	https://apr.apache.org/
bison	2.3	http://ftp.gnu.org/gnu/bison/
flex	2.5.33	http://prdownloads.sourceforge.net/flex/
sensors	3.4.0	https://fossies.org/linux/misc/
m4	1.4.17	https://ftp.gnu.org/gnu/m4
libiio	1.0	https://github.com/analogdevicesinc/libiio.git
hwloc	1.11.2	https://www.open-mpi.org/software/hwloc/v1.11/downloads/

EXCESS queue	release/0.1.0	https://github.com/excess-project/data-structures-library.git
GPU-hosts		
Nvidia GDK	352.55	https://developer.nvidia.com/gpu-deployment-kit/
FPGA-hosts		
Vivado Design Suite - HLx Editions	2017.4	https://www.xilinx.com/support/download.htmlns

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 1). 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 5).

Listing 1: Monitoring Client setup

```
$ git clone https://github.com/phantom-monitoring-framework/phantom_monitoring_client
$ cd  phantom_monitoring_client
$ ./setup.sh  "unique_name_of_platform"

$ 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 5). The Resource Manager contains settings for each specific type of the supported hardware resources (e.g. x86 CPU, ARM-based CPUs, etc.), which are used to tailor the algorithms of monitoring thus improving its quality.

3.3 UPCOMING ACTIONS

The following major actions will be performed by the final release:

Better support of reconfigurable hardware. In order to support monitoring of FPGA-hosting devices (such as Zynq boards of x86 with plugged FPGA cards), a special IP core will be developed that will implement the basic monitoring functions for FPGA devices. It will provide important insights into the functioning of the FPGA device and its communication with the host-CPU.

Broaden scope of monitoring functions. The existing monitoring plug-ins will be extended to support all metrics that are identified by the taxonomy in Figure 9.

Integration with Resource Manager. Monitoring Client will be able to receive settings for the specific type of the architecture directly from the Resource Manager, so that no additional actions for the users are required.

Distribution packages for major architecture types. To simplify the configuration by the end-users, a set of distribution packages for the major targeted hardware platforms will be provided. The packages will contain a pre-configured set of plug-ins and settings that should simplify the installation and configuration of the Monitoring Client.

Improved documentation. We will provide monitoring handbooks for the major hardware platforms that are supported by the Monitoring Framework. Such a handbook will contain typical usage strategies of the monitoring components, getting-started guides for the end-users as well as troubleshooting instructions.

4. RELEASED COMPONENT: MONITORING SERVER

4.1 FUNCTIONALITY

4.1.1 Overview

The PHANTOM Monitoring Server is responsible for:

- gathering information from all Monitoring Clients of the heterogeneous distributed PHANTOM infrastructure,
- storing the collected monitoring data in a scalable and distributed monitoring database,
- hosting services (scripts or application containers) that perform analytics on the historical sets of the collected data, and
- providing access to the analytics services through a generic RESTful interface to the interested customers (users as well as the other components of the PHANTOM framework, such as the Multi-Objective Scheduler or the Resource Manager).

The Server includes storage for keeping all data collected from the Clients, and a web-server for hosting the analytics applications and services. See Figure 11 below.

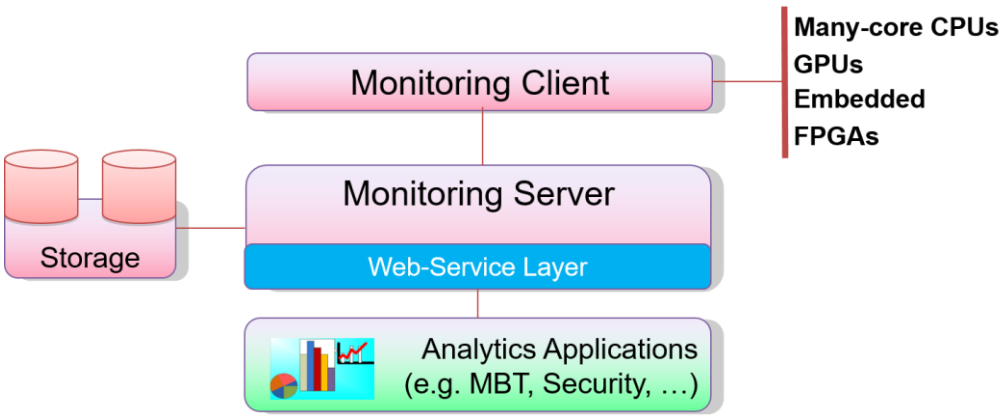


Figure 11: Architecture of Monitoring Server

4.1.2 PHANTOM Advances to State-of-the-Art

The PHANTOM Monitoring Server is based on the previously available ATOM solution, which has been extended to support the requirements that are specific for the PHANTOM platform. Some of the major extensions are as follows:

- support of the metric data buffering in order to reduce the network traffic overhead between the Clients and the Server
- support of a much broader scope of monitoring data analytics application, from retrieving max, min and average values of the specific metrics to reasoning on the complex data across one or many applications (especially for the application-level monitoring, as described in Section 3)

4.2 DEPENDENCIES AND INSTALLATION

The Monitoring Server's dependencies are summarized in Table 4.

Table 4: Monitoring Server dependencies

Component	Version	Link
Elasticsearch	1.4.4	https://www.elastic.co/products/elasticsearch
Node.js	0.9	https://apr.apache.org/
npm	1.3.6	https://www.npmjs.com/

To install all the prerequisites, a setup script is provided. The following commands are used for this (Listing 2).

Listing 2: Monitoring Server installation scripts

```
$ https://github.com/phantom-monitoring-framework/phantom_monitoring_server.git
$ cd phantom_monitoring_server
$ ./setup.sh
```

The server can be easily controlled as a Linux service by the following commands (Listing 3).

Listing 3: Monitoring Server management scripts

```
$ sudo service phantom_server start
$ sudo service phantom_server stop
$ sudo service phantom_server restart
$ sudo service phantom_server status
```

To simplify the process of environment setup, the release provides a bash script in the source code repository, which downloads all dependencies and installs them locally in the project directory. After the prerequisites installation, the PHANTOM monitoring

server can be compiled, built, and installed by using the provided Makefile. Further installation instructions are provided at:

https://github.com/phantom-monitoring-framework/phantom_monitoring_server

4.3 USAGE EXAMPLES

The PHANTOM Monitoring Server can be reached by means of RESTful GET/PUT requests in order to obtain the information/statistics about the controlled hardware devices and applications. Listing 4 shows some basic queries that are often used by the customers (the platform's MOM or the end-users).

Listing 4: Basic queries to Monitoring Server

```
# APPLICATIONS/WORKFLOWS
GET  /v1/phantom_mf/workflows
GET  /v1/phantom_mf/workflows/:application_id
PUT  /v1/phantom_mf/workflows/:application_id -d '{...}'
# EXPERIMENTS
GET  /v1/phantom_mf/experiments
GET  /v1/phantom_mf/experiments/:execution_id?workflow=:application_id
POST /v1/phantom_mf/experiments/:application_id -d '{...}'
# METRICS
GET  /v1/phantom_mf/metrics/:application_id/:task_id/:execution_id
POST /v1/phantom_mf/metrics -d '{...}'
POST /v1/phantom_mf/metrics/:application_id/:task_id/:execution_id -d '{...}'
# PROFILES
GET /v1/phantom_mf/profiles/:application_id
GET /v1/phantom_mf/profiles/:application_id/:task_id
GET /v1/phantom_mf/profiles/:application_id/:task_id/:execution_id
GET /v1/phantom_mf/profiles/:application_id/:task_id/:execution_id?from=...&to=...
# RUNTIME
GET /v1/phantom_mf/runtime/:application_id/:execution_id
GET /v1/phantom_mf/runtime/:application_id/:task_id/:execution_id
# STATISTICS
GET /v1/phantom_mf/statistics/:application_id?metric=...
GET /v1/phantom_mf/statistics/:application_id?metric=...&host=...
GET /v1/phantom_mf/statistics/:application_id?metric=...&from=...&to=...
GET /v1/phantom_mf/statistics/:application_id?metric=...&host=...&from=...&to=...
GET /v1/phantom_mf/statistics/:application_id/:execution_id?metric=...
GET /v1/phantom_mf/statistics/:application_id/:execution_id?metric=...&host=...
GET /v1/phantom_mf/statistics/:application_id/:execution_id?metric=...&from=...&to=...
GET /v1/phantom_mf/statistics/:application_id/:execution_id?metric=...&host=...&from=...&to=...
GET /v1/phantom_mf/statistics/:application_id/:task_id/:execution_id?metric=...
GET /v1/phantom_mf/statistics/:application_id/:task_id/:execution_id?metric=...&host=...
GET /v1/phantom_mf/statistics/:application_id/:task_id/:execution_id?metric=...&from=...&to=...
GET /v1/phantom_mf/statistics/:application_id/:task_id/:execution_id?metric=...&host=...&from=...&to=...
```

4.4 UPCOMING ACTIONS

The following major actions will be performed by the final release:

Providing a richer set of analytics applications. We will continue the implementation of the containers (to be hosted by the server) for the most typical analytics operations, as suggested by the use case providers. Some examples of such applications are obtaining the average duration of the execution in different resource configurations, analysis of the hardware utilization rate by the applications, etc.

Simplification of interfaces. We will provide a more light-weight specification for direct accessing the monitoring data with a possibility of performing basic analysis functions (i.e. without the need to create an analytics application, as described for the previous bullet) directly from the RESTful queries.

Improved documentation. We still have to provide monitoring handbooks for the major hardware platforms that are supported by the Monitoring Framework.

5. RELEASED COMPONENT: RESOURCE MANAGEMENT SERVICE

5.1 FUNCTIONALITY

5.1.1 Overview

The PHANTOM Resource Manager allows connection of different heterogeneous resources into a common distributed infrastructure, including CPU-only, as well as GPU- and FPGA-hosting devices. The Manager allows the control of all devices, included into the infrastructure, as well as of the applications running on them.

Some other goals include:

- Dynamic identification of the application bottlenecks with the aim of better hardware utilization and thus decreased execution time (by leveraging the Monitoring Framework services)
- Dynamic Power Management aiming to apply techniques like DVFS (Dynamic Voltage and Frequency Scaling) to decrease the power consumption of the hardware while ensuring acceptable quality metrics
- Implementation of security management options as provided by the Security Manager

The Resource Manager is constituted by

- A **management application** that is deployed as a Monitoring Server container along with the other analytics applications. The application provides the following services:
    - **Resource registry service**. This service is responsible for maintaining the list of all heterogeneous devices of the common infrastructure. The service provides interfaces for adding/editing/deleting devices to or from the infrastructure. Each entry of the resource registry contains information about the resources such as the network IP address, type of resource (CPU, GPU, …), static configuration information (like number of cores), etc.
    - **Resource status tracking service**. This service provides online data about the status of all registered infrastructure resources. Utilization of computation cores, available RAM, actual I/O and memory bandwidth as well as other essential information on the resources are included (Figure 12).

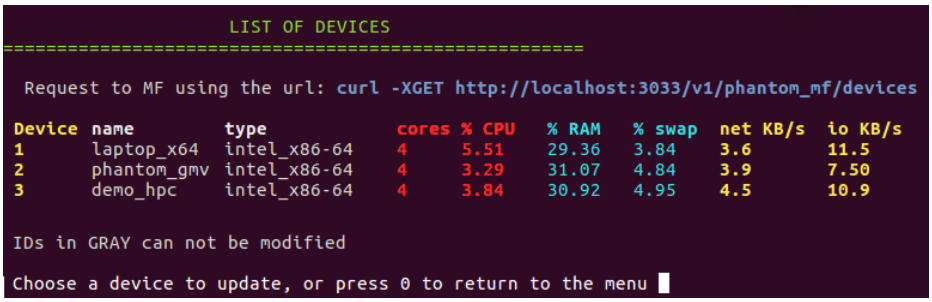


Figure 12: Interface of resource status tracking service

The consolidated resource data are obtained by the Resource Manager from the Monitoring Server and updated constantly.

- **Application registry service**. This service allows application developers to enroll their applications into a central registry that retains the most essential information about the application such as location of source code in the Repository, availability of binaries, etc.

- **Application status tracking service**. This service is responsible for acquiring the status of the application instances. For already completed executions, the duration and the resource utilization (including the energy consumption) are retrieved. For applications that are still running the current consumption is retrieved. The service relies on the data obtained and consolidated from the Monitoring Server (Figure 13).

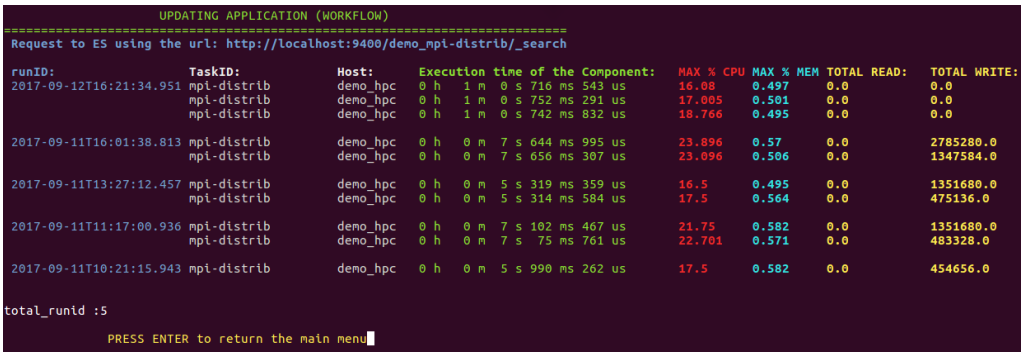


Figure 13: Interface of application status tracking service

- **Dynamic power management service**. The service consolidates a set of scripts that are used to control the power mode in which the controlled device is operating. The scripts as well as the underlying management tools are outcomes of the DreamCloud project. For the CPU-based devices, for example, the scripts apply the DVFS (Dynamic Voltage and Frequency Scaling) technique in order to reduce the power consumption of the individual cores.

The data of all above-shown interfaces can be obtained from the RESTful API of the Resource Manager (see an example in Table 5).

Table 5: RESTful APIs of PHANTOM Resource Manager service

configs – resource configuration service		
/configs	GET	Get a list of all registered platforms with links to their configuration details
/configs/:platform_id	GET	Get configuration details (e.g. monitoring-related parameters) for a specific platform
	PUT	Add/Change the configuration (e.g. monitoring-related parameters) for a specific platform
resources – status tracking service		
/resources	GET	Get a list of all registered platforms with links to their resources details
/resources/:platform_id	GET	Get information about the resources available for a specific platform
	PUT	Create/Update the resources information for a specific platform

5.1.2 PHANTOM Advances

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 (see Section 3) and it is not necessary introduce an additional monitoring layer with a considerable overhead.

Therefore, the PHANTOM Resource Manager was developed, leveraging the technologies and ideas from past projects; DreamCloud and EXCESS.

5.2 DEPENDENCIES AND INSTALLATION

The PHANTOM Resource Manager requires the availability of the PHANTOM Monitoring Server, which acts as source of online data and also the host of the Resource Manager application container.

5.3 UPCOMING ACTIONS

The following major actions will be performed by the final release:

- **Specification of RESTful interfaces.** Resource Manager relies on a rich set of hierarchically organized data. All interfaces accessing those data need to be properly documented, tested and evaluated.

- **Development of GUIs.** The GUIs that aim to represent the information obtained from the RESTful interfaces of the Resource Manager in a human/user-friendly form need to be developed. A basic set of command-line and also graphical user interfaces will be provided by the time of the final release.

6. RELEASED COMPONENT: PHANTOM FPGA LINUX DISTRIBUTION AND FPGA INFRASTRUCTURE

6.1 FUNCTIONALITY

6.1.1 Overview

The Linux distribution that is used for the PHANTOM FPGA targets is defined and standardised to aid coordination between partners. This component allows for the parts of the distribution to be easily rebuilt and extended, for example to support new FPGA boards.

The main purpose of the PHANTOM Linux is to be able to automatically integrate PHANTOM IP cores into an FPGA platform (CPU plus FPGA) when instructed by the Multi-Objective Mapper.

The PHANTOM distribution also contains the scripts which create PHANTOM-compatible FPGA designs. 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.

The 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.

6.1.2 PHANTOM Advances

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. This is encapsulated in a provided API, which is documented in the release. This API is used by the PHANTOM IP core developers, while writing the software part of their IP core, to implement the transfer of data in and out of the reconfigurable logic.

- 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.

6.2 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.

6.3 FILESYSTEM CONFIGURATION

Based on feedback from application providers, it was observed that it is necessary to tailor the size of the generated filesystem image for when small embedded systems with limited storage are being used. To this end, the distribution gives two options for filesystem generation: a full Debian-based system with a customisable selection of packages, and a limited BusyBox-based system generated with Buildroot (https://buildroot.org).

The Debian filesystem is designed to be run from an SD card (or similar) as persistent, read-write storage. The build scripts can generate this automatically using Multistrap, including the necessary PHANTOM Linux components. The Multistrap configuration file contains a default set of packages that creates a system of just under 400MB. Packages can be added or removed from this configuration as required, with a minimal system totalling under 200MB.

The BusyBox-based filesystem 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. The build scripts can generate this automatically using Buildroot, including the necessary PHANTOM Linux components. A default Buildroot configuration is provided, which can be customised through editing the configuration file or using Buildroot's menu-based configuration editor.

6.4 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.

6.4.1 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.

It is therefore necessary to be able to access arbitrary memory ranges from user-space, so that the software parts of a PHANTOM component can communicate with its hardware parts on the FPGA. It is possible to modify the kernel to allow this access, but this bypasses the kernel memory protection and so compromises the integrity of the system.

The three standard methods of accessing memory in Linux are UIO (Userspace I/O), VFIO (Virtual Function I/O), and the kernel's system memory device node (`/dev/mem`). The system memory device node does not offer any mechanism for memory protection, allowing any process with access to this special file the ability to read or write practically any system memory. The VFIO driver relies on an IO memory management unit which is not present on a Zynq-7000 SoC. Therefore UIO is used.

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

6.4.2 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 and access to this is controlled using a memory management unit (MMU) core on each component's AXI master bus. The MMU allows address regions to be set-up at synthesis time, each with read/write permissions set for the specific 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.

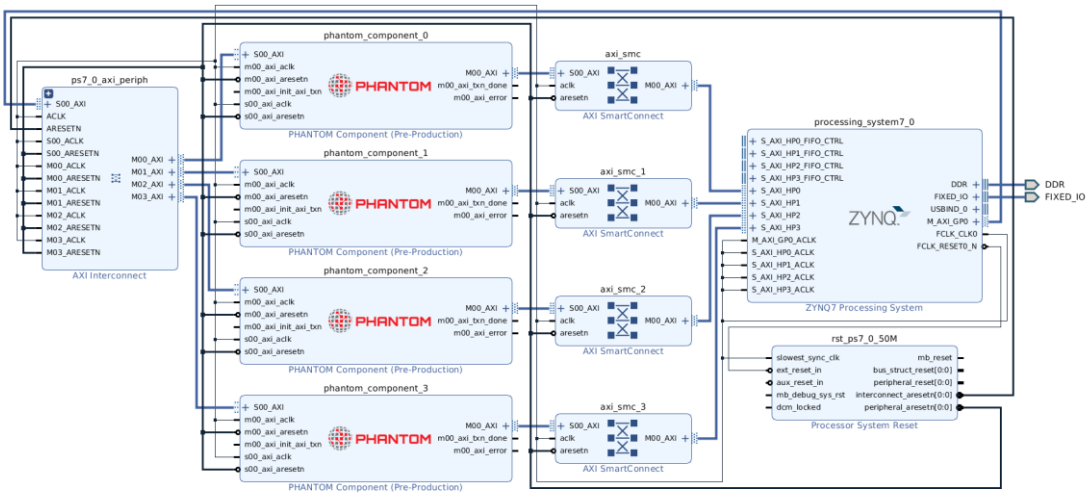


Figure 15: AXI SmartConnect MMU used to restrict hardware access to memory on new FPGA devices.

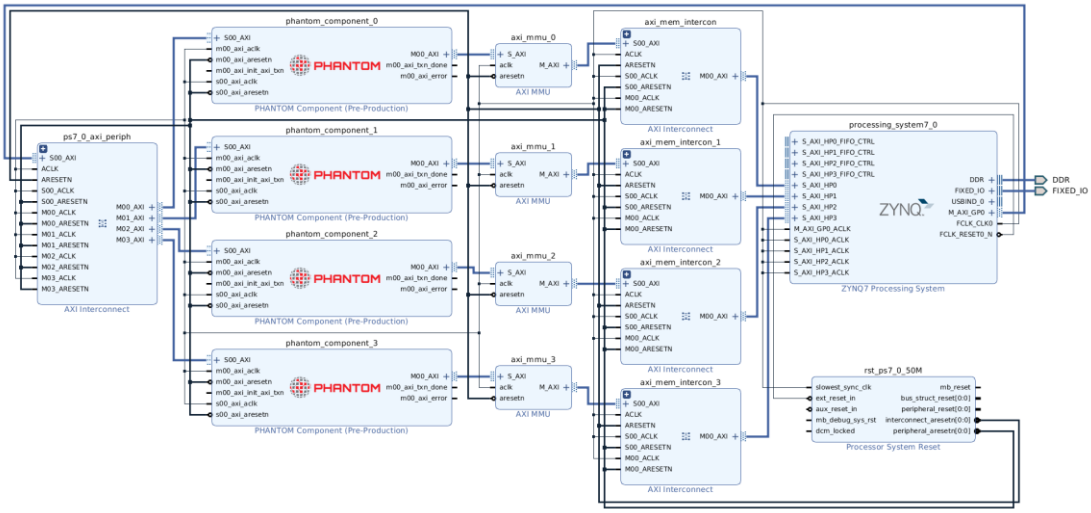


Figure 16: Memory protection is supported on older FPGA devices with the AXI MMU and AXI Interconnect cores.

6.5 UPCOMING ACTIONS

The following major actions will be performed by the final release:

Action 1. Complete support for 64-bit architectures and FPGAs.

Action 2. The implementation of custom designs to monitor bandwidth use between the CPU and FPGA.

7. RELEASED COMPONENT: PHANTOM IP CORES MARKETPLACE

7.1 FUNCTIONALITY

7.1.1 Overview

The IP Core Marketplace can be seen as a part of the PHANTOM Repository specific for storing all the dedicated FPGA logic block. 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.

Currently, the following IP Cores are available in the IP Core Marketplace:

- Discrete Wavelet Transform (DWT)
    - Input: Image to be processed with size 2048x2048
    - Output: 4 images with size 1024x1024
- Inverse Discrete Wavelet Transform (IDWT)
    - Input: 4 images to be processed with size 1024x1024
    - Output: Image with size 2048x2048

7.1.2 PHANTOM Advances

A conjugation of the easy to use PHANTOM platform with the pre-designed IP cores can accelerate applications without the need to redesign from scratch the FPGA specific part.

A short list of the major extensions is as follows:

- Reusability of IP Cores that started from application specific needs.
- Implementation of generic functions, that can be used outside PHANTOM application cases.

7.2 DEPENDENCIES AND INSTALLATION

IP Core Marketplace can be fetched from the following git repository.

```
$ git clone https://github.com/PHANTOM-Platform/PHANTOM-IP-Core-Marketplace
```

7.3 USAGE EXAMPLES

Discrete Wavelet Transform (DWT)

A discrete wavelet transform (DWT) is any wavelet transform for which the wavelets are discretely sampled. As with other wavelet transforms, a key advantage it has over Fourier transforms is temporal resolution: it captures both frequency and location information (location in time). Wavelets are often used to denoise two dimensional signals, such as images. The original image is low-pass filtered, high-pass filtered and downscaled yielding four images, each describing local changes in brightness (details) in the original image.

Figure 17: a) Image before DWT b) Image after DWT

Design Flow

This filter is implemented in FPGA logic encapsulated in an IP Core that can be found in the PHANTOM IP Core Marketplace. This DWT IP Core (Figure 18) receives an image via an AXI Stream, does the processing transparently and outputs four images also via AXI Stream interfaces. The IP Core can be controlled (Start, Stop, etc.) via the axi_CONTROL_BUS.

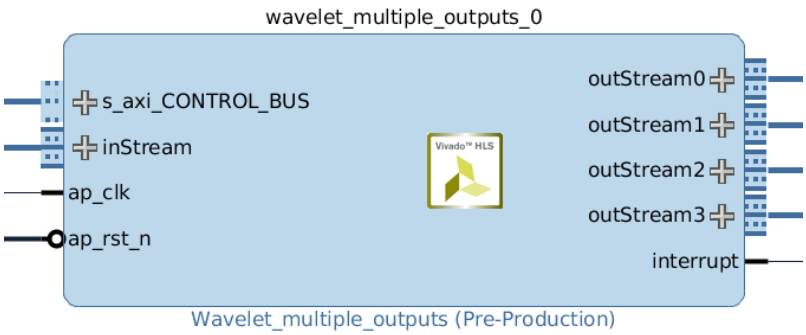


Figure 18: IP core interfaces example

AXI is part of ARM AMBA, a family of micro controller buses first introduced in 1996. The first version of AXI was first included in AMBA 3.0, released in 2003. AMBA 4.0, released in 2010, includes the second major version of AXI, AXI4.

- There are three types of AXI4 interfaces:
- AXI4: For high-performance memory-mapped requirements.
- AXI4-Lite: For simple, low-throughput memory-mapped communication (for example, to and from control and status registers).
- AXI4-Stream: For high-speed streaming data.

To be able to use this IP Core it needs to be integrated in a functional FPGA design that exploits these types of interfaces. The design in Figure 19 shows the normal usage of this DWT IP Core integrated with four DMAs to handle all the data streaming, interconnects to manage the connections between different blocks and the Zynq Processing System, to control the data transfer between the DMAs and the IP Core and also to control the IP Core functionality.

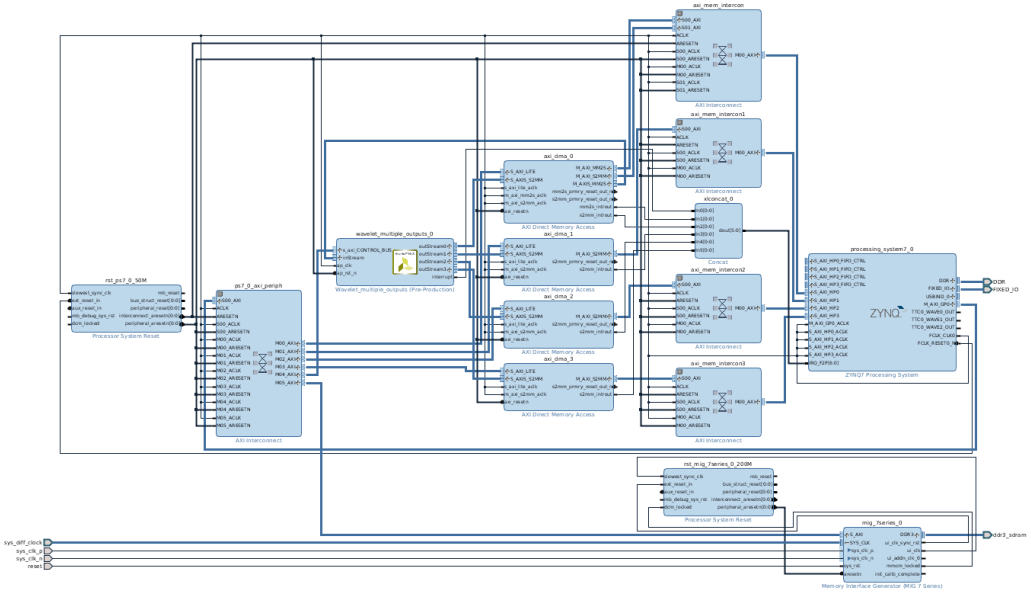


Figure 19: IP core design example

All the logic blocks are controlled using default AXI interfaces like the ones described above.

- S_AXI_LITE -> DMA Control
- S_AXI -> Memory Interface Control
- S_AXI_CONTROL_BUS -> IP Core Control

AXI Stream interfaces are used between the IP Core and DMAs to handle all the high speed data transfer that goes in and out of the IP Core.

- M_AXIS_S2MM -> Master Stream-to-Memory-Map (S2MM)
- M_AXIS_MMS2 -> Master Memory Map-to-Stream (MM2S)
- S_AXIS_S2MM -> Slave Stream-to-Memory-Map (S2MM)
- S_AXIS_MMS2 -> Slave Memory Map-to-Stream (MM2S)

The DMAs are also connected to the Zynq Processing System via S_AXI_HP (High Performance Slave Interface) that gives them direct access to the physical DDR RAM to allow very fast memory transfers from the DDR to the IP Core and back to the DDR RAM.

- S_AXI_HP (HP0, HP1, HP2, HP3) -> High Performance Slave Interface

Software Flow

The code flow to use the IP Core and DMAs is as follows.

We start by initializing the DMAs and DWT IP Core.

```
CfgPtr = XAxiDma_LookupConfig(XPAR_AXI_DMA_0_DEVICE_ID);
XAxiDma_CfgInitialize(&axiDma0,CfgPtr);
CfgPtr = XAxiDma_LookupConfig(XPAR_AXI_DMA_1_DEVICE_ID);
XAxiDma_CfgInitialize(&axiDma1,CfgPtr);
CfgPtr = XAxiDma_LookupConfig(XPAR_AXI_DMA_2_DEVICE_ID);
XAxiDma_CfgInitialize(&axiDma2,CfgPtr);
CfgPtr = XAxiDma_LookupConfig(XPAR_AXI_DMA_3_DEVICE_ID);
XAxiDma_CfgInitialize(&axiDma3,CfgPtr);

CfgPtr=XWavelet_multiple_outputs_LookupConfig(XPAR_WAVELET_MULTIPLE_OUTPUTS_0_DEVICE_
ID);
XWavelet_multiple_outputs_CfgInitialize(&waveletFilter0,CfgPtr);
```

Then we flush the cache to avoid sending garbage to the IP Core.

```
Xil_DCacheFlushRange((u32)m_dma_buffer_Src,SIZE_ARR*sizeof(u8));
```

Then the IP Core is started and the DMAs are instructed to start transferring data.

```
XWavelet_multiple_outputs_Start(&waveletFilter0);
XAxiDma_SimpleTransfer(&axiDma0,(u32)m_dma_buffer_Src,SEND_BLOCK*sizeof(u8),XAXIDMA_D
MA_TO_DEVICE);
XAxiDma_SimpleTransfer(&axiDma0,(u32)m_dma_buffer_Dst0,RECV_BLOCK*sizeof(u8),XAXIDMA_
DEVICE_TO_DMA);
```

```
XAxiDma_SimpleTransfer(&axiDma1,(u32)m_dma_buffer_Dst1,RECV_BLOCK*sizeof(u8),XAXIDMA_
DEVICE_TO_DMA);
XAxiDma_SimpleTransfer(&axiDma2,(u32)m_dma_buffer_Dst2,RECV_BLOCK*sizeof(u8),XAXIDMA_
DEVICE_TO_DMA);
XAxiDma_SimpleTransfer(&axiDma3,(u32)m_dma_buffer_Dst3,RECV_BLOCK*sizeof(u8),XAXIDMA_
DEVICE_TO_DMA);
```

We wait for the DMAs and IP Core to finish the work.

```
while(XAxiDma_Busy(&axiDma0,XAXIDMA_DEVICE_TO_DMA));
while(XAxiDma_Busy(&axiDma1,XAXIDMA_DEVICE_TO_DMA));
while(XAxiDma_Busy(&axiDma2,XAXIDMA_DEVICE_TO_DMA));
while(XAxiDma_Busy(&axiDma3,XAXIDMA_DEVICE_TO_DMA));
```

And then we invalidate the cache to avoid reading garbage.

```
Xil_DCacheInvalidateRange((u32)m_dma_buffer_Dst0,SIZE_ARR_OUT*sizeof(u8));
Xil_DCacheInvalidateRange((u32)m_dma_buffer_Dst1,SIZE_ARR_OUT*sizeof(u8));
Xil_DCacheInvalidateRange((u32)m_dma_buffer_Dst2,SIZE_ARR_OUT*sizeof(u8));
Xil_DCacheInvalidateRange((u32)m_dma_buffer_Dst3,SIZE_ARR_OUT*sizeof(u8));
```

7.4 UPCOMING ACTIONS

The following major actions will be performed by the final release:

Action 1. Extension of available IP Cores to correlate with the existing use cases.

8. CONCLUSIONS

The deliverable presented the enhanced version of the PHANTOM monitoring platform, runtime environment for reconfigurable (FPGA) components, and resource management framework. These components play an essential role for the PHANTOM software stack and are used by the other PHANTOM components, e.g. by the Multi-Objective Mapper (MOM). In particular, the Monitoring Framework provides useful insights in the application performance and hardware utilization, which is essential for scheduling. The FPGA Linux distribution enables using reconfigurable platforms in the same way as the standard CPUs and GPUs require.

This release builds on the previous one by incorporating the first round of feedback from the application providers. This has led to improvements in the handling of user-defined monitoring metrics, analytics, and their documentation. Improvements to the Linux subsystem target a wider range of devices, including more powerful 64-bit systems, and smaller embedded devices with limited storage.

In the remaining lifetime of the PHANTOM project, the components will be further improved according to the identified actions and provided in the upcoming final (D4.4 at M32) releases.

APPENDIX 1. MONITORED METRICS

Table 6: List of metrics for standard CPU-based devices

category		metrics	methodology	unit	remarks
Application-level	performance	execution time	int clock_gettime(CLOCK_REALTIME, struct timespec *tp)	ns	
		CPU execution time	int clock_gettime(CLOCK_PROCESS_CPUTIME_ID, struct timespec *tp)	ns	
	resources utilization	CPU utilization	(process cpu time (/proc/[pid]/stat)) / (global cpu time (/proc/stat))	%	http://stackoverflow.com/questions/1420426/calculating-cpu-usage-of-a-process-in-linux
		RAM utilization	VmRSS (/proc/[pid]/status) / MemTotal (/proc/meminfo)	%	
		swap utilization	VmSwap (/proc/[pid]/status) / SwapTotal (/proc/meminfo)	%	
		virtual memory size	VmSize (/proc/[pid]/status)	KB	
	IO	disk IO throughput	(read_bytes + write_bytes) / seconds (/proc/[pid]/io)	Bytes/s	since kernel 2.6.20
	power	power (with given pid)	ptop (include CPU, memory, disk, wireless network)	milli-watt	
Infrastructure-level	performance	floating point instructions per second	PAPIF_flips	Mflip/s	PAPI
		floating point operations per second	PAPIF_flops	Mflop/s	
	resources utilization	CPU utilization	total_cpu_time – idle_time / total_cpu_time (proc/stat)	%	https://github.com/Leo-G/DevopsWiki/wiki/How-Linux-CPU-Usage-Time-and-Percentage-is-calculated
		RAM utilization	(MemTotal – MemFree) / MemTotal (proc/meminfo)	%	http://www.computerworld.com/article/2722141/it-management/making-sense-of-memory-usage-on-linux.html
		swap utilization	(SwapTotal – SwapFree) / SwapTotal (proc/meminfo)	%	
	IO	disk IO utilization	(Field 10 (# of milliseconds spent doing I/Os) in /proc/diskstat) / (total time)	%	https://www.kernel.org/doc/Documentation/iostats.txt
			(Field 11 (weighted # of milli-seconds spent doing I/Os) in /proc/diskstat) / (total time)	%	
	temperature	temp per core	libsensors	°c	
	network	network throughput	(bytes_recv + bytes_trans) / seconds (/proc/net/dev)	Bytes/s	
	power	power	ptop (include CPU, memory, disk, wireless network)	milli-watt	
		power	external power measurement hardware	milli-watt	https://www.tindie.com/products/BayLibre/acme-power-measurement-kit/

Table 7: List of metrics for accelerated GPU-based devices

category		metrics	methodology	unit	remarks
Application-level	performance	execution time	int clock_gettime(CLOCK_REALTIME, struct timespec *tp)	ns	
Infrastructure-level	resources utilization	GPU utilization	utilization.gpu / 100 (nvmlDeviceGetUtilizationRates (nvmlDevice_t device, nvmlUtilization_t *utilization))	%	Percent of time **over the past sampling period** during which one or more kernels was executing on the GPU.
		GPU mem accessing rate	utilization.memory / 100 (nvmlDeviceGetUtilizationRates (nvmlDevice_t device, nvmlUtilization_t *utilization))	%	Percent of time **over the past sampling period** during which global (device) memory was being read or written.
		GPU memory total	Memory.total (nvmlDeviceGetMemoryInfo (nvmlDevice_t device, nvmlMemory_t *Memory))	bytes	
		GPU memory used	Memory.used (nvmlDeviceGetMemoryInfo (nvmlDevice_t device, nvmlMemory_t *Memory))	bytes	
	IO	PCIe transmit throughput	value / 0.020 (nvmlDeviceGetPcieThroughput (nvmlDevice_t device, NVML_PCIE_UTIL_TX_BYTES, unsigned int *value))	bytes/s	return bytes transmitted in 20 ms; transform to bytes/s
		PCIe receive throughput	value / 0.020 (nvmlDeviceGetPcieThroughput (nvmlDevice_t device, NVML_PCIE_UTIL_RX_BYTES, unsigned int *value))	bytes/s	return bytes received in 20 ms; transform to bytes/s
	temperature	GPU temp	nvmlDeviceGetTemperature (nvmlDevice_t device, NVML_TEMPERATURE_GPU, unsigned int *temp)	°c	
	power	GPU power (for entire board)	nvmlDeviceGetPowerUsage (nvmlDevice_t device, unsigned int *power)	milliwatt	power for entire GPU board
		total power	ptop (power for CPU board) + GPU power	milliwatt	

Table 8: List of metrics for reconfigurable FPGA-based devices

category			metrics	methodology	unit	reference
PS (Processing System)	Application-level	performance	execution time	int clock_gettime (CLOCK_REALTIME, struct timespec *tp)	ns	
			CPU execution time	int clock_gettime (CLOCK_PROCESS_CPUTIME_ID, struct timespec *tp)	ns	
		resources utilization	CPU utilization	(process cpu time (/proc/[pid]/stat)) / (global cpu time (/proc/stat))	%	http://stackoverflow.com/questions/1420426/calculating-cpu-usage-of-a-process-in-linux
			RAM utilization	VmRSS (/proc/[pid]/status) / Mem Total (/proc/meminfo)	%	
			swap utilization	VmSwap (/proc/[pid]/status) /Swap Total (/proc/meminfo)	%	
			virtual memory size	VmSize (/proc/[pid]/status)	KB	
		IO	disk IO throughput	(read_bytes + write_bytes) / seconds (/proc/[pid]/io)	Bytes/s	since kernel 2.6.20
		power	power (with given pid)	ptop (include CPU, memory, disk, wireless network)	milli-watt	
	Infrastructure-level	performance	floating point instructions per second	PAPIF_flips	Mflip/s	PAPI high-level API
			floating point operations per second	PAPIF_flops	Mflop/s	
		resources utilization	CPU utilization	total_cpu_time – idle_time / total_cpu_time (proc/stat)	%	https://github.com/Leo-G/DevopsWiki/wiki/How-Linux-CPU-Usage-Time-and-Percentage-is-calculated
			RAM utilization	(MemTotal – MemFree) / MemTotal (proc/meminfo)	%	http://www.computerworld.com/article/2722141/it-management/making-sense-of-memory-usage-on-linux.html
			swap utilization	(SwapTotal – SwapFree) / SwapTotal (proc/meminfo)	%	
		IO	disk IO utilization	(Field 10 (# of milliseconds spent doing I/Os) in /proc/diskstat) / (total time)	%	https://www.kernel.org/doc/Documentation/iostats.txt
				(Field 11 (weighted # of ms spent doing I/Os) in /proc/diskstat) / (total time)	%	
		temperature	temp per core	libsensors	°c	
		network	network throughput	(bytes_recv + bytes_trans) / seconds (/proc/net/dev)	Bytes/s	
		power	power	ptop (include CPU, memory, disk, wireless network)	milli-watt	
PL (Programmable Logic)	Infrastructure-level	IO	read/ write transactions	AXI performance monitor	number of requests	
			read/ write latency (avg)	AXI performance monitor		
			read/ write latency (std. Dev)	AXI performance monitor		
			read/ write throughput	AXI performance monitor	MB/s	
PS + PL	infrastructure-level	power	total power	via UCD9248	milli-watt	http://www.wiki.xilinx.com/Zynq+Power+Management

APPENDIX 2. MONITORING CLIENT'S PLUG-INS

Currently, the Monitoring Client supports 7 plug-ins, whose implementation and design details are collected in the directory src/plugins. The monitoring client is designed to be pluggable. Loading a plug-in means starting a thread for the specific plug-in based on the users' configuration at run-time. The folder src/agent plays a role as the main control unit, as managing various plug-ins with the help of pthreads. Folders like src/core, src/parser, and src/publisher are used by the main controller for accessorial support, including parsing input configuration file (src/mf_config.ini), publishing metrics via HTTP, and so on.

Board_power plug-in

This plug-in is dedicated to collect power metrics sampled by an external ACME power measurement kit for a target platform. The ACME power measurement kit can be seen as a compact solution composed of both the hardware and software (http://baylibre.com/acme/). From hardware aspect, there are two key components: a BeagleBone Black board as the main processing unit and an ACME cape as the extension for multi-channel power measurements. The ACME software suite contains an iio-daemon, which reads power measurements from the IIO (Industrial I/O) devices and sends the data out continuously via Ethernet (http://wiki.baylibre.com/doku.php?id=acme:start).

As shown in Figure 20, the Board_power plug-in runs on a hosting platform with the ACME power measurement kit being accessible through given IP address. The target platform, whose power consumptions are interested, is connected with the ACME power measurement kit by the supported power probe. As the ACME power probe is standard, the plug-in and ACME power measurement kit could be used as a generic solution for all possible platforms.

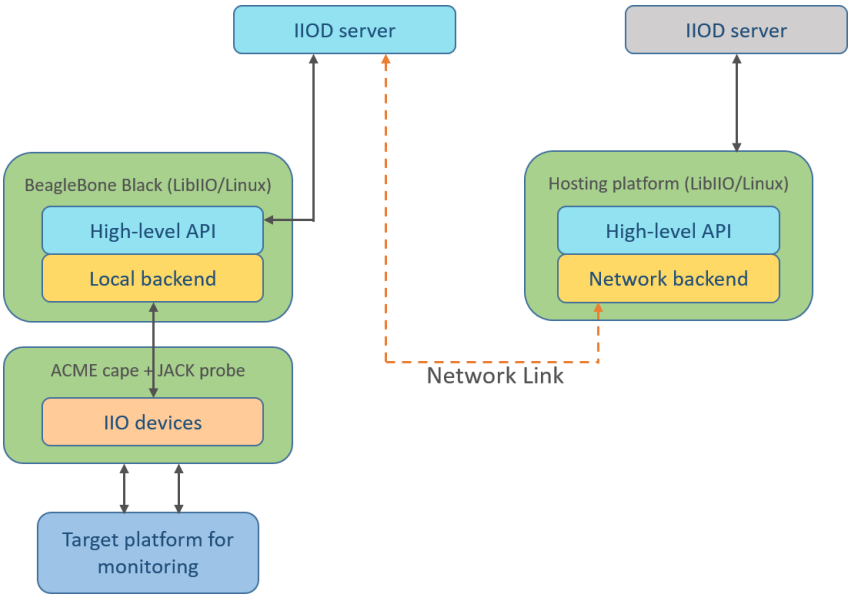


Figure 20: Connection between ACME power measurement kit and the monitoring platform

We use Libiio library for the plug-in's implementation (https://wiki.analog.com/resources/tools-software/linux-software/libiio). It is a library that has been developed by Analog Devices to ease the development of software interfacing Linux IIO devices. Since the iio-daemon running on the ACME kit supports network backends, we use the Libiio APIs firstly to create a network context, before filling IIO buffers and filtering the associated metrics.

CPU_perf plug-in

This plug-in measures performance-related metrics of CPUs in a fine granularity – per CPU core. For each CPU core, the floating-point operations per second, floating-point instructions per second, and total instructions per second are sampled and collected.

The implementation of the plug-in depends on essentially the PAPI hardware counters and PAPI library (http://icl.utk.edu/papi/) (http://icl.cs.utk.edu/projects/papi/wiki/Introduction_to_PAPI-C). It is advisory to check at first if the required PAPI events are supported on the target platform, which are PAPI_FP_OPS, PAPI_FP_INS, and PAPI_TOT_INS respectively.

CPU_temperature plug-in

The target of this plug-in is CPU's temperature per CPU core. To achieve this, the lm_sensors (Linux monitoring sensors) library is used (https://wiki.archlinux.org/index.php/lm_sensors). It is a free and open-source tool which provides interfaces for monitoring CPU's temperature and thermal properties.

Linux_resources plug-in

This plug-in is implemented to retrieve run-time information about Linux kernel and processes by using Linux provided /proc file system. The metrics supported include the CPU utilization percentage, memory utilization percentage, disk I/O statistics and network statistics.

We calculates the CPU utilization rate by dividing the CPU usage time by the total CPU time, both of which can be derived from the file /proc/stat (https://github.com/Leo-G/DevopsWiki/wiki/How-Linux-CPU-Usage-Time-and-Percentage-is-calculated). Because that the values read directly from /proc/stat are time in cycles since system boot, it is thus necessary to calculate at first the time passed by during an interval before calculating the CPU utilization rate.

For memory monitoring, we read from /proc/meminfo the current total available physical memory size, the unused physical memory size, the total amount of swap available, and the total amount of swap free, in order to calculate the RAM usage rate and the swap usage rate at the current time point (http://www.computerworld.com/article/2722141/it-management/making-sense-of-memory-usage-on-linux.html).

The total system disk I/O read and write bytes are derived by adding all read/write bytes per process during the sampling interval, which can be read from the /proc/[pid]/io files

(https://www.kernel.org/doc/Documentation/iostats.txt). Then the system I/O throughput can be retrieved by dividing the total amount of read/write by the time interval. Similarly, we calculate the network throughput based on statistics read from file /proc/net/dev (http://man7.org/linux/man-pages/man5/proc.5.html).

Linux_sys_power plug-in

With the help of Linux kernel and /proc file system, we implement this plug-in, which is capable of power monitoring of various system components, including CPU, memory, disk I/O, and wireless network.

The CPU power consumption is estimated by using a Linux module named as cpufreq-stats (http://lxr.free-electrons.com/source/Documentation/cpu-freq/cpufreq-stats.txt) (https://www.kernel.org/doc/Documentation/cpu-freq/cpufreq-stats.txt). It is a driver that provides CPU frequency statistics for each CPU through its interface, which appears normally in the directory /sysfs/devices/system/cpu/cpuX/cpufreq/stats. By reading the values kept in the file time_in_state, we retrieve the amount of time spent in each of the frequencies supported by the dedicated CPU. We assume that the relationship between CPU frequency and power consumption is a linear correlation, consequently it is feasible to estimate the average CPU power during an interval with given minimum and maximum CPU power specifications.

For memory power estimation, we uses the system call "__NR_perf_event_open" to stat the hardware cache misses (http://man7.org/linux/man-pages/man2/perf_event_open.2.html). Together with reading the disk I/O read/write statistics, we calculate the memory power consumption with the following formula. The L2 cache miss latency and L2 cache line size can be obtained via some known calibrator (http://homepages.cwi.nl/~manegold/Calibrator/calibrator.shtml).

$$\frac{\left(\frac{IO_{read}+IO_{write}}{L2CacheLineSize}+MemAccess\right)\times L2CacheMissLatency\times MemPower}{Sample\ Interval}$$

Disk and wireless network power consumptions are calculated based on their activities, like read/write and receive/send bytes during the sampling interval. As long as the energy specifications of the disk and wireless network card are given, we could compute the constants, like energy cost per disk read/write and energy cost per wireless network receive/send, and get finally the energy consumed during a specific period.

Our implementation is based on the methodology proposed by the pTop project. Please refer to the project web page for more details and information. (http://mist.cs.wayne.edu/ptop.html) (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.360.7151)

NVML plug-in

The target of this plug-in is the Nvidia GPU, which is normally hosted by a connected CPU. The plug-in, which runs on the hosting platform, uses the NVML (Nvidia management library) provided C-based APIs to monitor the associated GPU devices'

properties (https://developer.nvidia.com/nvidia-management-library-nvml). The metrics supported by this plug-in, including GPU usage rate, GPU memory usage rate, PCIe throughput, GPU temperature and GPU power consumption, depend on the models of the GPU devices. For example, PCIe statistics are available only for Maxwell or newer architectures (https://docs.nvidia.com/deploy/pdf/NVML_API_Reference_Guide.pdf).

RAPL_power plug-in

For Intel CPU an alternative method for power monitoring is by using this plug-in, which is implemented with the help of RAPL (Running Average Power Limit) provided energy and power information. RAPL, as clarified in the reference, (https://01.org/zh/blogs/2014/running-average-power-limit-%E2%80%93-rapl?langredirect=1) is not an analog power meter, but rather uses a software power model to estimate energy usage by using hardware performance counters and I/O models.

In general cases, RAPL domains cover both the CPU package (including core and uncore devices) and the DRAM, as can be seen from Figure 21. However the specific RAPL domains available in a platform vary across product segments.

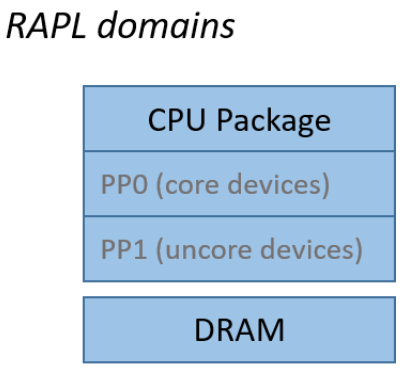


Figure 21: RAPL power measurements domains

APPENDIX 3. MOVIDIUS BOARD INTEGRATION IN MONITORING FRAMEWORK

Myriad2 general information

A high level diagram of the Myriad2 platform architecture is shown in Figure 22. On the Myriad2 platform, there are twelve 128-bit vector-processors units (SHAVE processors), SIPP Hardware Accelerators, 2MByte on-chip SRAM (CMX), 64-bit interface to DDR2/3 RAM running at up to 1033MHz and a range of other peripherals. Besides, the two Leon4 RISC processors are used to manage execution: the RISC1 core is designed to be real-time controller for scheduling the activity of Myriad2 while RISC2 core is designed to run an operating system such as Linux or RTEMS and to manage all the control interfaces. Each of the LEON4 cores also includes a fully IEEE754 compliant FPU with fp64 support which allows the platform to provide native fp64 support either standalone, or accelerated by the 12 on-board SHAVE processors which natively support fp16 or fp32 but not fp64. Each SHAVE is connected to the 256kB 2-way L2 Cache and has 2 x 64-bit data ports to CMX memory as well as a 128-bit instruction port. The SIPP Hardware Accelerators also have 16 x 2 x 64-bit data connections into the CMX at the nominal system clock frequency of up to 800MHz, delivering an aggregate 400GBytes/s of total bandwidth, necessary for sustained high performance for many numerical applications.

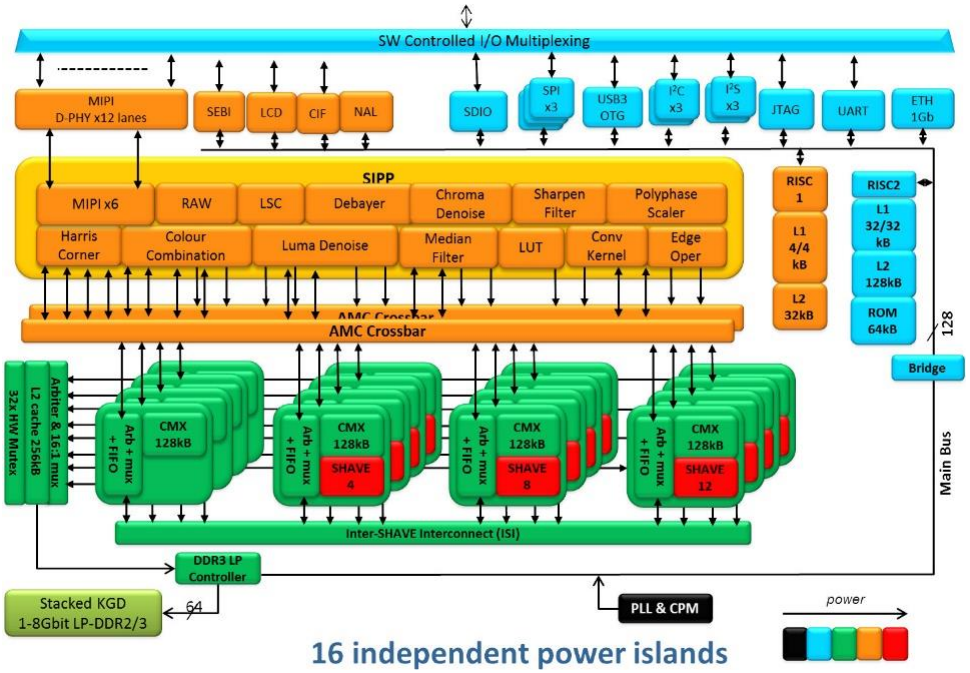


Figure 22: Hardware architecture and components of Myriad2 platform

Myriad2 software development

Software development tools

The Myriad2 software development environment is shown in Figure 23. Included in the MoviTools are its compiler, debugger, simulator, and so on, all developed by Movidius, as supporting Myriad2 various processors. The Movidius compiler now supports fully the C/C++ language with various libraries included. The operating system on Myriad2 is RTEMS, which provides a multi-threaded, multi-tasking environment for application's threads sharing the same memory space. Within RTEMS each subsystem is implemented as an independent manager (task, interrupt, clock, semaphore …). In the system, required managers and applications are linked to one binary image. The system includes also TCP/IP networking and local file-systems support.

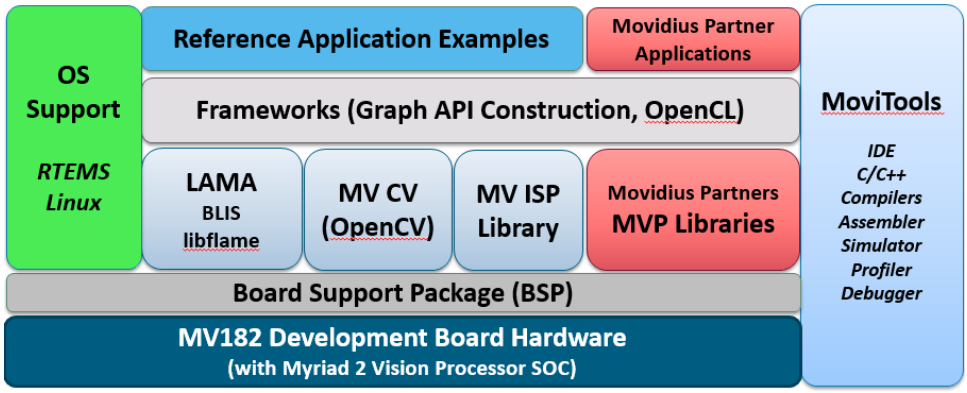


Figure 23: Myriad2 software development tools and environment

Programming paradigms

According to different hardware and operating system's availability, the programming paradigms of Myriad2 platform can be classified into three types, shown in Figure xx-xx respectively as follows.

Standard programming paradigm

The standard programming paradigm for Myriad2 involves using RTEMS running on LeonOS and the SIPP scheduler on LeonRT. The advantage of this paradigm is that it provides parallelization in an easy to use environment. The SIPP scheduler itself is able to ensure parallel pipeline configurations for managing the HW filters and exterior interfaces with a low footprint so as to ensure LeonRT optimized utilization. The SIPP used number of SHAVEs is configurable, so any extra number of SHAVEs not used for line based pipelines will remain free to be used by the RTEMS operating system running on LeonOS for various other purposes including (but not limited to) computer vision algorithms.

One Leon programming paradigm

Some applications might not require heavy line based processing. Such applications might choose to completely switch OFF the LeonRT processor and instead only use LeonOS with (or without) RTEMS. HW filters may still be used. Using this programming paradigm, as shown in Figure 24, LeonOS would control all of the applications running on the 12 SHAVE cores.

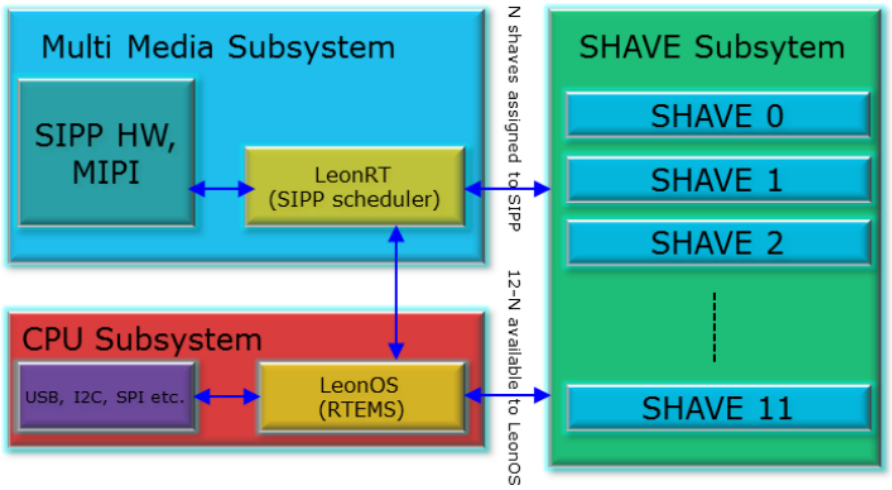


Figure 24: Myriad2 standard programming paradigm

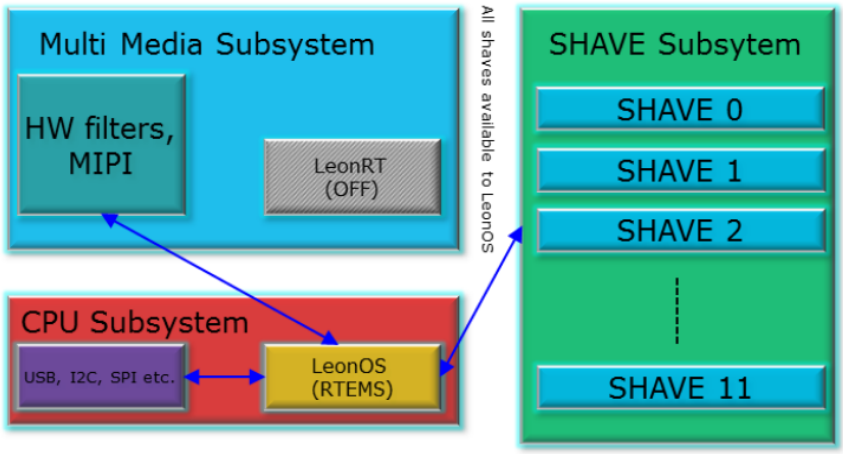


Figure 25: Myriad2 one Leon programming paradigm

Bare metal programming paradigm

A bare metal programming paradigm (cf. Figure 26) will also be supported by the MDK build system. This will allow developer to use both LEON cores without any operating system, only minimal schedulers running to control the pipelines application. This paradigm requires more integration efforts but allows developers to write applications which will not be affected by any operating system overhead.

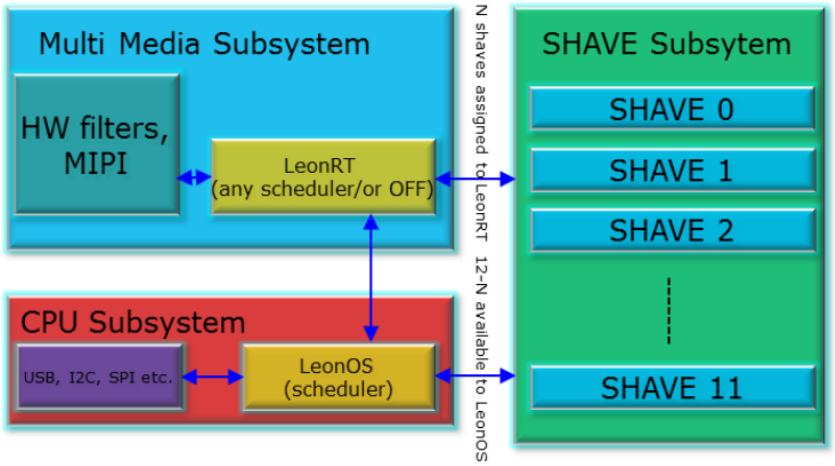


Figure 26: Myriad2 bare metal programming paradigm

According to users' requirements and sensors availability on Myriad2, we implement several functions for temperature and power monitoring and provide APIs for code instrumentation of a generic Myriad2 application.

The board-specific metrics that are supported by the Monitoring Framework are listed in Table 9.

Table 9: Metrics supported for Myriad2 platform

Type	Metrics	Units	Description
Power	power_core	mW	Power consumption of the cores and processors
	power_ddr	mW	Power consumption of DDR memory
Temperature	temperature_CSS	°c	Temperature of the CPU Sub System (CSS), equals to the Leon RSIC2 processor
	temperature_MSS	°c	Temperature of the Media Sub System (MSS), equals to the Leon RSIC1 processor
	temperature_UPA0	°c	Average temperature of half of the Microprocessor Array (6 VLIW SHAVE vector processors)
	temperature_UPA1	°c	Average temperature of the other half of the Microprocessor Array (the other 6 VLIW SHAVE vector processors)

The power monitoring is achieved based on the MV0198 power measurement daughtercard, which is composed of 4 ADCs and samples 13 power rails and 2 voltage rails of the Myriad2 motherboard MV0182. With the enabled I2C interface to the motherboard

and drivers provided APIs for reading the measurements, we implement functions in C to initialize the MV0198 driver and gather measurements periodically.

For temperature monitoring, we uses the temperature sensor library provided in the Leon RSIC2 operating system. As can be seen from **Table 2**, temperature metrics are divided into four parts, which can be mapped to accordingly different hardware components.

For users convenience, we design and implement the application-level APIs for Myriad2 platform same as these for other platforms. Listing below gives an example shows how to use these APIs in a Myriad2 application.

Listing : Example of a Myriad2 application with integrated monitoring APIs

```
void POSIX_Init(void *args)
{
UNUSED(args);
/* NEED FOR USING ETHERNET */
initClocksAndMemory();
EthPHYHWReset();
InitGpioEth(INVERT_GTX_CLK_CFG);
initGrethAndNet();
/* SETUP METRICS */
metrics m_resources;
m_resources.num_metrics = 2;
m_resources.local_data_storage = 0;    // TOD: local data storage
m_resources.sampling_interval[0] = 1000;       // 1s
strcpy(m_resources.metrics_names[0], "power_monitor");
m_resources.sampling_interval[1] = 1500;       // 1.5s
strcpy(m_resources.metrics_names[1], "temp_monitor");
/* START MONITORING */
mf_start("141.58.0.8", "movidius", &m_resources);
/* DO THE WORK */
sleep(15);
/* STOP MONITORING */
mf_end();
exit(0);
}
```

Function “mf_start” creates threads for monitoring power and/or temperature, which are configured by setting the metrics’ names and sampling intervals. Each thread runs in a loop, samples the specialized metrics, and uses the board’s network stack to send the sampled metrics to the server. After execution of the target code block, “mf_end” terminates the running threads and ends accordingly the metrics sampling process. It is noted that the monitoring data are sent to the server via HTTP and hardware sockets, therefore programmers should initialize and configure the system’s Ethernet module properly before using the provided APIs.