D6.1 OpenStack Support for UniServer

Contract number	688540
Project website	http://www.uniserver2020.eu
Contractual deadline	Project Month 18 (M18): 31st July 2017
Actual Delivery Date	31th July 2017
Dissemination level	Public
Report Version	1.0
Main Authors	Mustafa Rafique (IBM), Srikumar Venugopal (IBM), Bin Wang (QUB), Christos D. Antonopoulos (UTH), Chris Kalogirou (UTH)
Contributors	
Reviewers	Peter Lawthers (APM), Alejandro Lampropulos (WSE), Georgios Karakonstantis (QUB)
Keywords	OpenStack, X-Gene, Hypervisor, System Software Interface

Notice: The research leading to these results has received funding from the European Community's Horizon 2020 Programme for Research and Technical development under grant agreement no. 688540.

Acknowledgements
The work presented in this document has been conducted in the context of the EU Horizon 2020. UniServer is a 36-month project that started on February 1st, 2016 and is funded by the European Commission. The partners in the project are:

The Queen’s University of Belfast (QUB)

The University of Cyprus (UCY)

The University of Athens (UoA)

Applied Micro Circuits Corporation Deutschland Gmbh (APM)

ARM Holdings UK (ARM)

IBM Ireland Limited (IBM)

University of Thessaly (UTH)

Worldsensing (WSE)

Meritorius Audit Limited (MER)

Sparsity SL (SPA)

More information

Public UniServer reports and other information pertaining to the project are available through the UniServer public Web site under http://www.uniserver2020.eu.

Change Log

Version	Description of change
0.1	Initial draft
0.2	Added Ceilometer and Nova details
0.3	Added Ceilometer enhancements
0.4	Added Nova enhancements
0.5	Added results from applications
0.8	Added introduction and executive summary
0.9	Added contributions from UTH
1.0	Addressed reviewer’s comments

Table of Contents

Index of Figures

Index of Tables

Executive Summary

This document describes the OpenStack support for UniServer on Merlin-based X-Gene2 boards as developed in Task 6.1 within the Work Package 6 (WP6) of the UniServer Project Description of Action (DoA). This is in fulfillment of Deliverable D6.1, OpenStack Support for UniServer.

OpenStack is a middleware for cloud, which runs on multiple servers (nodes) that are part of a cloud cluster. It is primarily used on x86 based machines, and has recently been made available for 64-bit ARM micro-servers. However, in order to meet the unique requirements of UniServer project, special enhancements and extensions are required in the current state of the art OpenStack. In particular extensions are needed to get the enhanced telemetry data, run statistical analysis on the collected data and to get the characteristics of the running virtual machines (VMs) in the cloud data centers. The work done in this deliverable would be used in the other tasks of WP6, especially for the development of workload scheduler for UniServer and fault tolerance techniques to improve the resilience of cloud data center.

1. Introduction

One of the main use cases and objectives of UniServer project is to run the target applications in a cloud platform with improved energy efficiency while meeting the performance requirements of the target applications. Several open source cloud platforms middleware software are available, such as OpenShift [1], Cloudify [2], Cloud Foundry [3], etc., however, OpenStack [4] is considered as the de facto standard and a solution of choice for cloud providers. To this end, Work Package 6 (WP6) aims at providing enhancements and specialized resource management policies for OpenStack running on 64-bit ARM based micro-servers. WP6 aims at improving the management of virtual machines (VMs) which are running on nodes with heterogeneous power, and performance settings. This heterogeneity in power and performance adds an additional parameter, i.e. reliability, that should be incorporated in managing the running VMs thus requires developing techniques for improving the resilience of cloud infrastructure and the running workloads.

OpenStack is an open source software made available under the Apache 2.0 license, and is managed by the OpenStack Foundation, a non-profit organization that oversees the software development as well as the community around the project. Currently, around 150 organisations have contributed to OpenStack in some form or shape. The presence of a large and active community was one of the drivers for the adoption of OpenStack as the compute infrastructure management platform for the UniServer project.

In this deliverable (D6.1), we present the enhancements and extensions to the OpenStack framework [4] that are introduced in off-the-shelf OpenStack for meeting the objectives of UniServer project. These enhancements and extensions are developed and evaluated using X-Gene2 Merlin boards. The goal of these enhancements and extensions is to demonstrate that the OpenStack framework can be used to fully utilize the exposed extended margins from the underlying micro-servers in cloud deployments. The exposed extended margins are available to OpenStack through the Libvirt [5] interfaces. In order to support the extended Libvirt interfaces, extensions are required in the telemetry component of the OpenStack framework to collect and store the fine-grain monitoring data in the persistent database. Furthermore, in order to make efficient and effective resource management decision, it is required to statistically analyse historic resource utilization data of the running VMs. As part of this deliverable, we provide extensions to the telemetry service of the OpenStack framework to collect new metrics as collected by the UniServer hypervisor, as well as addition of a new module in the compute component of the OpenStack framework to run the statistical analysis on the historic resource utilization data.

The enhancements and extensions that are introduced in the OpenStack as part of this deliverable would be used in developing advanced scheduling algorithm (D6.2), resource manager (D6.3), and proactive and reactive resilience techniques (D6.5) for UniServer. The advanced scheduler and resource manager for UniServer would use the enhanced telemetry data in determining the power, performance, and reliability of the available node, and then determine the appropriate node for running the given VM. The resilience techniques for UniServer would incorporate these enhancements and extensions in determining the potential node failure and to trigger the migration process of a VM in the case of a node failure.

1.1. Organization

The rest of this deliverable is organized as follows. Section 2 presents the requirement for UniServer from data center perspective and provide a background of OpenStack and its connection with the hypervisor. Section 3 details extensions to OpenStack for UniServer, specifically in the telemetry and the compute components of the OpenStack, and present results of determining the characteristics of the target applications while they are running on OpenStack cloud hosted on Merlin X-Gene2 boards, which are 64-bit ARM based micro-servers. Finally, Section 4 provides details of our OpenStack deployment on Merlin X-Gene2 boards.

2. UniServer Data Center Management Requirements

2.1. UniServer in the Data Center

One of the UniServer project's primary objectives is to improve the power efficiency of data centers by taking advantage of the extended margins of the processors to drive down micro-servers into a low voltage, low power state. One of the primary consumers of power in a data center is the cooling system for the servers [6]. By running the servers in a low-voltage configuration, UniServer aims to reduce the stress on the cooling system, thereby improving power usage effectiveness in the data center.

However, these advantages are accompanied by additional requirements at the system software level for dealing with cloud infrastructure whose individual components are at higher risk of malfunction while running at extended margins [7]. Customers deploy their applications within virtual machines that are deployed to physical machines in the data center. For data center operators to make hardware configuration decisions without affecting such workloads, the system software will be engineered such that it takes care of any faults transparently without requiring customer applications to be re-engineered. This puts the onus on the data center management software to be able to predict failures and detect anomalies, and migrate workloads in order to protect them from failures.

In the following sections, we describe the design and development of mechanisms to meet this requirement at the operating system, the hypervisor, and the data center levels. OpenStack was chosen as the management software due to its open-source nature and its deployment in data centers across the globe. At the hypervisor level, we describe facilities for providing detailed information about the power and performance of the underlying processors. We also describe the efforts made at the operating system level to isolate unreliable domains in the main memory. Then, we describe the extensions made to OpenStack in order to incorporate this extended information.

2.1.1. Hypervisor and OpenStack

Hypervisor, i.e. KVM [8] in the context of UniServer, is a module within the kernel space along with other modules of the operating system such as the scheduler, governors, cgroups etc. which are used to manage and direct the overall system operation. The hypervisor is responsible for creating and running one or more VMs on the guest machines (i.e., in the user space). In a typical operating system, memory is divided into at least two distinct accessible regions: user space and kernel space. The user space, is a set of locations where normal user processes, i.e. everything other than the kernel, run. The role of the operating system kernel is to manage applications running in this space from interfering with each other, and the machine. The kernel space, is the location where the code of the kernel is stored and executed. Processes running under the user space have access only to a limited part of the memory, whereas the kernel has access to all the memory. Processes running in user space also do not have access to the kernel space. User space processes can only access a small part of the kernel via an interface exposed by the kernel through the system calls**.**

Shown in Figure 1, at the hypervisor layer, hardware metrics related mainly to power and performance (such as CPU and memory utilization, cache misses etc.) are already monitored. by an existing application programming interface (API). In UniServer we plan to enhance such an API to make it able to collect and monitor information related to reliability by interacting with UniServer specific daemons. For example, CPU errors could be acquired from the HealthLog module through new functions.

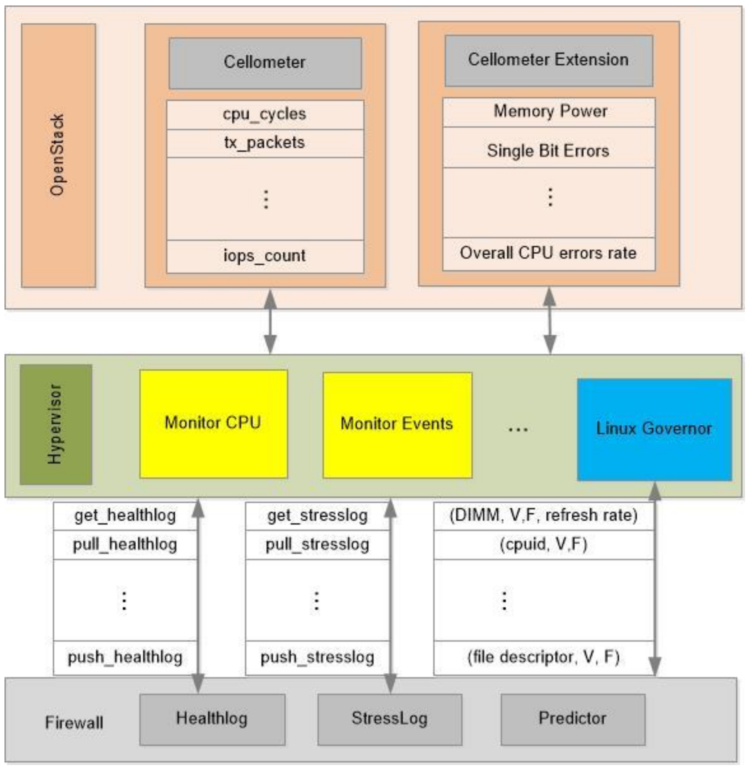

Figure 1: Relationship between hypervisor and OpenStack in UniServer Module

Going further-up to the OpenStack, which is another guest layer for the resource management, it extracts information for every VM and the overall system using Libvirt [5]. Libvirt is a hypervisor-independent virtualization API and toolkit that interacts with the virtualization capabilities of a range of operating systems. Essentially, Libvirt is an intra-node manager that will be responsible for the communication between the OpenStack and the hypervisor in UniServer. Within the OpenStack, different components, such as Ceilometer, Nova, etc. talks with the Libvirt interface, which collects information and distributes it to other OpenStack components as required.

2.2. OpenStack Framework Overview

2.2.1. OpenStack

OpenStack is a collection of software components that enable the management of collections of compute, storage and network resources so that these can be virtualized and made available to end users following the cloud computing paradigm. That is, OpenStack enables the Infrastructure as a Service (IaaS) model by provisioning and managing the underlying hardware such that users can obtain on-demand virtual machines with specific requirements for processing, memory and storage.

Briefly, the primary components of OpenStack are as below:

1. **Nova:** responsible for provisioning and management of virtual machines on compute nodes
2. **Glance:** stores the images that are used as the disk templates for booting VMs
3. **Keystone:** manages authentication and authorisation for invoking OpenStack commands
4. **Swift:** is an object storage system wherein uniquely identified data items can be manipulated independent of their location
5. **Cinder:** allocates filesystem blocks that can be attached to VMs as extended disk storage
6. **Neutron:** provides networking services for OpenStack components as well as VMs

7. **Ceilometer:** provides telemetry services to keep track of resource usage and performance
8. **Horizon:** is a web-based user interface to interact with an OpenStack installation

The UniServer project focuses on enabling energy efficient yet high performance micro servers in the data centers to enable a seamless virtualised environment that spans cloud as well as edge computing environments. UniServer aims to achieve its objectives by exposing more detailed information about a physical machine as well as provide dynamic information about the machine's health through an extended HEI (Hardware Exposure Interface). This extended information is aimed to be used for dynamic scheduling of VMs in order to achieve energy efficiency in the data center. As such, the focus of the UniServer project and Work Package 6 within it, is on two important components out of the eight listed above: Ceilometer, for collecting the extended information exposed by the UniServer machines, and Nova, for supporting dynamic scheduling and management of nodes using this additional information.

The next two sections will discuss Ceilometer and Nova in detail followed by the discussion on our extensions to these two components to realise UniServer objectives.

2.2.2. OpenStack Telemetry (Ceilometer)

Figure 2 shows the data collection, storage and usage workflow of Ceilometer [9]. The data is first collected from the physical and virtual resources using APIs exposed by the relevant system resource and components either through polling or notification agents. The data can also be sent directly into Ceilometer system by other OpenStack components (e.g., Nova, Glance, Neutron, etc.) by adding a message to the notification bus. The collected sample from a meter can then be transformed into more meters if required. After the transformation, the data is sent to one or more publisher (e.g., Gnocchi, Oslo messaging service, etc.), where each publisher saves the data into persistent storage, typically in a database, through the message bus or dispatch it to the external system for consumption. Finally, the stored data can be read by OpenStack services or any other external entity through REST APIs exposed by the Ceilometer.

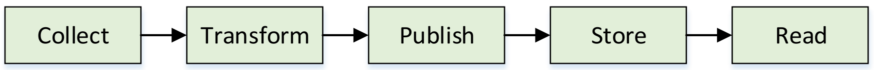

Figure 2: Ceilometer Data Collection Workflow

Figure 3 shows the interaction between different Ceilometer components. Ceilometer also provides two agents, namely notification agent and polling agent, to collect data about physical and virtual resources of the datacenter. OpenStack components inserts the notifications in the notification bus, which is picked by the notification agent for processing. The notification agent converts the notifications to events and samples and applies preconfigured publishing pipeline and transformation rules that are specified in `pipeline.yaml` and `event_pipeline.yaml` configuration files in YAML (Yet Another Markup Language) format. The polling agents provide daemons that periodically poll the data from other OpenStack services and external system components (e.g., hypervisor). There are three pollster plugins, i.e. compute, central, and IPMI, used by a typical deployment. Finally, the collector stores the samples in the database for persistent storage and can also send them to the external systems for storage and monitoring.

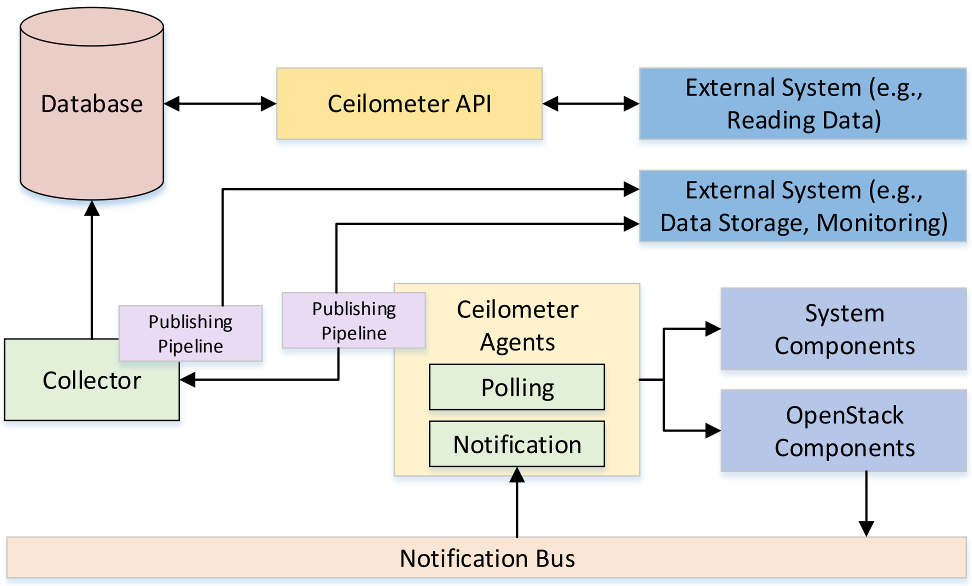

Figure 3: System Architecture and Interaction between Ceilometer Components

There are several built-in meters that are available to the Ceilometer. These meters report data about physical and virtual resources that are available in the data centers. Logically, the predefined meters can be organized into different system components, e.g., IPMI, SNMP, and SDN controllers etc., and OpenStack components, e.g., compute, image, block and object storage, and networking components, etc. The complete list of existing meters that are currently available can be found at OpenStack Administrator Guide [10].

2.2.3. OpenStack Compute (Nova)

Figure 4 shows the architecture and interaction between different components of OpenStack Nova module. The API module receives the requests from the user and adds them for processing in the queue. It also receives the results back from the queue and sends them to the user. The Scheduler service in Nova determines how the requests should be processed. One of its important tasks is to determine which physical host the given VM should be executed on. Console and ConsoleAuth services in Nova provide remote console or remote desktop access to the running VMs. The Certificate service generates certificates for euca-bundle-image for EC2 compatible API, and is not required in deployment where images are not required to be bundled for EC2. The Conductor service in Nova isolates the database functionalities for the compute nodes, and enables them to function without accessing the database. The Compute service manages the physical hosts for running the VMs, and controls the communication between the VMs and the hypervisor through hypervisor

APIs, such as, Libvirt API [5], Xen API [11], and VMware API [12].

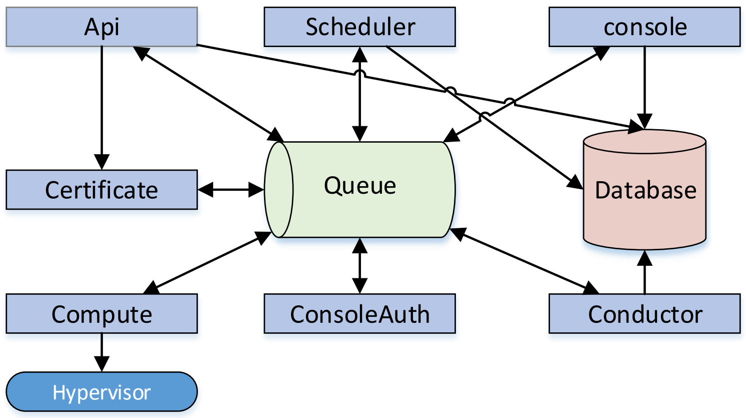

Figure 4: System Architecture and Interaction between Nova Components

3. OpenStack Enhancements for UniServer

As discussed in the previous sections, the enhancements to OpenStack are critical for obtaining the information required by the cloud middleware to improve the performance, reliability, and the energy consumption of the data center. To this end, the two components in OpenStack that are of particular interest for UniServer are the telemetry (Ceilometer) and compute (Nova). Ceilometer provides detailed performance and utilization measurements that are related to the physical and virtual resource, while Nova provides the computing fabric controller for the cloud and manages the virtual machines running in the data center. The following subsections describe the enhancements made in Ceilometer and Nova components of OpenStack for meeting the objectives of UniServer.

3.1. Extensions to Ceilometer

Currently, the monitoring information reported by the Libvirt and hypervisor to the Ceilometer is focused mainly towards the virtual resources. However, because of having reliable and unreliable servers operating in a single cloud setup, it is required for the UniServer project to gather information about physical resources pertaining to the health of the underlying servers. In Ceilometer, node level measurements are gathered through IPMI inspector and pollster, however, these measurements are primarily focused towards OpenStack bare metal (Ironic) services. Originally, the IPMI was a part of Ceilometer, but now it has been moved to Ironic code base, making it specific to bare metal services. Furthermore, existing meters provided in the OpenStack for underlying resources do not collect data for the metrics, such as correctable and uncorrectable errors in memory, caches and MCUs, which are required in the UniServer project for determining the health of the underlying server. To close the gap between the information currently being gathered by Ceilometer, and the requirements of the UniServer project, we extended Ceilometer by adding new meters in it.

As discussed in the previous sections, Ceilometer component in OpenStack gathers various data about the health and performance of the underlying physical and virtual resources. In the context of UniServer, Figure 6 shows the flow of monitoring data from Hardware-Exposure-Interface (HEI) to OpenStack. OpenStack Ceilometer service collects the data from the hypervisor through Libvirt. The hypervisor gets the information through StressLog and HealthLog daemons which interact with the HEI to get the up-to-date information about the health and performance of resources.

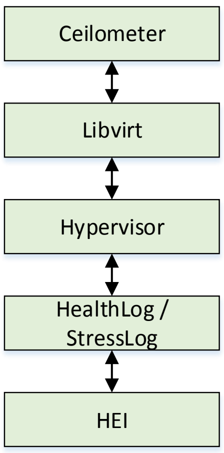

Figure 5: Flow of monitoring data in OpenStack for UniServer

3.1.1. OpenStack Metrics for UniServer

One of the goals of OpenStack extensions for UniServer is to enhance its telemetry service with fine grained and specialized measurements related to the utilization and consumption of the underlying system resources. To this end, UniServer introduces several new measurements to the existing meters available in the Ceilometer. Table 1 provides the list of new meters that are added for UniServer in the Ceilometer component of OpenStack. These meters largely deal with the current state of the system in terms of utilization, temperature, correctable and uncorrectable errors in memory and cache hierarchies. These metrics would be useful in the related tasks of WP6 where we plan to develop scheduling policies for the cloud workloads and resource management policies for improving the resilience and fault tolerance capabilities of the cloud infrastructure. Note that such metrics will be estimated using inputs from the low level metrics that were described in deliverables D3.2 and D3.3.

Table 1: List of Meters for UniServer in OpenStack

Meter	Meter Information
system.cpu.utilization	Total time spent for system by all CPUs in msec
system.memory.free.absolute	Absolute amount of free system memory in KB
system.cpu.power.absolute	Absolute power consumption of all CPUs in Watts
system.memory.power.absolute	Absolute power consumption of system memory in Watts
system.socket.temperature.absolute	Absolute CPU temperature in Celsius
system.dram.errors.correctable.absolute	Absolute amount of correctable DRAM errors
system.dram.errors.uncorrectable.absolute	Absolute amount of uncorrectable DRAM errors
system.mcu.errors.correctable.absolute	Absolute amount of correctable MCU errors
system.mcu.errors.uncorrectable.absolute	Absolute amount of uncorrectable MCU errors
system.l3.errors.correctable.absolute	Absolute amount of correctable L3 cache errors
system.l3.errors.uncorrectable.absolute	Absolute amount of uncorrectable L3 cache errors
system.l2.errors.correctable.absolute	Absolute amount of correctable L2 cache errors
system.l2.errors.uncorrectable.absolute	Absolute amount of uncorrectable L2 cache errors
system.l1.errors.correctable.absolute	Absolute amount of correctable L1 cache errors
system.l1.errors.uncorrectable.absolute	Absolute amount of uncorrectable L1 cache errors
system.dram.errors.correctable.frequency	Frequency of correctable DRAM errors per hour
system.dram.errors.uncorrectable.frequency	Frequency of uncorrectable DRAM errors per hour
system.mcu.errors.correctable.frequency	Frequency of correctable MCU errors per hour
system.mcu.errors.uncorrectable.frequency	Frequency of uncorrectable MCU errors per hour
system.l3.errors.correctable.frequency	Frequency of correctable L3 cache errors per hour
system.l3.errors.uncorrectable.frequency	Frequency of uncorrectable L3 cache errors per hour
system.l2.errors.correctable.frequency	Frequency of correctable L2 cache errors per hour

Meter	Meter Information
system.l2.errors.uncorrectable.frequency	Frequency of uncorrectable L2 cache errors per hour
system.l1.errors.correctable.frequency	Frequency of correctable L1 cache errors per hour
system.l1.errors.uncorrectable.frequency	Frequency of uncorrectable L1 cache errors per hour

3.1.2. Extending Polling Agents

The implementation of adding new meters in the polling agent requires the following:

- Implementation of a new meter in pollster module
- Implementation of wrapper function to use the exposed Libvirt interface in the inspector module of Ceilometer

UniServer uses a hierarchical approach and encapsulates the functionality of adding a new meter in the base class. The base class is extended from the `PollsterBase` object to get the minimum amount functionality and data to implement a new meter. Figure 6 shows the implementation of the base class, which provides the required `get_samples` functionality to the derived classes. It also implements two important properties, namely `inspector` and `default_discovery`, which is required by the polling agent to determine if new samples should be extracted during this polling cycle. It should be noted that `default_discovery`, which is an abstract method from the base class, would return `'local_node'` for the meters which are related to the physical resources of the data centre, whereas it should return `'local_instances'` for the meters that gather data about the virtual instances running in the data centre.

```
class _Base(plugin_base.PollsterBase):

    @property
    def inspector(self):
        try:
            inspector = self._inspector
        except AttributeError:
            inspector = virt_inspector.get_hypervisor_inspector(self.conf)
        return inspector

    def setup_environment(self):
        super(_Base, self).setup_environment()
        self.polling_failures = 0

    @property
    def default_discovery(self):
        return 'local_node'

    @abc.abstractmethod
    def get_value(self, stats):
        """Returns value for a sample."""

    @abc.abstractmethod
    def read_data(self, cache):
        """Returns data sample for meter."""

    def get_samples(self, manager, cache, resources):
        try:
            stats = self.read_data(cache)
        except Exception, e:
            self.polling_failures += 1

        self.polling_failures = 0

        metadata = {'node': self.conf.host }

        if stats:
            data = self.get_value(stats)
            yield sample.Sample(
                name=self.NAME, type=self.TYPE, unit=self.UNIT,
                volume=data, user_id=None, project_id=None,
                resource_id=self.conf.host, resource_metadata=metadata)
```

Figure 6: Implementation of Base Class for Adding Meters

In order to add a new measurement, a specialized class can derive from this base class and override the required class variables and functions as appropriate for the specific meter. Figure 7 shows the implementation of `CPUPowerPollster` meter, which returns the absolute power consumed by a CPU in watts. Typically, there are three types of meters that can be defined in the Ceilometer, i.e., cumulative, gauge, and delta. Since this meter returns the absolute power consumption of the CPU at a given instance, therefore it sets the meter type as ‘`TYPE_GAUGE`’. It is also required to implement the abstract method `read_data` to read the measurement using the implemented Libvirt API in the Libvirt client module of Ceilometer which is returned in the consolidate data structure with different system metrics. Finally, the abstract method `get_value` is implemented to extract and return the specific value of interest that is related to the target meter from the statistics that are collected from the Libvirt client module.

```
class CPUPowerPollster(_Base):
    NAME = "system.cpu.power.absolute"
    TYPE = sample.TYPE_GAUGE
    UNIT = "Watts"

    def read_data(self, cache):
        system_info = self.inspector.inspect_cpu_power()
        return system_info

    def get_value(self, stats):
        return stats[0]
```

Figure 7: Implementation of CPUPowerPollster Meter

3.1.3. Extending Libvirt Ceilometer Client

OpenStack encapsulate all functionalities and interfaces that are related to the Libvirt in a separate client/module for each of its software components, such as, Nova, and Ceilometer. For Ceilometer, all Libvirt functionalities are encapsulated inside the compute mode under virt/Libvirt client, where inspector abstraction is implemented in `LibvirtInspector` module. In order to use the extended Libvirt interfaces that have been proposed in WP5, a new python API should be implemented in this module. Figure 8 shows the implementation of `inspect_cpu_power` function which fetches the current power consumption of the CPU as it is implemented by the extended Libvirt for UniServer by using the local Libvirt connection object. This implementation uses the local connection to the Libvirt daemon to call the specific API to get the broader statistics about the CPU power consumption. The required statistics are then stored and returned in a newly created `PowerStats` named tuple collection for the consumption in the `CPUPowerPollster` object.

```
def inspect_cpu_power(self, instance=None, duration=None):
        stats = self.connection.getPowerUniserver()
        system_stat = stats['CPU']
        if system_stat is None:
            LOG.warning("inspect_cpu_power system_stat is null")
            return virt_inspector.PowerStats(watts=-1)
        return virt_inspector.PowerStats(watts=system_stat[0])
```

Figure 8: Implementation of Libvirt inspect_cpu_power Function

3.2. Extensions to Nova

One of the objectives of WP6 is to develop adaptive scheduling algorithms and resource management techniques for running workloads in the cloud. In order to develop efficient and effective scheduling and resource management techniques, it is important for the cloud middleware, i.e., OpenStack, to know the current characteristics, in terms of resource requirements, of the running VMs. To this end, we have extended OpenStack Compute, i.e., Nova, component [13] with a new VM Characteristics module that returns the characteristics of a specific VM. Moreover, a corresponding API has been added to Nova to access this module. In the context of UniServer project, VM Characteristics module currently focuses on CPU and memory resources, however, this can be further extended to incorporate other resources, such as network and storage.

3.2.1. VM Characteristics Component

VM Characteristics component can be accessd either through command line interface (CLI) or through the

REST API. The VM Characteristics fetches the historic data about the relevant CPU and resident memory consumption metrics from the Ceilometer database and runs linear regression on the retrived data to get the VM resource requirements trend during the specific time window. For UniServer workloads, especially, the jammer detector application and Polaris benchmark that are discussed in the following section, we have found out that an analysis window of 10 minutes gives us a stable and reasonable estimate about a VMs resource characteristics.

Figure 9 shows the architecture of VM Characteristics component with respect to Nova, and how it interacts with other Nova components. User request is captured by Nova API, and then passed on to the compute manager which interacts with the Libvirt driver and calls the `get_characteristics` method and passes on the name of the VM to get its characteristics. The `get_characteristics` method retrieves the historic data through python Ceilometer client and runs the statistical analysis on the retrieved data. The results are then packed inside an object and returned back to the manger. Eventually, the results are transformed and reported back to the user either in a tabular or JSON format based on the original request, i.e., CLI or REST API, respectively.

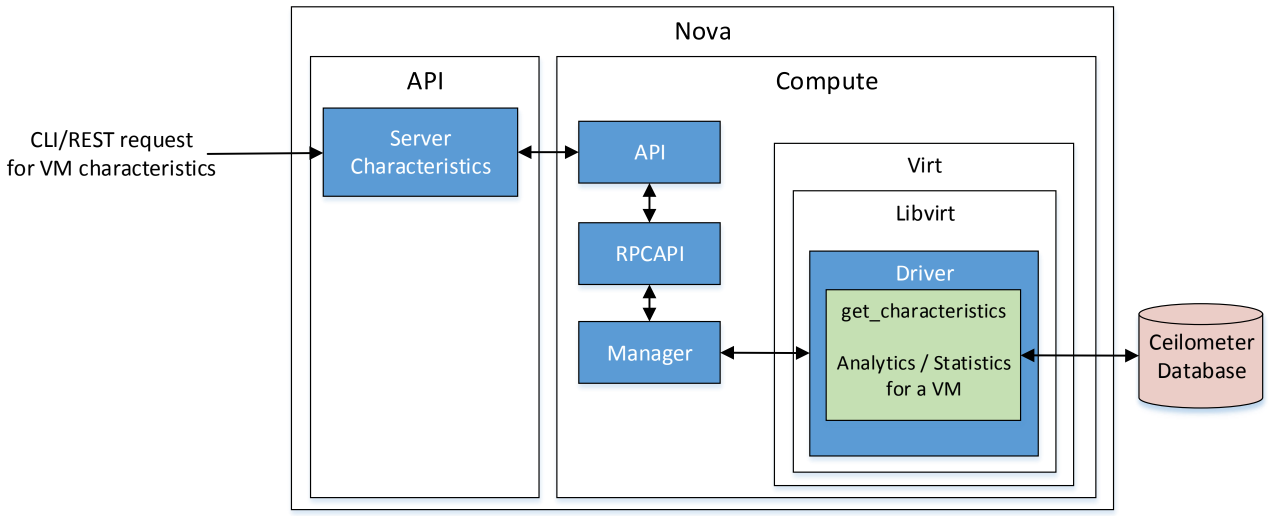

Figure 9: VM Characteristics Component for UniServer

VM Characteristics component provides a historic view of the resource consumption trend of a particular VM, and we plan to use it in D6.2, D6.3, and D6.5 of the project. Moreover, we also plan to enhance this component after finalizing the metrics collected by the UniServer hypervisor in D5.3.

3.2.2. VM Characterization Properties

The following table provide a list of properties that are being returned by the VM Characteristics component.

Table 2: List of properties returned by VM Characteristics component

Property	Description
cpu_util_amax	Maximum CPU utilization in percentage of the VM during the analysis window
cpu_util_amin	Minimum CPU utilization in percentage of the VM during the analysis window

Property	Description
cpu_util_mean	Mean CPU utilization in percentage of the VM during the analysis window
cpu_util_median	Median CPU utilization in percentage of the VM during the analysis window
cpu_util_peak_to_peak	Difference between the maximum and minimum CPU utilization in percentage of the VM during the analysis window
cpu_util_slope	Slope of the line of best fit of the historic data for CPU utilization during the analysis window
cpu_util_slope_intercept	Intercept of the slope of the line of best fit of the historic data for CPU utilization during the analysis window
cpu_util_slope_std_err	Standard error of estimate in calculating the slope of the line of best fit for CPU utilization of the historic data during the analysis window
cpu_util_std	Standard deviation in CPU utilization of the VM during the analysis window
cpu_util_variance	Variance in CPU utilization of the VM during the analysis window
image_memory	Maximum amount of memory in MB the instance can have
resident_memory_amax	Maximum resident memory in MB at host for the VM during the analysis window
resident_memory_amin	Minimum resident memory in MB at host for the VM during the analysis window
resident_memory_mean	Mean resident memory in MB at host for the VM during the analysis window
resident_memory_median	Median resident memory in MB at host for the VM during the analysis window
resident_memory_peak_to_peak	Difference between the maximum and minimum resident memory in MB at host for the VM during the analysis window
resident_memory_slope	Slope of the line of best fit of the historic data for resident memory during the analysis window
resident_mem _slope_intercept	Intercept of the slope of the line of best fit of the historic data for resident memory during the analysis window
resident_mem_slope_std_err	Standard error of estimate in calculating the slope of the line of best fit of the historic data for resident memory during the analysis window
resident_memory_std	Standard deviation in resident memory of the VM at host during the analysis window
resident_memory_variance	Variance in resident memory of the VM at host during the analysis window

3.2.3. UniServer Workload Resource Utilization Characteristics

The VM Characteristics component is used to determine the characteristics of two of the target applications that would be used in UniServer, i.e., jammer detector application from WSE and Polaris benchmark from MER. These applications are executed on two-node OpenStack cluster running on X-Gene2 Merlin boards. Each X-Gene2 boards has 8 CPUs and 32GB of main memory. These applications are executed to observe the patterns of their CPU utilization in the VM and resident memory utilization at the host server for the respective application as reported by the VM Characteristics component, and also to demonstrate a working OpenStack on X-Gene2 boards.

3.2.3.1 Jammer Detector Application

The Jammer Detection benchmark is based on a WSE solution called Denial of Service (DoS) sensing. This solution consists of a sensor capable of detecting wireless interference (jamming) signals that attempt to cause wireless networks DoS. The use of these wireless jamming devices is considered a security attack and the main goal of the solution is to detect the threat and identify the type of jammer attack that is being performed. The details of this benchmark are provided in D4.4 of UniServer project.

Jammer detector benchmark is executed on an OpenStack cloud running on X-Gene2 Merlin boards. Currently, this application is configured as on offline application, such that it reads and processes the input data from within the VM. Figure 10 shows the trend of CPU utilization in the VM as observed by periodic use of VM Characteristics API, where four of the properties returned by the VM Characteristics module that are related to CPU utilization are plotted. It can be observed that the jammer detector application uses about 45% maximum overall CPU inside the VM. We also plot the average CPU utilization of the VM as it can be observed from within the VM during the execution of the benchmark using the `top` Linux utility. At each point the observation window of VM Characteristics module is 10 minutes, therefore, for the first 10 minutes the minimum CPU utilization observed during the observation window is close to zero. However, after the 10th minute, the maximum and minimum utilization observed during the particular window point come close to each other and follow this trend till the application execution is completed at 31st minutes.

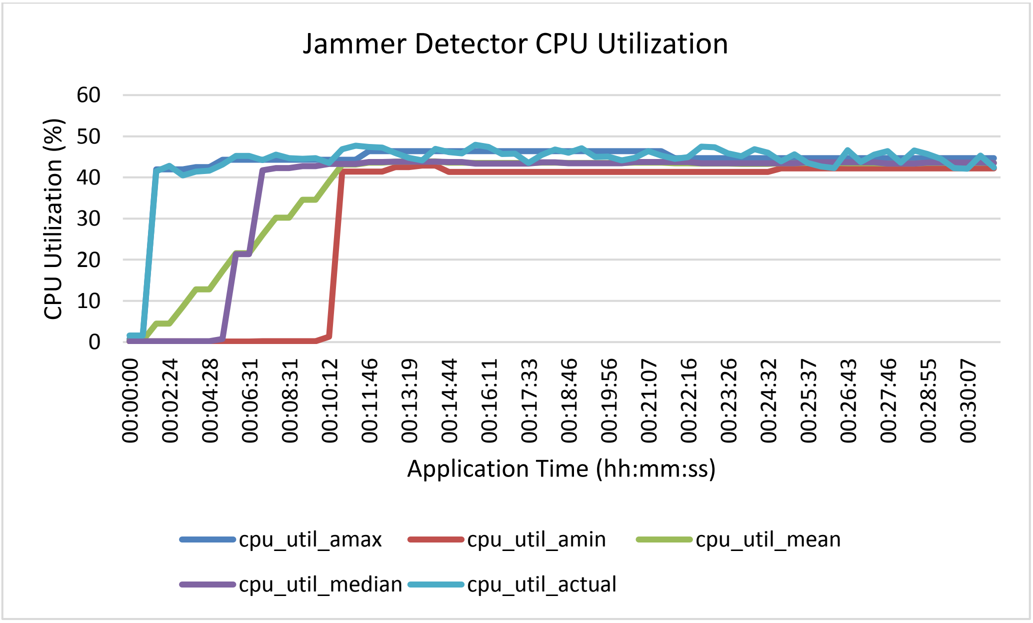

Figure 10: CPU Utilization of Jammer Detector Application

Figure 11 shows the trend in the resident memory consumption on host while running the jammer detector application. The resident memory on the host includes the memory allocated by the VM in addition to the overhead of QEMU process, and it is reported by the hypervisor. It shows that this application is memory intensive on the host, and it consumes about 6.5 GB of memory from the host, however, it the resident memory utilization gets stabilized around the 13th minute of its execution.

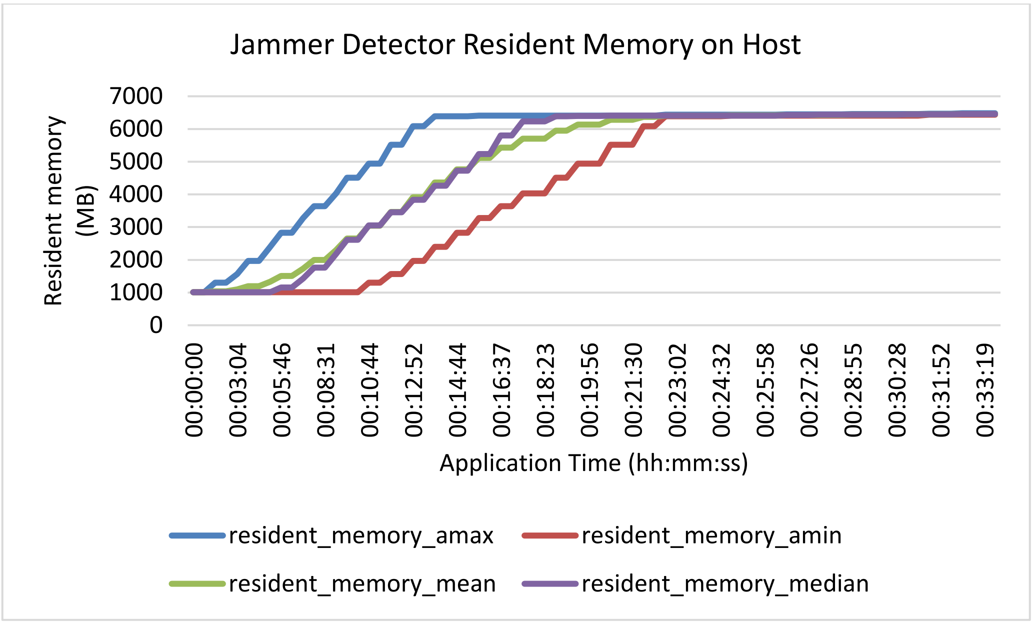

Figure 11: Resident Memory of Jammer Detector Application on Host

3.2.3.2 Polaris Benchmark

The Polaris Platform was developed as a third-party service for Investment Firms (IFs) to comply with the European Markets Infrastructure Regulation (EMIR) that came into force on 16 August 2012. The Regulation requires that all derivatives transactions need to be reported to Trading Repositories (TRs). The reporting of a derivative transaction should take place at T+1, where T is the transaction date, and it involves any daily modifications/updates of the transaction until the termination of the derivative contract. This requires the processing and validation of a large amount of information and handling sensitive client's data. The details of this benchmark are provided in D4.3 of the UniServer project.

Polaris benchmark is executed on an OpenStack cloud running on X-Gene2 Merlin boards. Polaris benchmark initializes the database for execution and then run the benchmark on the dataset that is currently packaged inside the VM. Figure 12 shows the trend of CPU utilization in the VM as observed by periodic use of the VM Characteristics API. We also plot the average CPU utilization of the VM as it can be observed from within the VM during the execution of the benchmark using the top Linux utility. This show that the Polaris benchmark steadily uses about 58% overall maximum CPU during its operations. However, the gap between the maximum and the minimum CPU utilization is more for the Polaris benchmark as compared to the jammer detector application. This can be explained by the fact that Polaris benchmark reads the data from database whereas the jammer detector application reads the data from the file, therefore, making the later continuously operate at the same pace.

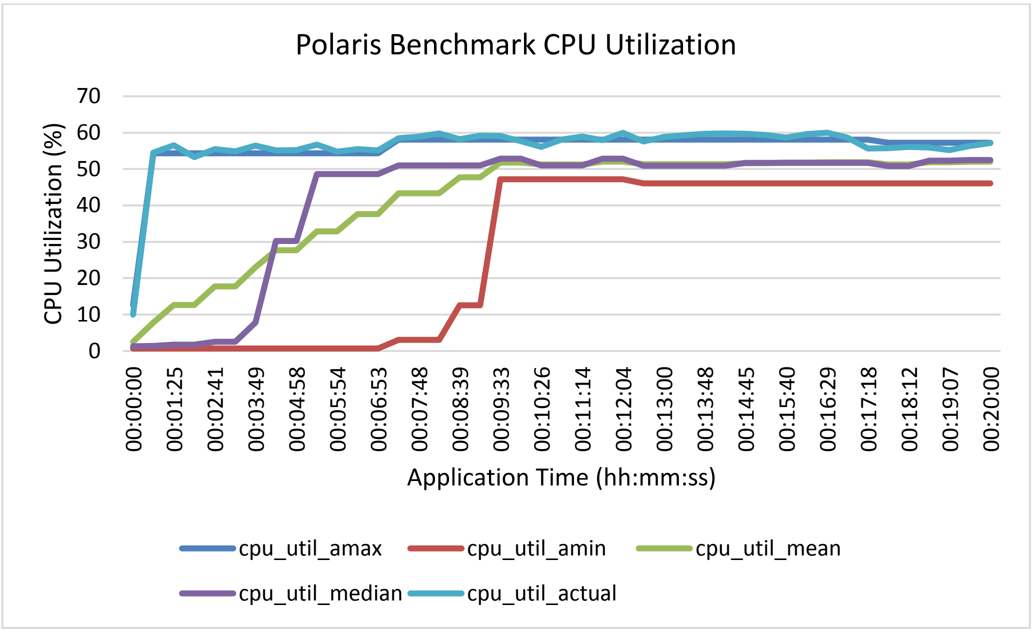

Figure 12: CPU Utilization of Polaris Benchmark

Figure 13 shows the trend in resident memory consumption on host while running the Polaris benchmark. Compared to the jammer detector application, the resident memory consumption of Polaris benchmark is significantly less, i.e., the maximum observed resident memory consumed at host by the Polaris benchmarks is about 1.5 GB, which indicates that it can be packaged in VMs with less amount of memory and these VMs can be scheduled on servers with less amount of available memory.

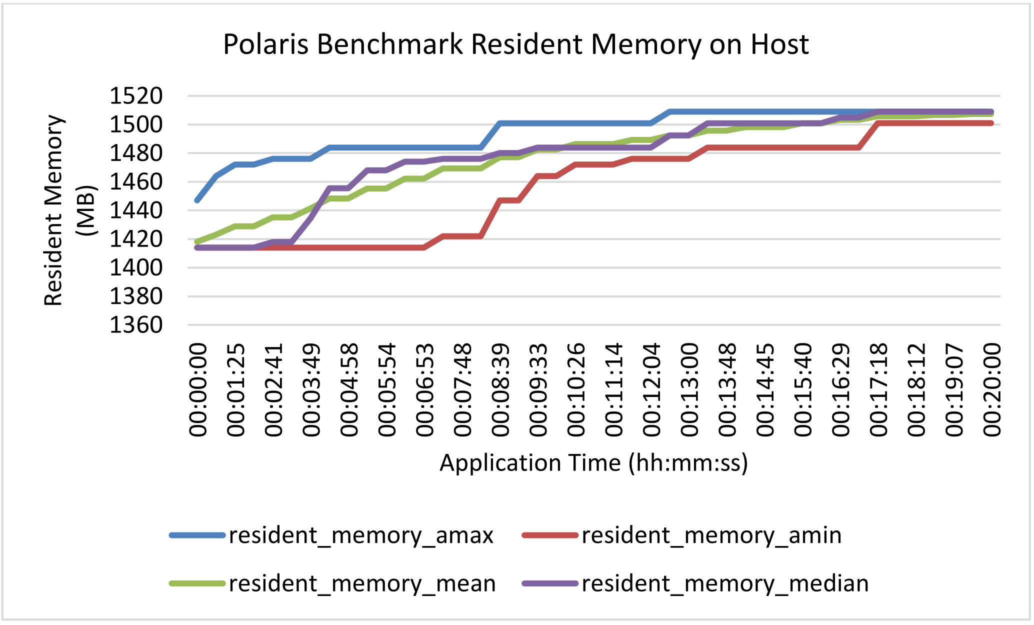

Figure 13: Resident Memory of Polaris Benchmark on Host

3.3. Libvirt Extensions and Configurations for OpenStack on X-Gene

As discussed in WP5, OpenStack can monitor the status of cloud/edge nodes through Libvirt. The extended version of the Libvirt for UniServer, described in D5.1, can provide information about the thermal state, the power consumption and the hardware-detected error rate of nodes. Except from reporting the status of the node to OpenStack, we have also extended Libvirt to accept requests from OpenStack to configure nodes as required by the cloud workload and the performance, power and reliability constraints OpenStack tries to satisfy at the system-level.

OpenStack uses a python interface, similar to the ones used for the information flow from the hypervisor to OpenStack to request node configuration at a specific performance, power and reliability level. Libvirt receives the request and notifies the hypervisor. The hypervisor tries to configure the system to satisfy the request, and reports success or failure (to Libvirt, which forwards the status to OpenStack). The interface and the implementation will be described in detail in deliverable D5.3. Figure 14 extends the flow of information between the Libvirt and OpenStack with the flow of the requests from OpenStack to Libvirt.

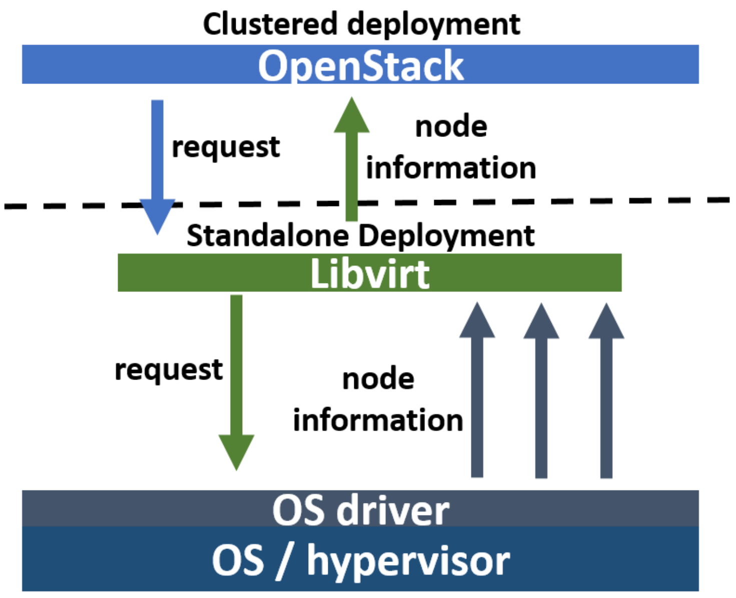

Figure 14: Requests and information flow through Hypervisor, Libvirt and OpenStack

We provide multiple levels of performance, power efficiency and reliability, for the CPU and multiple levels of power efficiency versus reliability for RAMs. The levels span both nominal configuration points of the hardware, as well as points exploiting the extended margins recognized by the characterization of the specific parts. More aggressive configurations within the extended margins are more power/performance-efficient, however at the cost of an increased probability of faults. OpenStack monitors the status of the node and decides the appropriate configuration level. For example, if the expected error rate is higher than the acceptable rate for the SLAs OpenStack has to conform to, it can send a request to reset the configuration within or closer to the nominal settings envelope.

4. References

[1] S. Pousty and K. Miller, Getting Started with OpenShift, 1491900474, 9781491900475: O'Reilly Media, Inc., 2014.

[2] M. Cinque, D. Cotroneo, F. Frattini and S. Russo, “To Cloudify or Not to Cloudify: The Question for a Scientific Data Center,” *IEEE Transactions on Cloud Computing,* vol. 4, no. 1, pp. 90-103, 2016.

[3] C. F. Foundation, “Cloud Foundry Overview,” [Online]. Available: https://docs.cloudfoundry.org/concepts/overview.html.

[4] OpenStack, “Open source software for creating private and public clouds,” [Online]. Available: https://www.openstack.org/.

[5] Libvirt, *libvirt - The virtualization API,* http://libvirt.org/index.html.

[6] E. Pakbaznia and M. Pedram, “Minimizing Data Center Cooling and Server Power Costs,” in *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, San Fancisco, CA, USA, 2009.

[7] J. Shuja, K. Bilal, S. A. Madani, M. Othman, R. Ranjan, B. P. and S. U. Khan, “Survey of Techniques and Architectures for Designing Energy-Efficient Data Centers,” *IEEE Systems Journal,* vol. 10, no. 2, pp. 507-519, 2016.

[8] A. Kivity , Y. Kamay, D. Laor, U. Lublin and A. Liguori, “KVM: the Linux Virtual Machine Monitor,” in *Ottawa Linux Symposium (OLS)*, Ottawa, 2007.

[9] OpenStack, “Welcome to the Ceilometer developer documentation!,” [Online]. Available: https://docs.openstack.org/ceilometer/latest/.

[10] SUSE OpenStack Cloud 7, *OpenStack Administrator,* Cambridge MA, USA: SUSE LLC, 2017.

[11] D. Chisnall, The Definitive Guide to the Xen Hypervisor, Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.

[12] S. Jin, VMware VI and vSphere SDK: Managing the VMware Infrastructure and vSphere, Upper Saddle River, NJ, USA: Prentice Hall PTR, 2009.

[13] OpenStack, “Welcome to Nova’s developer documentation!,” [Online]. Available: https://docs.openstack.org/nova/latest/.

[END OF DOCUMENT]