uniserver

D5.1 1st Report on Hypervisor / System / Software Interface

Contract number	688540
Project website	http://www.Uniserver2020.eu
Contractual deadline	Project Month 9 (M9): 31st October 2016
Actual Delivery Date	16th November 2016
Dissemination level	Public
Report Version	1.0
Main Authors	Bin Wang (QUB), Christos Antonopoulos (UTH), Charalambos Chalios (QUB), Georgios Karakonstantis (QUB), Srikumar Venugopal (IBM), Mustafa Rafique (IBM), Christos Kalogirou (UTH), Panos Koutsovasilis (UTH), Emmanouil Maroudas (UTH), Dimitrios Nikolopoulos (QUB)
Reviewers	Peter Lawthers (APM), Spyros Lalis (UTH)
Keywords	Hypervisor, OpenStack, 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.

© 2016. Uniserver Consortium Partners. All rights reserved

Disclaimer

This deliverable has been prepared by the responsible Work Package of the Project in accordance with the Consortium Agreement and the Grant Agreement Nr 688540. It solely reflects the opinion of the parties to such agreements on a collective basis in the context of the project and to the extent foreseen in such agreements.

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)

Meritorious Audit Limited (MER)

Sparsity (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	• Add metrics of interest
0.3	• Modify sections 2, 3
0.4	• Modify Introduction • Add metrics and APIs about Ceilometer, Healthlog and StressLog • Update Figure 1
0.5 and 0.6	• Add detailed metrics for OpenStack • Add Libvirt API extension
0.7	• Reorganize Sections 2 and 3
0.8	• Delete the redundant information and revise the figures.
0.9	• Add references
1.0	• Integrate comments from partners, update table of contents. • Fix type and grammar errors • Reorganize Section 2, 3 • Revise Figure 1

Table of Contents

Index of Figures

Index of Tables

Executive Summary

Uniserver seeks to improve the performance and energy efficiency in servers by automatically discovering the capabilities of the underlying hardware components and extending their operation into new wider Voltage/Frequency/Refresh rate (VFR) operating points than what the conventional worst case design dictates. To fully utilize such new operating points and ensure easy adoption of the developed technologies, Uniserver plans to port and enhance state-of-art software packages for virtualization (i.e. KVM) and resource management (i.e. OpenStack), essential components of modern data-centres. A major breakthrough of Uniserver lies not only on the utilization of existing mechanisms of such state-of-the art software packages on ARM based micro-servers but more significantly on the extension of their capabilities. In particular, Uniserver will enable the monitoring of the hardware behaviour by the system software as reported by the Health daemon (specified in the deliverable D4.2) and will optimize system level metrics (at a node and rack granularity) by tuning the extended operating points revealed by the Stress daemon and Predictor (specified in the deliverable D4.3). An essential breakthrough of the Uniserver is the control and minimization of the effects of potential hardware faults on system software and end-user applications, without incurring an overhead which would outweigh the benefits of the extended margins.

The first step to achieve the aforementioned objectives, is to enable the communication of new vital information from the hardware to system software and all the way up to the application layer. This is the target of this deliverable: to specify the bidirectional communication between the hypervisor and the OpenStack infrastructure enabling them to synergistically configure the system in a way compatible with application requirements and hardware capabilities. OpenStack informs the hypervisor about the virtual resources required by applications, whereas the hypervisor communicates hardware status, physical resources assigned and associates detected faults with the VMs that are potentially affected. To minimize the overheads of the new interface we need to ensure that only the required information is passed on to the next upper layer in a way that is relevant and useful to the target layer. Therefore, in D5.1 we start by identifying the existing and the Uniserver related metrics of interest at each layer, which we use as guidance for the selection of the parameters that we need to monitor in each layer, as well as in the specification of the necessary communication system software interface.

1. Introduction

The Uniserver project targets a wide range of use cases, ranging from deployments in remote locations close to the end users to deployments in cloud data centers. To facilitate such diverse use cases, the Uniserver platform must be equipped with a complete software stack able to manage efficiently any compute and storage resources by offering easy installation, migration and replication of tasks, either at the node or server-rack level [1]. Therefore, in Uniserver project state-of the art software packages for virtualization and resource management will be ported on the targeted 64-bit ARM based micro-server, which was analyzed in the deliverable D3.1. In particular, we will adopt a KVM [2] hypervisor which lends to the developed ecosystem the numerous benefits provided by virtualization such as easier installation, replication, migration of tasks. At the upper layer, Uniserver has selected to adopt the OpenStack [3] software framework, which is open source and pairs well with popular existing enterprise and open source technologies. Note that both KVM and OpenStack were only recently ported on 64-bit ARM based systems, while recently (October 2016) it was announced by major software and processor vendors that OpenStack will be made commercially available at an enterprise-grade, indicating the timely and novel character of the Uniserver developments at the software stack.

But above all a major breakthrough of the Uniserver project lies on the utilization of new wider Voltage/Frequency/Refresh rate (VFR) operating points that are going to be exposed by the developed mechanisms at the firmware level (Health and Stress Daemon, explained in deliverables D4.2 and D4.3). Operation very close to the operating limits (essentially without any guardband as opposed to the traditional paradigms) increases significantly the risk of errors within each core and memory components [4]. Therefore, the system software needs to be enhanced with new mechanisms for effectively controlling and minimizing the effects of potential faults, while trying to not incur significant overhead, which would outweigh the benefits of operating at extending points. The operating points may dynamically change depending on the workload, variations of environmental conditions, chip aging etc. and thus the system software should be able to decide on the right energy efficient configuration parameters very fast and reliably [5]. To enable the efficient management of each node and of the overall data-centre there is need to rethink the existing mechanisms and communication interfaces and allow the exchange of new information about the behaviour of the hardware from the bottom layers to hypervisor and on to the OpenStack. This is exactly the target of the deliverable D5.1, which aims at specifying the bidirectional communication between the hypervisor and the OpenStack infrastructure.

Before going into the details of such an interface next we describe some of the basic components of the system and some potential ways in managing the extended operating points. This will help at this early stage of the project to better understand the interaction between the system layers and identify the mechanisms and extensions that will need to be explored in the next tasks of the project. Note that the design of the fault tolerant hypervisor and the implementation of the power management policies is undertaken by the other tasks in work-package 5 (WP5) and will be described in more detail in the next deliverables.

1.1. The Uniserver Cross-Layer System Architecture: Definitions and Assumptions

Figure 1 depicts the different layers of the Uniserver ecosystem and visualizes the flow of the information between them through the essential components. One of the most fundamental assumptions in the system architecture of the Uniserver platform is that each CPU and DRAM DIMM may have intrinsically different capabilities. As explained in the deliverables D3.1 and D4.1 in the X-GENE2 [6] the chassis of the Uniserver the CPU Voltage/Frequency can be set separately on each PMD [7] (consisting of 2 CPUs and each of 2 levels of cache L1/L2) and the DRAM refresh-rate/Voltage/Frequency can be set per channel. As explained in the deliverable D4.1 the configuration values such as Voltage, Frequency and Refresh rate of the system and their subcomponents can be controlled by accessing the appropriate hardware sensor register and ACPI [8] state register through the hardware registers and software (e.g. i2c for Linux) available on the X-Gene microserver. Power and thermal of different hardware components such as DRAM, DIMMs, SoC and PMD can be recorded in the X-Gene microserver through the available thermal and power sensors, whereas

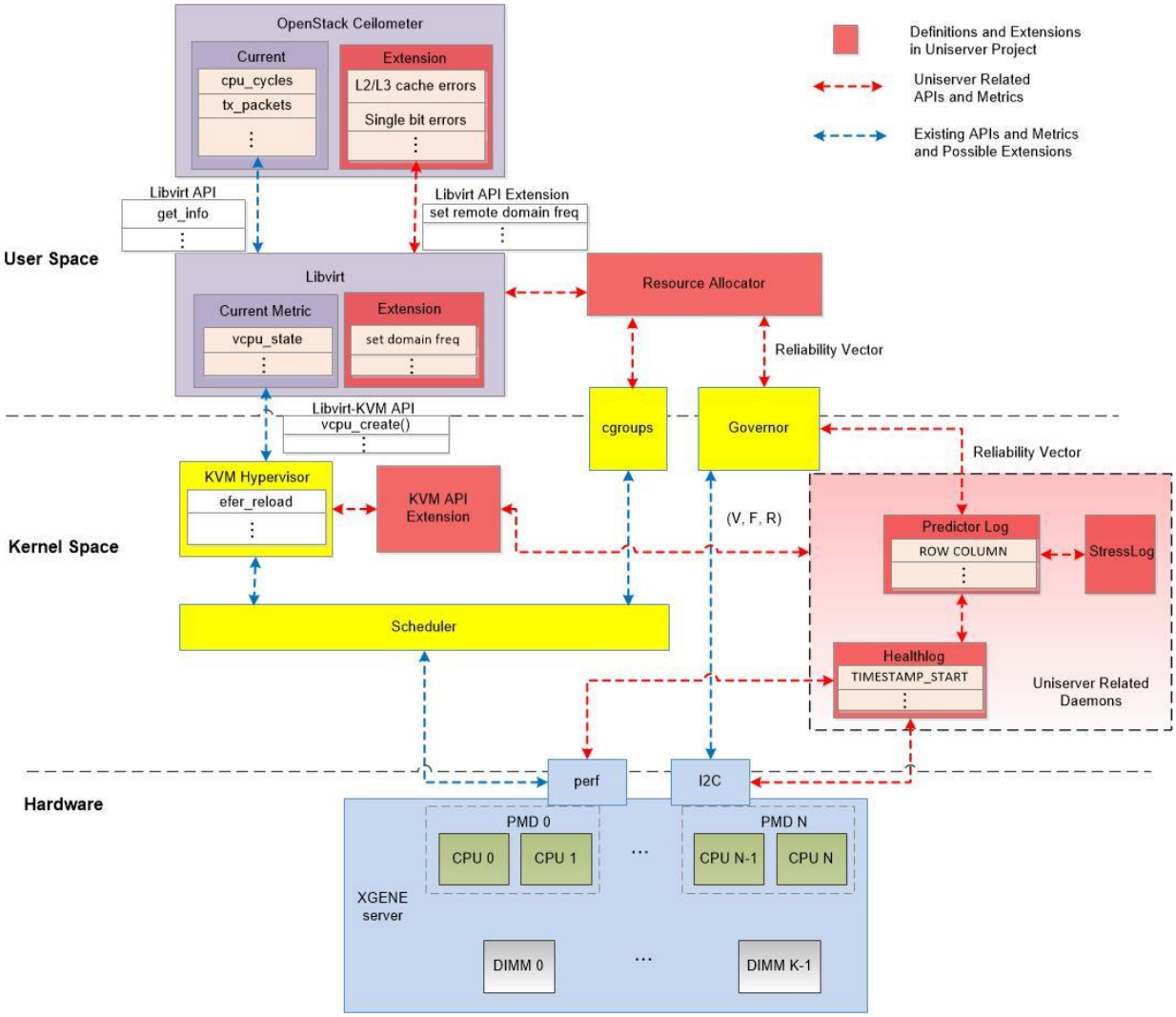

Figure 1: Potential exchange of information across system layers

correctable/uncorrectable errors within L2/L3 caches and DRAMs can be exposed to the software through the Hardware Exposure Register Interface (HEI) described in deliverable D4.1.

At the firmware level, Uniserver introduces the Predictor, Health and Stress daemon. The Health daemon (specified in deliverable D4.2) is a Linux daemon responsible for monitoring and logging the hardware state on the so called HealthLog, taking first-level actions when errors occur and also act as an end-point gateway for the hypervisor to access hardware metrics. On the other hand, the StressLog (specified in deliverable D4.3) is a Linux daemon responsible for offline, on-demand stress testing APM [6] (ARM architecture-based) computing systems and producing an output vector containing the new safe system VFR margins that will be suggested to the hypervisor for future usage. It also produces log files that will record the raw data provided the Hardware Exposure Interface (HEI) such as errors (correctable or uncorrectable) [9], system configuration values (e.g. voltage, frequency, refresh rate), sensor readings and performance counters. The two logs are differently used to provide information to different modules. In particular, Health daemon collects events occurrence (errors and others) during normal operation and other related information and stores them to logfiles for further use by the hypervisor, the predictor and other processes. On the other hand, Stresslog is used to provide for an offline performance analysis of the server using real applications and application-specific viruses. Predictor is another mechanism which tries to predict the HW behavior offering to the hypervisor

estimations about potential failures helping to decide on the operating points. The predictor can be periodically re-configured (by invoking the StressLog during idle periods) in case that a strange behavior is being detected by the hypervisor or the OpenStack.

At the hypervisor layer, HW metrics related mainly to power and performance are already monitored such as CPU and memory utilization, cache misses etc. by utilizing an existing API. In Uniserver we plan to enhance such an API to make it able to collect and monitor Reliability related information by interacting with the introduced daemons. For example, CPU errors could be acquired from Healthlog to Monitor CPU module through new functions as explained in Section 3.

We would like to note here that the hypervisor (i.e. KVM) is a module within the kernel space [10] 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 virtual machines (VMs) on the so called guest machines (i.e. in the userspace [11]). In general, in Linux, the used Operating System (OS) in Uniserver that the memory is divided into two distinct the user space and kernel space. The user space, is a set of locations where normal user processes run (i.e. everything other than the kernel). The role of the OS kernel is to manage applications running in this space from messing with each other, and the machine. The kernel space, is the location where the code of the kernel is stored, and executes under. Processes running under the user space have access only to a limited part of memory, whereas the kernel has access to all of the memory. Processes running in user space also don't 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, the system calls**.**

Going further-up the OpenStack, which is another guest layer for resource management, extracts information for every virtual machine and the overall system using the so called Libvirt [12], which is a hypervisor-independent virtualization API and toolkit that is able to interact 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, a dedicated module called Ceilometer [12] talks with the Libvirt interface, which collects information and distributes it across all current OpenStack core components.

The flow of information from the OpenStack towards the lower system software layers mainly include the requirements for reliability that the system needs to provide. Generally, this information will be per VM (when possible) reflecting the different Service-Level Agreement (SLA) [13] between the Cloud provider and the end-user of each VM. Providing information about reliability requirements per VM allows the Uniserver framework to utilize the underlying hardware optimally, by pushing each separate component (PMD, memory domain, etc.) at the proper margins that maximize the targeted metric and at the same time ensuring the requested reliability levels.

The design of those components that will be responsible to implement the policies for managing the reliability level of the various components and mapping of VMs to hardware will determine the set of interfaces that will need to be implemented between the various components and the exact flow of information.

One possible design, shown in Figure 1, that we will examine, implements all the policies selecting the reliability level of each hardware component and mapping VM to resources at the userspace. Such a design minimizes the development time by re-using well defined interfaces that Linux provides for configuring hardware components and allocating resources to userspace components. OpenStack communicates the requirements of reliability for every VM through Libvirt to the Resource Allocator which is a new component responsible for implementing all the policies for choosing the optimal operating point of every HW component taking into account the requirements of all VMs running on the microserver. Linux provide Governors which implements policies for Voltage and Frequency scaling of CPUs. Governors provide a *sysfs* interface [10] to the user applications to select the operating points of the machine. In Uniserver, we will potentially implement an enhanced Governor that will get the required reliability for every hardware component as an input. It will consult

the predictor in order to find the operating points of the corresponding hardware component that match the requested reliability level and apply the configuration. The Resource Allocator (RA) shown in the figure uses the *cgroups* interface to allocate resources to VMs. *cgroups* is a *vfs*-based interface [14] that allows user applications to apply restrictions to the amount of resources (e.g. number of CPUs, CPU time, amount of memory, etc.) [15] each user process of the system can consume.

We would like to note here that the system and interactions described above provide the potential extensions at the userspace; alternatively, the Uniserver related component that implements all the resource allocation policies and selection of operation points for hardware parts can be implemented within the kernel. This design would provide higher flexibility in resource allocation, but at the cost of development effort. For example, *cgroups* provide a static way of mapping processes to resources. A more dynamic scheme would allow to react faster in variations of the workload. In this scenario the Linux scheduler will be in charge of enforcing the reliability requirements per VM, each Virtual CPU (VCPU) is mapped to a kernel thread and thus it can be scheduled by the Linux scheduler. The reliability information [16] per VM will be communicated through the KVM interface, which in this case will be the only interface between the userspace and the kernel space.

Finally, in both cases the KVM will need to be extended and implement the policies of choosing the optimal configuration for every hardware resource of the system. As we said, this is the target of the rest of the tasks in WP5 and the chosen implementation will be discussing in more detail in the following deliverables.

Concerning the communication interfaces which are the main target of this deliverable, we would like to stress that in both potential implementations of the enhanced management/resource allocator module there is additional information that need to be exchanged between the hypervisor and the OpenStack (i.e. Ceilometer). The additional reliability information obtained the Health daemon and the predictor need to be packed and propagated by the hypervisor to the OpenStack layer.

1.2. Organization

In Section 2 we describe the metrics of interest that are already being monitored by the OpenStack and describe the Uniserver related extensions in the list of the metrics that need to be monitored for enabling the management of the system operation at extended margins. In Section 3, we define and extend the interface between Libvirt, OpenStack and KVM hypervisor that is the target of this deliverable.

2. Metrics of Interest

As discussed in the previous section, Uniserver's system architecture spans across all layers of the system stack. In this section, we discuss the existing metrics of interest as well as the Uniserver related extensions at each layer starting from the OpenStack and moving down to the rest of the layers. This can help to identify the information that is required to be exchanged between different layers of the system stack, and is essential for specifying the potential communication interface in Section 3.

In general, the optimization of the operations at the extended margins in Uniserver will be guided by the system requirements of the end-user for each VM, which are typically communicated to the Cloud provider through SLAs. These workload-specific requirements reflect the main metrics of interest based on which the OpenStack manages the individual nodes (or machines) that make up the cloud infrastructure (i.e. data-centre). The main metrics of interest are the node Availability, Utilization and Energy Usage. In the context of Uniserver, an additional metric, i.e., node reliability [9], has also been included in the list of metrics such that nodes can be characterized based on the experienced error rates [17]. In brief, the main high-level metrics of interest for Uniserver are the following:

- **Availability:** defined as the percentage of time a node is capable of serving its intended function. When OpenStack controllers such as the scheduler and the performance monitoring system are not able to reach the node, then it will be considered as unavailable. We will also consider VMs as being available or unavailable depending on how they respond to OpenStack probes.
- **Utilization:** defined as the ratio of occupied resources (CPU, memory, disk, etc.) in a node to their total capacity.
- **Energy Usage:** defined as the total energy usage of a node in *kiloWattHour* (kWh) for a specific time period.
- **Reliability:** defined as the probability that a node could fail in a defined time horizon. MTBF (Mean Time Between Failures) and MTTR (Mean Time to Recovery) are related measures for a node reliability. We will classify a node as reliable or unreliable based on the probability of failure of its sub-components, which in the context of Uniserver are CPU, memory, or disk. An unreliable node will have a higher probability of failure, based on a yet to be defined threshold.

2.1. OpenStack Metrics

Currently to estimate the above metrics of interest from each node the OpenStack employs a data collection service, i.e., Ceilometer [7], at each node that is a part of the OpenStack cluster, which gathers information from the local Libvirt daemons. The information collected by Ceilometer is used by the OpenStack to monitor the 'health' of the physical nodes on which virtual machines are running and appropriately managing the underlying resources based on the user requirements. The details of the OpenStack will be discussed in WP6, however, in brief as shown in Figure 2, the Ceilometer consists of the following:

- Polling agent: Daemon that polls OpenStack services and define meters to measure a specific aspect of resource usage
- Notification agent: Daemon that listens to notifications and generates events
- Collector: Daemon that gather and record event and metering data created by Agents
- API: Service to query and view the recorded data

Gnocchi is a time series database, which provides optimized storage and querying for the metrics of interest.

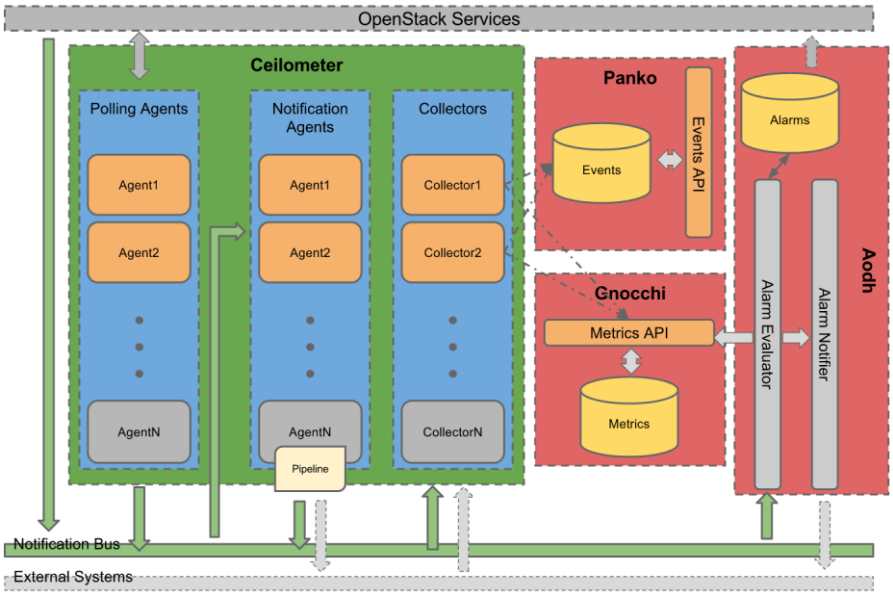

Figure 2: Ceilometer Architecture

Table 1 describes the metrics that are already being collected by the Ceilometer.

Table 1: Ceilometer Metrics

Component	Metric	Metric Information
Instance	*Name*	Name of the instance
	UUID	UUID associated with the instance
CPU	*Number*	Number of CPUs
	Time	Cumulative CPU time
	CPU_util	CPU utilization in percentage
	l3_cache_usage	Amount of CPU L3 cache used
Memory	*usage*	Amount of memory used
	resident	Amount of resident memory
	total	Total system bandwidth from one level of cache
	local	Bandwidth of memory traffic for a memory controller
Perf Events	*CPU_cycles*	Number of CPU cycles one instruction needs
	instructions	Count of instructions
	cache_references	Count of cache hits
	cache_misses	Count of caches misses

Component	Metric	Metric Information
Network	*name*	Name of the vNIC
(Virtual)	*mac*	MAC address
	Fref	Filter ref
	parameters	Miscellaneous parameters
	rx_bytes	Number of received bytes
	rx_packets	Number of received packets
	tx_bytes	Number of transmitted bytes
	tx_packets	Number of transmitted packets
	rx_bytes_rate	Rate of received bytes
	tx_bytes_rate	Rate of transmitted bytes
Disk	*device*	Device name for the disk
	read_bytes	Number of bytes read
	read_requests	Number of read operations
	write_bytes	Number of bytes written
	write_requests	Number of write operations
	errors	Number of errors
	read_bytes_rate	Number of bytes read per second
	read_requests_rate	Number of read operations per second
	write_bytes_rate	Number of bytes written per second
	write_requests_rate	Number of write operations per second
	disk_latency	Average disk latency
	iops_count	Number input/output operations per second (IOPS)
	capacity	Capacity of the disk
	allocation	Allocation of the disk
	Physical	Usage of the disk

2.2. Hypervisor Metrics

As we discussed in Section 1, in Uniserver we will use KVM hypervisor to provide virtualization services, and gather information from the introduced daemons and relate it to the executed VMs. This information will be

propagated to the OpenStack through the Libvirt interface for managing the resources at the server granularity, and will be used by the hypervisor for optimizing the system operation at the node granularity.

Currently, the KVM has already modules that collect various system metrics-information for managing the system operation. In particular, the hypervisor used the *kvm_stat* which is essentially a Python [18] script which retrieves runtime statistics from the KVM kernel module. The *kvm_stat* command is used to diagnose guest behavior visible to the hypervisor, such as performance related issues with guests. Currently, the reported statistics are for the entire system and the behavior of all running guests is reported. Table 2 details the basic KVM metrics from *kvm_stat*:

Table 2: KVM State Metrics

Metric	Metric Information
efer_reload	The number of Extended Feature Enable Register (EFER) reloads
fpu_reload	The number of times a VMENTRY reloaded the FPU state. The fpu_reload is incremented when a guest is using the Floating Point Unit (FPU).
halt_exits	Number of guest exits due to halt calls. This type of exit is usually seen when a guest is idle.
halt_wakeup	Number of wakeups from a halt
host_state_reload	Count of full reloads of the host state (currently tallies MSR setup and guest MSR reads).
Hypercalls	Number of guest hypervisor service calls.
insn_emulation	Number of guest instructions emulated by the host.
insn_emulation_fail	Number of failed insn_emulation attempts.
io_exits	Number of guest exits from I/O port accesses.
irq_exits	Number of guest exits due to external interrupts.
irq_injections	Number of interrupts sent to guests.
irq_window	Number of guest exits from an outstanding interrupt window.
Largepages	Number of large pages currently in use.
mmio_exits	Number of guest exits due to memory mapped I/O (MMIO) accesses.
mmu_cache_miss	Number of KVM MMU shadow pages created.
mmu_flooded	Detection count of excessive write operations to an MMU page. This counts detected write operations not of individual write operations.
mmu_pde_zapped	Number of page directory entry (PDE) destruction operations.
mmu_pte_updated	Number of page table entry (PTE) destruction operations.
mmu_pte_write	Number of guest page table entry (PTE) write operations.
mmu_recycled	Number of shadow pages that can be reclaimed.

Metric	Metric Information
mmu_shadow_zapped	Number of invalidated shadow pages.
mmu_unsync	Number of non-synchronized pages which are not yet unlinked.
nmi_injections	Number of Non-maskable Interrupt (NMI) injections to the guest.
nmi_window	Number of guest exits from (outstanding) Non-maskable Interrupt (NMI) windows.
pf_fixed	Number of fixed (non-paging) page table entry (PTE) maps.
pf_guest	Number of page faults injected into guests.
remote_tlb_flush	Number of remote (sibling CPU) Translation Lookaside Buffer (TLB) flush requests.
request_irq	Number of guest interrupt window request exits.
signal_exits	Number of guest exits due to pending signals from the host.
tlb_flush	Number of tlb_flush operations performed by the hypervisor.
Exits	The count of all VMEXIT calls.

2.3. Extended Metrics for Uniserver

In addition to the existing metrics of interest described above, in Uniserver platform, the hypervisor will have access to additional information for monitoring the status of the hardware which will be made available through the introduced Health, Stress and Predictor daemons which have been elaborated in detail in deliverables D4.2 and D4.3. Note that part of the additional monitored information will also be propagated to the OpenStack.

Based on the information vectors defined in D4.2 and the HEI capabilities defined in D4.1 we have summarized in the Table 3 the metrics that will be offered by the hypervisor to the Libvirt. For each metric, the table also defines the source of information in the Uniserver software and hardware ecosystem.

We plan to proceed with an initial implementation of the Libvirt extentions to provide these metrics based on information sources that are available from typical interfaces of Linux systems, such as the *proc* and *sys pseudofilesystems*, *NetLink* sockets which serve as a communication channel between kernel and userspace processes, as well as directly from the target XGene hardware. When the Health daemon will be released, we will switch to an implementation for these metrics using HealthLog as the primary source of information whenever possible.

Table 3: Uniserver metrics of interest – Source of information in the Uniserver Software / Hardware ecosystem

Metric	Source
Number of CPUs	*/proc/CPUinfo*
CPU capacity (MHz)	*/proc/CPUinfo*
Average CPU utilization (% and MHz)	*/proc/stat*
Per core CPU utilization (% and MHz)	*/proc/stat*

Metric	Source
CPU capacity reserved for throttling (% and MHz)	*/sys/devices/system/CPU/* Derived metric, see below
Total memory (MB)	*/proc/meminfo*
Available memory (% and MB)	*/proc/meminfo*
Memory speed / refresh rate (MHz)	HEALTHLOG
Cached memory (% and MB)	*/proc/meminfo*
Memory used by the individual VM (% and MB)	Libvirt (/proc/PID/stat)
Memory used by all VMs (% and MB)	Libvirt (/proc/PID/stat)
Contention for memory (% and MB)	*/proc/meminfo* Derived metric, see below
Total memory swap space (MB)	*/proc/meminfo*
Wait time for individual VM in swapping (% and msec.)	*NetLink* Socket Derived metric, see below
Average wait time for all VMs in swapping (% and msec.)	*NetLink* Socket Derived metric, see below
CPU wait time (msec.)	*/proc/stat*
CPU contention time (% and msec.)	*/proc/loadavg, /proc/CPUinfo* Derived metric, see below
CPU idle time (% and msec.)	*/proc/stat*
CPU system time (% and msec.)	*/proc/stat*
CPU power state optimization (i.e., optimized for power or performance)	*/sys/devices/CPU/*
CPU time running at maximum utilization (msec.)	*/proc/loadavg* Kernel extension required, see below
VM interruption time to run system services for VM or other VM (msec.)	*/proc/PID/stat* (execution time - guest time)
Number of VMs	Libvirt
Number of active VMs	Libvirt
Overall power consumption of the host (Watts)	HEALTHLOG
Power consumption by the memory (Watts)	HEALTHLOG

Metric	Source
CPU temperature per core (Celsius)	HEI
Average CPU temperature of the host (Celsius)	HEALTHLOG
Overall memory errors rate (errors per minute)	HEALTHLOG
Overall CPU errors rate (errors per minute)	HEALTHLOG (all caches)
Number of L1/L2/L3 cache errors	HEALTHLOG
Frequency of L1/L2/L3 cache errors (errors per minute)	HEALTHLOG
Number of single bit CPU errors	HEALTHLOG (all caches)
Frequency of single bit CPU errors (errors per minute)	HEALTHLOG (all caches)
Number of multi-bit CPU errors	HEALTHLOG (L2/L3 caches)
Frequency of multi-bit CPU errors (errors per minutes)	HEALTHLOG
Number of single bit memory errors	HEALTHLOG
Frequency of single bit memory errors (errors per minute)	HEALTHLOG
Number of multi-bit memory errors	HEALTHLOG
Frequency of multi-bit memory errors (errors per minute)	HEALTHLOG
Number of single bit L1/L2/L3 cache errors	HEALTHLOG
Frequency of single bit L1/L2/L3 cache errors (errors per minute)	HEALTHLOG
Number of multi-bit L2/L3 cache errors	HEALTHLOG
Frequency of multi-bit L2/L3 cache errors (errors per minute)	HEALTHLOG

Note that CPU capacity reserved for throttling is calculated as:

$$ReservedCapacity_{CPU} = 1 - \frac{CPUFrequency_{current}}{CPUFrequency_{max}}$$

The memory contention is calculated as:

$$Contention_{mem} = 1 - \frac{MemFree - (SwapTotal - SwapFree)}{MemTotal}$$

Values greater than 1 mean that the system is under memory contention.

Furthermore, the CPU contention is calculated as:

$$Contention_{CPU} = \frac{\#\ of\ Runnable\ Processes\ (within\ a\ time\ window)}{\#\ of\ Cores}$$

Values greater than 1 mean that the system is under CPU contention.

The *wait* time individual VMs spent in swapping can be attained through a NetLink socket. The average *wait* time for all VMs in swapping can be calculated by combining the wait time for individual VM in swapping for all VMs.

Finally, the time CPUs run at maximum utilization (msec.) can be calculated as follows:

A new timestamp is set when the system load reaches the number of CPUs. When the system load drops below the number of CPUs, the difference between current time and the timestamp adds to the total CPU max utilization time and the timestamp resets. This information will be exported through an extension of the */proc/loadavg* interface.

3. System Software Interface and API Extensions

Having identified the metrics of interest and the sources of the required information, in this section we describe in more detail the Libvirt API between OpenStack and hypervisor and specify the possible extensions which we will introduce for enabling the exchange of the metrics of interest between the two layers.

3.1. Libvirt

Libvirt is a hypervisor-independent virtualization API [19] and toolkit that is able to interact with the virtualization capabilities of a range of operating systems. Libvirt already supports plenty of configuration options and manages virtual machines and the virtual machine storage and network efficiently. It also supports different virtualization hypervisors among them the KVM/QEMU and it offers bindings in other languages such as Python. OpenStack exploits the Python interface of the Libvirt to extract information for a virtual machine or the system and for also handling any functionality of the virtual machines. Figure 3 shows the structure of Libvirt.

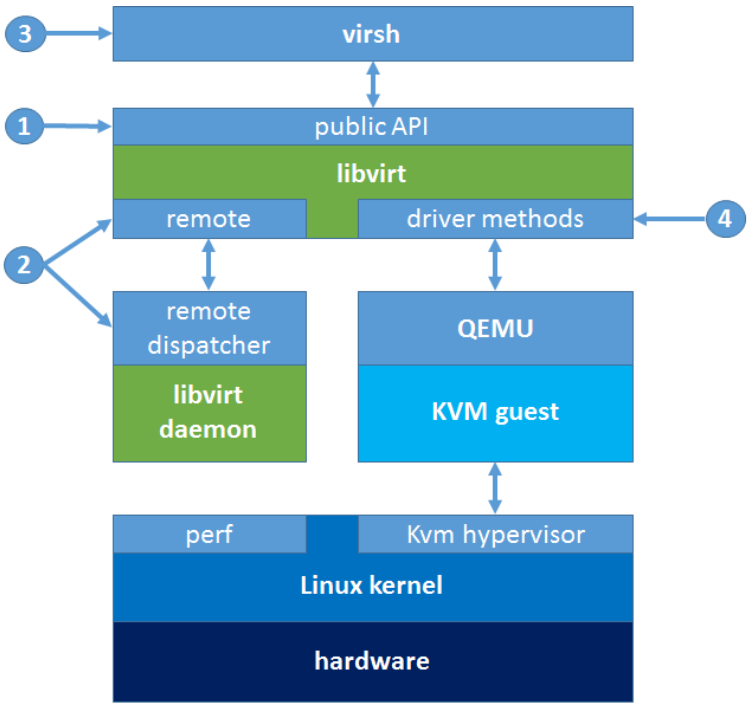

Figure 3: Libvirt structure and interface. The numbers indicate the different steps involved in extending the API.

Libvirt consists of two parts: (a) a public API for third-party applications to use the library, and (b) a driver API. The latter contains the drivers, which enable communication with different hypervisor implementations. The drivers implement a common API to communicate with Libvirt, which is internally translated to the API each hypervisor implementation exports. The goal is to interact with hypervisors and virtual machines in a common way, regardless of the hypervisor implementation. However, when an external application communicates with Libvirt, it uses an URI, that defines, via the virInitialize API, which driver to use.

Furthermore, Libvirt offers remote management facilities by implementing a remote driver on the client and a

daemon for handling requests, called *libvirtd*, on the server side. Requests from a client are tunneled through the remote driver to the server, where the specific hypervisor is running. The *libvirtd* on the server receives the remote commands and locally invokes the appropriate driver.

The last piece of Libvirt software is Virsh [20], which is a virtualization shell built on top of Libvirt. This shell permits use the Libvirt functionality, but in an interactive, shell-like fashion.

Table 4 gives the details of the existing API used to collect and record VM information.

Table 4: Libvirt API Metric

API	API Information
get_max_memory	Get the maximum memory allocation.
get_info	Get information about a domain
get_xml_desc	Get the XML description of a domain.
get_scheduler_type	Get the scheduler type.
get_scheduler_parameters	Get the array of scheduler parameters.
pin_vCPU	Pins a domain vCPU to a bitmap of physical CPUs.
get_vCPUs	get_vCPUs dom maxinfo maplen returns the pinning information for a domain.
get_pCPU_stats dom	Returns the physical CPU stats for a domain.
get_max_vCPUs	Returns the maximum number of vCPUs supported for this domain.
get_domains	Get the active and/or inactive domains using the most efficient method available.
get_domains_and_infos	This gets the active and/or inactive domains and the domain info for each one using the most efficient method available.

3.1.1. Extending Libvirt API

In the context of Uniserver Libvirt is going to be extended to enrich the information exchanged between the node and the cloud management framework (OpenStack). Beyond monitoring the status and health of VMs, the additional information allows OpenStack to monitor the status, configuration and health (errors) of the node.

The following subsections discuss in more detail the steps involved in extending the Libvirt API. As a running example we use the implementation of a new function which allows the user to change the frequency of the cores.

3.1.2. Definition and Implementation of the Public API

The first step (Figure 3) is the definition and implementation of the new function (Figure 4) serving as the public API for the new functionality. The function prototype is declared in the public header file and is exported so that any other program is able to use it. The public headers are included in *include/libvirt/* and the function name should be added in *src/libvirt_public.syms* in order to be exported. The function of our running example is implemented in *src/libvirt-domain.c*, wiring up the public API with the internal driver API. After a number of

validity checks, the control is passed to the driver implementation.

```
int virNodeSetFrequency(virConnectPtr conn,
                        int core,
                        const char *frequency)
{
      VIR_DEBUG("conn=%p, core=%d, frequency=%s", conn, core, frequency);

      virResetLastError();
      virCheckConnectReturn(conn, -1);

      if (conn->driver->nodeSetFrequency) {
            int ret;
            ret = conn->driver->nodeSetFrequency(conn, core, frequency);
            if (ret < 0)
                  goto error;

            return ret;
      }

      virReportUnsupportedError();

      error:
            virDispatchError(conn);
            return -1;
}
```

Figure 4: Implementation of *virNodeSetFrequency* function

3.1.3. Implementation of the Remote Control

The implementation of the remote control (Figure 3, step 2) consists of the following:

- definition of the wire protocol
- implementation of the RPC client
- implementation of the server side dispatcher

Two new structs are needed for each new function for the API in *src/remote/remote_protocl.x*. The structs describe the parameters that are passed to the remote function and the value that is returned respectively. However, if the remote function returns just 0 or -1, as is the case in our running example, only the first struct is required. Next, an identifier for the new RPC must be reserved in the *remote_procedure enum* for each new function. The changes are shown in (Figure 5).

```
struct remote_domain_set_frequency_args {
      int core;
      remote_string frequency;
};

enum remote_procedure {
…
REMOTE_PROC_NODE_SET_FREQUENCY = 374,
};
```

Figure 5: Definition of the wire protocol of *virNodeSetFrequency* function

Finally, running 'make' in the *src/* directory is essential in order to create the .c and .h that are required by the remote protocol code. The generated .h files are used by the RPC client. The remote method calls are generated in *src/remote/remote_driver.c*. These files are also used by the server side remote dispatcher. The server side dispatcher is implemented in the *daemon/remote.c* (Figure 6). Its function is mainly to deserialize the RPC arguments and invoke the public API function implementing the service requested by the RPC on the node. After completing the three aforementioned implementation steps of the remote call, and having updated all the generated files, it is also essential to update the *src/remote_protocol-structs* file.

```
static int
remoteDispatchNodeSetFrequency(virNetServerPtr server ATTRIBUTE_UNUSED,
                               virNetServerClientPtr client,
                               virNetMessagePtr msg ATTRIBUTE_UNUSED,
                               virNetMessageErrorPtr rerr,
                               remote_node_set_frequency_args *args)
{
      int rv = -1;
      char *frequency;
      struct daemonClientPrivate *priv =
virNetServerClientGetPrivateData(client);

      if (!priv->conn) {
            virReportError(VIR_ERR_INTERNAL_ERROR, "%s", _("connection not
open"));

            goto cleanup;
      }

      frequency = args->frequency ? *args->frequency : NULL;
      if (virNodeSetFrequency(priv->conn, args->core, frequency) < 0)

      goto cleanup;
      rv = 0;

      cleanup:
            if (rv < 0)
                  virNetMessageSaveError(rerr);

      return rv;
}
```

Figure 6: Remote dispatch of virNodeSetFrequency function

3.1.4. Expose the New API in Virsh

A new command for each new function can be added in Virsh (Figure 3, step 3).

Each command needs two structs: (a) the *vshCmdInfo* struct, which contains information on the command and the help text, and (b) the *vshCmdOptDef* struct that contains the parameters that are needed by the function implementing the command. Also the new command has to be added in the command array. These changes take place into the *tools/* directory and the implementation of the new command should be added in the appropriate file according to the functionality of the new command (Figure 7).

```
static const vshCmdInfo info_frequency[] = {
      {.name = "help",
      .data = N_("set frequency")
      },
      {.name = "desc",
      .data = N_("Set frequency of the specific core.")
      },
      {.name = NULL}
};

static const vshCmdOptDef opts_frequency[] = {
      {.name = "core",
      .type = VSH_OT_INT,
      .help = N_("core id")
      },
      {.name = "frequency",
      .type = VSH_OT_STRING,
      .help = N_("frequency number")
      },
      {.name = NULL}
};

static bool
cmdNodeSetFrequency(vshControl *ctl, const vshCmd *cmd)
{
      int core;
      const char *frequency = NULL;
      int result;

      virshControlPtr priv = ctl->privData;


      if (vshCommandOptInt(ctl, cmd, "core", &core) < 0)
            return false;

      if (vshCommandOptStringReq(ctl, cmd, "frequency", &frequency) < 0)
         return false;

         if ((result = virNodeSetFrequency(priv->conn,core,frequency)) < 0)
            return false;

         vshPrint(ctl, "%d\n", result);
         return true;
}
```

Figure 7: Virsh command of *virNodeSetFrequency* function

3.1.5. Definition and Implementation of the Driver Methods

In this step, the new public API function is associated with a driver (Figure 3, step 4). We add the prototype of the new function to the header file of the hypervisor driver functions /src/driver-hypervisor.h. Moreover, we add the respective fields, which associate the new method with the function that implements it, to struct

virHypervisorDriver. The next step is to implement the functions at the drivers of hypervisors of interest (in our case QEMU/KVM).

More specifically, the functionality is added to QEMU and the function is added in */src/qemu/qemu-driver.c*. In order to implement the *virHostSetFrequency* that changes the frequency of the corresponding core, the perf-events API must be extended. Namely, the changes take place into the *virPerfRdtEnable* function (Figure 8). The fields of the *rdt_attr* struct are changed in order to support the current functionality and the syscall function is used with the appropriate parameters as shown below. Finally, for the activation of the event the respective ioctl function is called.

```
if (event->type == VIR_PERF_EVENT_CYCLES){
      memset(&rdt_attr, 0, sizeof(rdt_attr));
      rdt_attr.size = sizeof(rdt_attr);
      rdt_attr.type = PERF_TYPE_HARDWARE;
      rdt_attr.config = PERF_COUNT_HW_CPU_CYCLES;
      rdt_attr.inherit = 1;
      rdt_attr.disabled = 1;
      rdt_attr.enable_on_exec = 0;

      event->fd = syscall(__NR_perf_event_open,&rdt_attr, pid, -1, -1, 0);

      if (event->fd < 0) {
            virReportSystemError(errno,_("Unable to open perf type=%d for
            pid=%d"), event_type, pid);
            goto error;
      }

      if (ioctl(event->fd, PERF_EVENT_IOC_ENABLE) < 0) {
            virReportSystemError(errno,_("Unable to enable perf event for
            %s"), virPerfEventTypeToString(event->type));

            goto error;
      }

      event->enabled = true;
      return 0;
}
```

Figure 8: Bind *virNodeSetFrequency* function to perf

3.1.6. Libvirt Proposed Python Interface

OpenStack can extract information from hypervisor through Libvirt using the Python interface.

The *virConnect* class represents the connection to the hypervisor and it is identified by a Uniform Resource Identifier (URI). Libvirt supports both local and remote connections to hypervisors. The path and the scheme of the URI specify the hypervisor and the host part the location.

Local URIs:

- driver:///system
- driver:///session
- driver+unix:///system
- driver+unix:///session

Remote URIs:

- driver[+transport]://[username@][hostname][:port]/[path][?extraparameters]

The *virDomain* class represents the domain name. The domain can be specified via one of the following unique identifiers:

- *ID*: a positive integer that is unique for each domain within a single host. Can be used for active domains only.
- *Name*: a short string that is unique for each domain within a single host. Can be used for both active and inactive domains.
- *UUID*: 16 unsigned bytes, unique for each domain on any host.

For metrics already implemented within Libvirt, we plan to port the implementation to ARM / APM XGene2/3 adhering to the existing interface. For new metrics and control functionality of particular interest to the Uniserver project, Libvirt is going to be extended and the functionality is going to be offered to OpenStack through the following Python API:

- **virConnect::getHostCPUStatisticsUniserver(self)**

  getHostCPUStatisticsUniserver is used to obtain information about the CPUs. If successful it returns a list of the number of the CPUs, the reserved capacity reserved for throttling (%) and the frequency of each core (MHz). If it fails, **None** is returned

- **virConnect::getHostMemoryStatisticsUniserver(self)**

  getHostMemoryStatisticsUniserver is used to obtain information about the memory of the host. If successful it returns a list of the total memory size (MB), the available memory size (% and MB), the memory speed (MHz), the cached memory (% and MB) and the total memory swap space (MB). If it fails, **None** is returned.

- **virDomain::getDomainStatisticsUniserver(self)**

  getDomainsStatisticsUniserver returns the memory usage (% and MB), the VM interruption time to run system services for the domain or other domains (msec) and the wait time in swapping (msec). If it fails, **None** is returned.

- **virConnect::getDomainsStatisticsUniserver(self)**

  getDomainsStatisticsUniserver returns a list of the number of the domains, the number of the active domains, the total memory usage (MB) and the average wait time in swapping of all domains (msec). If it fails, **None** is returned.

- **virConnect::getHostSystemStatisticsUniserver(self)**

  getHostTimeUniserver is used to obtain information about the system. It returns a list of the average utilization of the CPU (%), the wait time (%), the idle time (%) and the system time (%), the utilization of each core (%) and the CPU time running at maximum utilization (msec.). If it fails, **None** is returned.

- **virConnect::getHostContentionStatistics(self)**

getHostContentionStatistics is used to obtain information about the system contention. It returns a list of the contention of the memory (% and MB) and the contention of the CPU (% and msec). If it fails, **None** is returned.

- **virConnect::getHostGovernor(self)**

  getHostGovernor returns a string of the governor of the system. If it fails, **Null** is returned.

- **virConnect::getHostPowerConsumptionUniserver(self)**

  getHostPowerConsumptionUniserver is used to monitor the power consumption of the system. It returns a list of the power consumption of the CPU (Watt) and the memory (Watt). If it fails, **None** is returned.

- **virConnect::getHostTemperatureUniserver(self)**

  getHostTemperatureUniserver is used to monitor the temperature of the system. It returns a list of the average CPU temperature (Celcius) and the cores temperature (Celcius). If it fails, **None** is returned.

- **virConnect::getCorrectableErrorsUniserver(self)**

  getCorrectableErrorsUniserver is used to obtain the number and the frequency of the correctable errors that occurred in the system. It returns a list of the total number and the frequency of L2, L3, DRAM and MCU correctable errors. If it fails, **None** is returned.

- **virConnect::getUncorrectableErrorsUniserver(self)**

  getUncorrectableErrorsUniserver is used to obtain the number and the frequency of the uncorrectable errors that occurred in the system. It returns a list of the total number and the frequency of L2, L3, DRAM and MCU uncorrectable errors. If it fails, **None** is returned.

3.2. Hypervisor API and Extension

The KVM API is a set of input/output control system calls (ioctls) that are issued to control various aspects of a virtual machine. The ioctls belong to three classes.

- *System ioctls*: These query and set global attributes which affect the whole KVM subsystem. In addition a system ioctl is used to create virtual machines.
- *VM ioctls*: These query and set attributes that affect an entire virtual machine, for example memory layout. In addition a VM ioctl is used to create virtual CPUs (vCPUs). Only run VM ioctls from the same process (address space) that was used to create the VM.
- *vCPU ioctls*: These query and set attributes that control the operation of a single virtual CPU. Only run vCPU ioctls from the same thread that was used to create the vCPU.

3.2.1. Useful Existing KVM APIs

Table 5 enlists some useful hypercalls to provide insights into the available system calls at the KVM level.

Table 5: KVM API Metric in Universer Project

Metric	Metric Information
KVM_GET_VCPU_MMAP_SIZE	Get size of vCPU mmap area
KVM_GET_DIRTY_LOG	Given a memory slot, return a bitmap containing any pages dirtied since the last call to this ioctl.
KVM_GET_REGS	Reads the general purpose registers from the vCPU.
KVM_GET_SREGS	Reads special registers from the vCPU.
KVM_GET_MSRS	Reads model-specific registers from the vCPU.
KVM_GET_FPU	Reads the floating point state from the vCPU.
KVM_GET_CLOCK	Gets the current timestamp of KVM clock as seen by the current guest.
KVM_GET_VCPU_EVENTS	Gets currently pending exceptions, interrupts, and NMIs as well as related states of the vCPU.
KVM_GET_DEBUGREGS	Reads debug registers from the vCPU.
KVM_GET_MP_STATE	Returns the vCPU's current "multiprocessing state" (though also valid on uniprocessor guests).
KVM_GET_TSC_KHZ	Returns the tsc frequency of the guest.
KVM_GET_LAPIC	Reads the Local APIC registers and copies them into the input argument.
KVM_PPC_GET_SMMU_INFO	This populates and returns a structure describing the features of the "Server" class MMU emulation supported by KVM.
KVM_PPC_GET_HTAB_FD	This returns a file descriptor that can be used either to read out the entries in the guest's hashed page table (HPT), or to write entries to initialize the HPT.
KVM_GET_REG_LIST	This ioctl returns the guest registers that are supported for the KVM_GET_ONE_REG/KVM_SET_ONE_REG calls.
KVM_S390_GET_SKEYS	This ioctl is used to get guest storage key values on the s390 architecture.
KVM_S390_GET_IRQ_STATE	This ioctl allows userspace to retrieve the complete state of all currently pending interrupts in a single buffer.

3.2.2. KVM API Extension

In addition to the above API and monitored metrics, as we said hypervisor will have access to the additional information that will be collected by the Health, Stress and Predictor daemons as specified in WP4 deliverables, i.e. D4.2 and D4.3. It is expected that a new set of functions will be implemented such as *get_healthlog_memory(dimm id)* for providing the relevant information vector (i.e. error type (correctable/uncorrectable), …) about the particular errors occurring in a specific DIMM when requested by the hypervisor. Similarly, functions for allowing the hypervisor to query the other additional Uniserver related modules for specific CPU/memory events will also be utilized.

4. References

[1] K. V. Vishwanath, A. Greenberg and D. A. Reed, “Modular data centers: how to design them?,” in *1st Workshop on Large-Scale System and Application Performance*.

[2] H. Irfan, “Virtualization with KVM,” *Linux J.,* p. 166, 2008.

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

[4] V. Sridharan and D. Liberty, “A study of dram failures in the field,” in *In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC ’12*.

[5] B. Schroeder and G. Gibson, “A large-scale study of failures in high-performance computing systems,” in *In Dependable Systems and Networks*, 2006.

[6] APM, “X-Gene: World’s First ARMv8 64-bit Server on a Chip Solution,” [Online]. Available: https://www.apm.com/products/data-center/x-gene-family/x-gene/.

[7] Intel Corporation, “Intel 64 and IA-32 Architectures Software Developers Manual. Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C,” September 2014.

[8] “ACPI Platform,” [Online]. Available: https://firmware.intel.com/sites/default/files/resources/A_Tour_beyond_BIOS_Implementing_APEI_with_UEFI_White_Paper.pdf.

[9] P. Nikolaou, Y. Sazeides, L. Ndreu and M. Kleanthous, “Modeling the implications of DRAM failures and protection techniques on datacenter TCO,” in *In Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*, 2015.

[10] “Linaro Kernel,” [Online]. Available: https://wiki.linaro.org/LEG/Engineering/Kernel/RAS.

[11] A. Kleen, “Mcelog: memory error handling in user space”.

[12] libvirt, “implementing a new API in libvirt,” [Online]. Available: http://libvirt.org/api_extension.html.

[13] OpenStack, “OpenStack Metering Using Ceilometer,” [Online]. Available: https://www.mirantis.com/blog/openstack-metering-using-ceilometer/.

[14] D. Hardy, M. Kleanthous, I. Sideris, A. Saidi, E. Ozer and Y. Sazeides, “An analytical framework for estimating tco and exploring data center design space,” in *International Symposium on Performance Analysis of Systems and Software*.

[15] “Linaro RAS git,” [Online]. Available: https://git.fedorahosted.org/git/rasdaemon.git.

[16] X. Li, K. Shen, M. C. Huang and L. Chu, “A memory soft error measurement on production systems,” in *In Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, 2007.

[17] A. Saleh, J. Serrano and J. Patel, “Reliability of scrubbing recoverytechniques for memory systems,” in *Reliability IEEE Transactions*.

[18] “Error Handling Driver,” [Online]. Available: http://lxr.free-electrons.com/source/drivers/edac/xgene_edac.c?v=4.3#L479.

[19] P. Org, “Python Interface Definition,” [Online]. Available: https://www.python.org/.

[20] L. Group, “Virsh Command Reference,” [Online]. Available: http://libvirt.org/virshcmdref.html.

[END OF DOCUMENT]