D5.3 2nd Report on Hypervisor / System / Software Interface

Contract number	688540
Project website	http://www.Uniserver2020.eu
Contractual deadline	Project Month 21 (M21): 31st October 2017
Actual Delivery Date	26 November 2017
Dissemination level	Public
Report Version	1.0
Main Authors	Lev Mukhanov (QUB), Bin Wang (QUB), Christos Antonopoulos (UTH), Georgios Karakonstantis (QUB), Srikumar Venugopal (IBM), Mustafa Rafique (IBM), Christos Kalogirou (UTH), Panos Koutsovasilis (UTH), Emmanouil Maroudas (UTH), Christos Antonopoulos (UTH), Spyros Lalis (UTH), Dimitrios Nikolopoulos (QUB), Hans Vandierendonck (QUB)
Reviewers	Andreas Diavastos (UCY), Peter Lawthers (APM)
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.

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**
1.0	Final version prepared with all inputs and review comments addressed for submission to EU.

Table of Contents

Index of Figures

Index of Tables

Executive Summary

This document describes the system software API (Application Programming Interface) developed in Task 5.3 within the Work Package 5 (WP5) of the UniServer Project Description of Action (DoA). This is in fulfilment of the Deliverable D5.3.

The UniServer project attempts to reduce hardware safety margins by utilizing representative stress cases, constant hardware monitoring and predictive mechanisms within commercial servers. To enable this, all software layers should monitor the state of the underlying hardware components, communicate relevant information and coordinate their activities for optimizing energy efficiency while not compromising system availability. This implies that interfaces should be introduced to integrate all software layers, including the hypervisor, OpenStack and all other low level UniServer software modules (i.e. HealthLog, StressLog and Predictor).

In the deliverable D5.1, we defined the preliminary system software interface across different layers and the target of this deliverable is to enhance the introduced interfaces after the development of the underlying hardware monitoring modules (HealthLog, StressLog and Predictor) in WP4 and the enhancement of the hypervisor and OpenStack layers in WP5 and WP6, respectively. In particular, D5.3 presents the enhanced system software interfaces across all involved layers, focusing on the information flow between the hypervisor, which is a central component within each UniServer node with OpenStack and the hardware monitoring and prediction modules.

1. Introduction

As we discussed in previous deliverables, the UniServer platform must be equipped with a complete software stack to efficiently manage the compute and storage resources of datacenters by offering easy installation, migration and replication of tasks, either at the node or server-rack level [1] in divergent use cases. At the hypervisor / OS layer, KVM [2] lends itself to the developed framework as KVM has numerous benefits provided by virtualization such as easier installation, replication, migration of tasks. At the upper layer, the OpenStack [3] framework is deployed for scheduling VM workloads.

Besides the hypervisor and OpenStack, UniServer introduces low level software modules for monitoring the hardware behavior under a wider range of Voltage / Frequency / Refresh rate (VFR) operating points through direct access to the underlying firmware and error detection hooks [4].

As we discussed in the deliverables of WP4, such modules include Health and Stress Daemons along with the Predictor. Such hardware monitoring add-on modules are essential within the UniServer ecosystem since they enhance the default stack with error monitoring capabilities that were not available before [4]. As a result, the system software needs to be enhanced with new mechanisms to effectively consume the collected low-level information on the hardware components to control and minimize the effects of potential faults. At the same time, the system software should minimize any overhead in order to not outweigh the benefits of operating at extended points. In the whole UniServer system, the workload variation of environmental conditions, chip aging etc. will dynamically determine the operating points to achieve energy efficiency very fast and reliably [5]. In this deliverable (D5.3), we present the enhanced system interfaces with functions that allow the communication of the collected information from the low-level software modules up to OpenStack.

In D5.1, we introduced the main components of the UniServer system, such as OpenStack, Libvirt, HealthLog, StressLog, the hypervisor, Predictor, collected metrics and general-purpose interfaces, while in this report, we focus specifically on the API for communication between OpenStack and Libvirt / hypervisor which allows us to operate processor and DRAM at the extended margins to gain power savings. Particularly, we introduce a set of reliability levels for processor and memory that enables OpenStack to control the operation point of the node and to obtain feedback on its performance. We also discuss the API to communicate with the hypervisor, HealthLog and Predictor.

1.1. Overview of the UniServer Cross-Layer System Architecture

Figure 1 shows the overall software system stack of the UniServer framework, which includes OpenStack, Libvirt, HealthLog, StressLog, hypervisor and Predictor. Openstack is a widely used open source middleware for cloud setups that pairs well with the popular enterprise and open source technologies. Our extended version of OpenStack includes support for monitoring VMs and determining their dynamically changing characteristics and virtual resource utilization at a finer granularity than the existing state-of-the-art. In particular, the Ceilometer component of OpenStack gathers various data about the health and performance of the underlying physical and virtual resources in the datacenter with the help of Hypervisor that gathers the requested information through the StressLog and HealthLog daemons.

OpenStack Nova has the responsibility to manage the resources of the physical hosts, to map and deploy incoming VMs to available nodes, and to maintain the ‘good health’ of all running VMs. In the context of UniServer, Nova is extended to configure nodes using more power-efficient voltage-frequency settings. This could involve running a node at the extended margins which could lead to increased probability of faults affecting the applications running inside the VMs. Therefore, the VM scheduler within Nova has been extended to consider the sensitivity of applications to system errors before mapping VMs to nodes running in different configurations as specified in D6.2.

We use Libvirt to bridge the gap between OpenStack and Hypervisor, which is a hypervisor-independent virtualization API. Libvirt is used to run VMs and transfers all information required by OpenStack from hypervisor. We extend this library with interfaces used to control a reliability level and collect reliability-related information, such as the number of ECC errors reported by hardware, for each node.

The next layer in our architecture is the hypervisor which manages a node and interact with all system software and hardware layers within the node. It is responsible for creating an appropriate execution environment for Virtual Machines (VMs) by manipulating the power/performance/reliability tradeoffs in an educated and safe manner. Specifically, it sets the system at a just-right configuration, which reduces the power footprint of each node by eliminating unnecessary hardware guard-bands, without introducing negative effects on the services running within the VMs. UniServer follows a hypervisor-based approach based on QEMU / KVM ported to the ARM64 architecture, to leverage all benefits of virtualization, such as easier deployment, administration, replication and migration, which are necessary for the targeted datacenters at the Edge of Cloud.

To enable reliability facilities for hardware operated at the marginal operating hardware settings, hypervisor interacts with HealthLog and Predictor.

Operating outside the nominal hardware settings may introduce transient hardware errors during the system's lifetime. We have extended the error reporting capabilities of existing mechanisms with system configuration values, sensor readings and performance counters. We call this mechanism the HealthLog monitor that records runtime system metrics in the form of an information vector, stored in a system logfile. The HealthLog monitor interacts and exchanges information with higher system layers (e.g. Predictor and hypervisor). The HealthLog monitor provides two types of services: (a) Event-driven services, where it will collect information based on event occurrences in the system (e.g. errors) and (b) On-demand services, where the monitor will respond to requests from higher layers for specific information.

The StressLog monitor is spawned either periodically during a machine lifetime or is triggered by higher system layers (Predictor) in the case of anomalous machine behaviour. In this case, the machine being tested will be taken offline and as soon as the monitor receives the input stress target parameters from the higher system layers, it will initiate the stress test scenarios. The StressLog monitor also includes a workload suite, consisting of different benchmarks and kernels that either represent real-life applications or are hand-coded to stress specific components of the system. During a stress test, the HealthLog monitor executes in parallel to record system events (errors, system values, sensors and performance counters). The StressLog monitor takes the output of HealthLog and wraps all information into a vector to be passed to the higher layers.

Predictor is a software that utilizes online data and offline characterization data to predict the probability of failure for non-nominal voltage frequency states and DRAM refresh rates. Particularly, given availability constrains and desired number of cores and operating frequencies; predictor estimates most energy efficient voltages and DRAM refresh states that don't violate the given constraints set by OpenStack. In the UniServer framework, Predictor communicates with HealthLog to monitor a node collecting all metrics discussed in D5.1 and StressLog to run the stress-testing when it is required.

At the bottom level of our architecture there is a Hardware Exposure Interface (HEI) module which provides access to the APM firmware through I2C bus. The firmware exposes a set of hardware sensors and registers that allow software to closely monitor and control the processor. The HEI module provides a demand notification mechanism whereby software can register for a series of events that will be triggered when certain conditions are met. For example, HealthLog can invoke Low Level Handler (Deliverable D4.5) to get the logs when the specified event occurs.

Note that such a system stack can be deployed at classical centralized on the Cloud as well as new emerging de-centralized datacenters at the Edge of the Internet. The same software stack without the OpenStack could also be deployed on small datacenters where there is no need for specialized resource management modules as well as on individual nodes. The interfaces introduced in this deliverable will be used for any possible deployment within the centralized and de-centralized datacenters as well as bare-metal deployments on individual nodes.

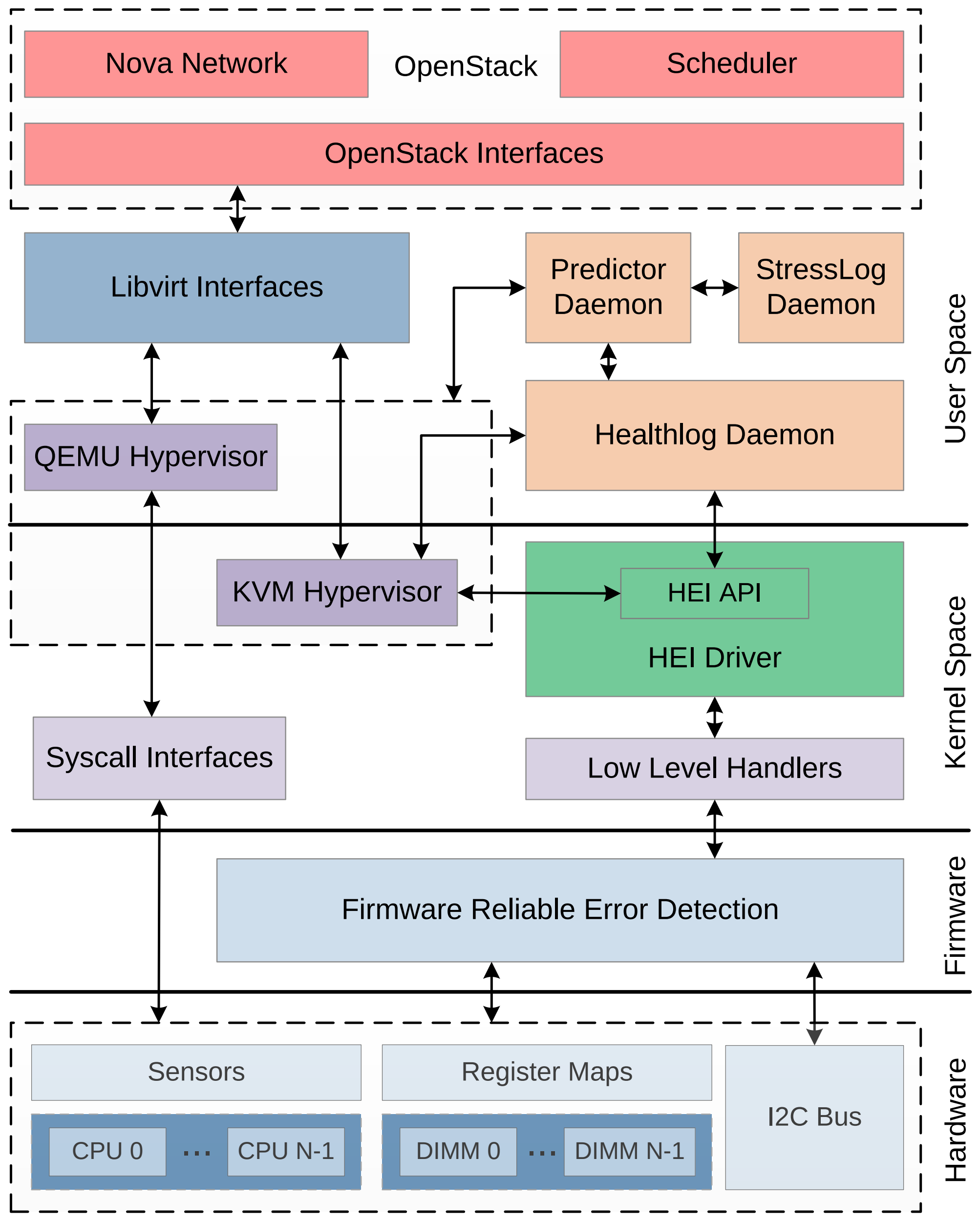

Figure 1: System software layers of the UniServer architecture

1.2. Organization

In Section 2, we define and extend the interface between Libvirt, OpenStack, and the QEMU / KVM hypervisor that is the target of this deliverable. In Section 3, hypervisor and system software interfaces are defined to deliver HealthLog information. Finally, in Section 4, we define interfaces between the Predictor and the hypervisor.

2. System Software Interface between Hypervisor and OpenStack

In this section, we present extensions of Libvirt made to enable communication between hypervisor and OpenStack.

2.1. Overview of Libvirt API Extensions

Libvirt is a hypervisor-independent virtualization API [6] and toolkit used to leverage the virtualization capabilities for a range of different operating systems. Libvirt already supports numerous configuration options, and manages virtual machines as well as VM storage and network. It supports different virtualization hypervisors, among them the KVM/QEMU, and it offers bindings in several different languages, such as Python. OpenStack is responsible for an efficient allocation of VMs in a clustered deployment. In order to achieve this, it requires information from the nodes [7]. To this end, we have extended Libvirt in order to propagate information such as resource availability (CPU, RAM) of the node to OpenStack, as well as the resources that are allocated for the VMs in a node.

In the context of Uniserver, we are able to operate processor and DRAM at the extended margins. We have modified Libvirt so that OpenStack can request the node to operate at these margins. Also, OpenStack can retrieve information related to operation of the node, such as power consumption, temperature and any errors that occur, and use it to take better high-level resource management decisions in the future. Figure 2 gives a high-level view of the interaction process between OpenStack and hypervisor through Libvirt.

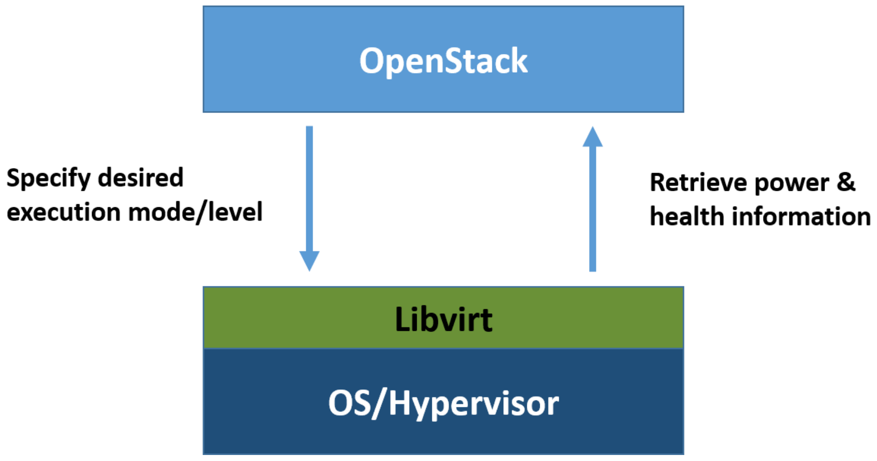
OpenStack
Specify desired execution mode/level
Retrieve power & health information
Libvirt
OS/Hypervisor

Figure 2. Information exchange and requests between OpenStack and hypervisor.

OpenStack is typically responsible for managing a very large number of nodes, and cannot deal with low-level resource management decisions at the level of individual cores, PMDs and DIMMs [8]. To allow for a high-level management of nodes that can scale for a large number of nodes, we let OpenStack configure an entire node at a coarse granularity. More specifically, it can set the node to operate at either nominal or extended margins, referred to as *nominal mode* and *extended margins mode*, respectively. The latter mode allows the node to be more energy-efficient but also increases the probability of having errors or crashes. If desired, the operating mode can be set separately for CPU and DRAM. Also, to allow for even greater flexibility, the mode can be set per VM that runs on the node. While this may not be practically useful for very large installations with a very large number of nodes that run numerous different VMs, it could be exploited in smaller or stand-alone installations where the number of VMs could be smaller and the high-level resource management entity could maintain workload-specific profiles and set the mode accordingly at VM (or VM bundle) granularity.

2.2. Extended Libvirt API for UniServer Operation

Table 1 summarizes new primitives that constitute the extended Libvirt for UniServer operation.

Table 1. Primitives of the extended Libvirt.

Primitive	Brief Description
getSystemStatisticsUniserver	Returns system information of a node.
getContentionUniserver	Returns the contention of CPU and RAM of a node.
getGovernorUniserver	Returns the governor.
getDomainStatisticsUniserver	Returns information for a specific domain.
getDomainsStatisticsUniserver	Returns information for all the domains of a node.
getPowerConsumptionUniserver	Returns the power consumption of a node.
getTemperatureUniserver	Returns the temperature of a node.
getErrorsUniserver	Returns the errors that have occurred at the node.
getModeCapabilityUniserver	Returns the capability of a node to operate its CPU and/or RAM at extended margins.
getCurrentModeUniserver	Returns the current mode for the CPU and/or RAM.
setModeUniserver	Requests for a mode for the CPU and/or RAM.
getCPUStatisticsUniserver	Returns CPU information of a node.
getMemoryStatisticsUniserver	Returns memory information of a node.

Below, we describe each primitive in more detail.

2.2.1. *getSystemStatisticsUniserver*

virConnect::getSystemStatisticsUniserver(self)	
no input parameter	-
return value	Dictionary – System information of the node. <u>output:</u> system_metric: [time_spent (msec), time_spent (%)]

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

2.2.2. getContentionUniserver

virConnect::getContentionUniserver(self)	
no input parameter	-
return value	Dictionary – Information about the contention of CPU and RAM of the node. <u>output:</u> contention_RAM: [percentage (%), size (KB)] contention_CPU: [percentage (%), time (msec)]

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

2.2.3. getGovernorUniserver

virConnect::getGovernorUniserver(self)	
no input parameter	-
return value	Dictionary – The governor of the node. <u>output:</u> coreX: [governor]

getGovernorUniserver returns a dictionary of the governor of the processors of a node. If it fails, *None* is returned.

2.2.4. getDomainStatisticsUniserver

virDomain::getDomainStatisticsUniserver(self)	
no input parameter	-
return value	Dictionary – Information for a specific domain. <u>output:</u> domain_name: [memory_usage (KB), memory_usage (%), interruption_time (msec), wait_time (msec)]

getDomainStatisticsUniserver returns a dictionary of the memory usage (% and KB), 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.

2.2.5. getDomainsStatisticsUniserver

virConnect::getDomainsStatisticsUniserver(self)	
no input parameter	-
return value	Dictionary – Information for all the domains in a node.

	output: domains: [domains_number] active_domains: [active_domains_number] memory_usage: [total_memory_usage_of_domains (KB)] wait_time: [average_wait_time_of_domains (msec)]

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

2.2.6. getPowerConsumptionUniserver

virConnect::getPowerConsumptionUniserver(self)	
no input parameter	-
return value	Dictionary – Information about the power consumption of the system. output: component: [power (Watt)]

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

2.2.7. getTemperatureUniserver

virConnect::getTemperatureUniserver(self)	
no input parameter	-
return value	Dictionary – Information about the temperature of the system. output: CPU: [temperature (Celcius)] coreX: [temperature (Celcius)]

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

2.2.8. getErrorsUniserver

virConnect::getErrorsUniserver(self)	
no input parameter	-
return value	Dictionary – Information about the errors of the system. output: component: [number_of_correctable, frequency_of_correctable, number_of_uncorrectable, frequency_of_uncorrectable]

getErrorsUniserver is used to monitor the errors of the system. It returns a dictionary of the number of the errors (UnCorrectable / Correctable) and the number of them for the last minute for different components of the system. If it fails, *None* is returned.

2.2.9. getModeCapabilityUniserver

virConnect::getModeCapabilityUniserver(self)	
no input parameter	-
return value	Dictionary – Capability of each component to operate at extended margins. 0 – not capable to operate at extended margins. 1 – capable to operate at extended margins. output: component: [capability]

getModeCapabillityUniserver is used to show the capability of the CPU and the RAM to operate at extended margins. It returns a dictionary of the capability of each component. If it fails, *None* is returned.

2.2.10. getMCurrentModeUniserver

virConnect::getCurrentModeUniserver(self,VMsignature)	
VMsignature	String – The VM signature. ALLSYSTEM – a special string that requests a mode for the entire system
return value	Dictionary – Per VM current mode of each component. 0 – nominal operating point. 1 – extended margins. output: VMSignature: [CPU, RAM]

getCurrentModeUniserver is used to obtain the current mode for a node. It returns a dictionary of the mode of each VM for CPU and RAM. If it fails, *None* is returned.

2.2.11. setModeUniserver

virConnect::setModeUniserver(self, VMsignature, component, mode)	
VMsignature	String – The VM signature. ALLSYSTEM – a special string that requests a mode for the entire system.
component	Integer – The component to switch mode. 0 – CPU 1 – RAM
Mode	Integer – The mode to request. 0 – nominal operating point 1 – extended margins

return value	Integer – Result of the call. -1 – failure of the call 0 – success

setModeUniserver is used to request for a mode for the CPU or the RAM for a specific VM for the entire system. It returns an integer of the result of the call.

2.2.12. getCPUStatisticsUniserver

virConnect::getCPUStatisticsUniserver(self)	
no input parameter	-
return value	Dictionary – CPU information of the node. output: cores: [Number_of_cores] coreX: [nominal_frequency (MHz), actual_frequency (MHz), reserved_capacity_for_throttling (%), utilization (%)]

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

2.2.13. getMemoryStatisticsUniserver

virConnect::getMemoryStatisticsUniserver(self)	
no input parameter	-
return value	Dictionary – Memory information of the node. output: total: [size (KB)] free: [size (KB), percentage (%)] cached: [size (KB), percentage (%)] swap: [size (KB)] speed: [speed (MHz)]

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

3. System Software Interface between Hypervisor and HealthLog

The purpose of the communication between hypervisor and HealthLog is for the former to provide the latter with higher level hints about failures and malfunctions that are detected/experienced at the level of hypervisor. These hints carry information about suspicious events happening in the system such as kernel warnings or kernel/application crashes that may result from operating at extended margins. This kind of information is complementary to the existing hardware events, and can be used to improve the intelligence of core mechanisms of the system, e.g., by predicting a next safe operational configuration and avoiding unsafe extended margins, or initiating proactive actions before system failure.

3.1. Overview

The primary component of the interface between hypervisor and HealthLog is a new character device under the */dev* system directory. Hypervisor writes messages to that device whenever it detects some deviation from the expected/normal behaviour. This device is read-only from user-space (one-way communication) and allows only one reader (HealthLog) at any time. This is more a precautionary decision than a limitation in order to ensure that all messages are delivered to one recipient (typically, HealthLog).

There is also an optional component that gives write access to the device from user-space. This feature, apart from debugging, can be used from other monitoring tools (aside from hypervisor) in order to provide their own feedback/hints to HealthLog. This write-only endpoint uses the *sysfs* subsystem of the Linux kernel. Figure 3 shows the communication layout and connectivity between hypervisor and HealthLog.

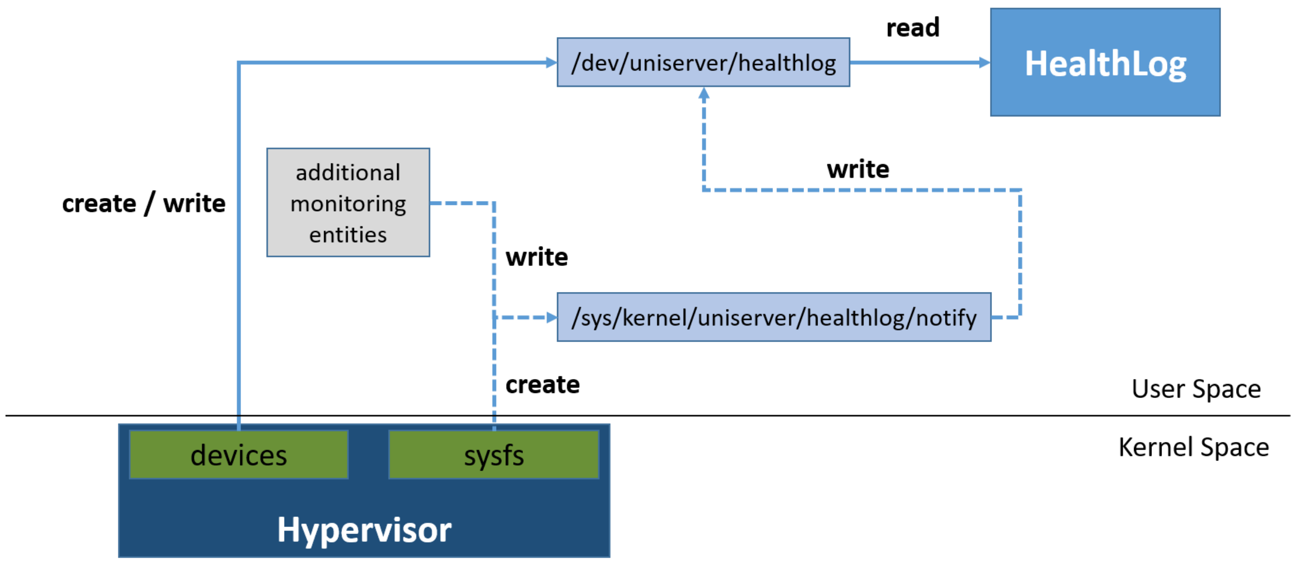

Figure 3. Communication layout

3.2. IPC Objects

Table 2 shows the IPC objects used for the communication between hypervisor and HealthLog, and their attributes.

Table 2. Hypervisor - HealthLog communication channels

Component	Attributes	
	kernel-space	user-space
/dev/uniserver/healthlog	Write-only, FIFO	Read-only, single reader, FIFO
/sys/kernel/uniserver/healthlog/notify		Write-only, multiple writers, FIFO

The maximum number of messages and length of each message is configured at build-time. As we do not expect many suspicious events when operating in just-right extended margins, and assuming that HealthLog monitors this communication channel for new messages often enough, the default capacity of the device is kept rather small. The maximum length of messages depends mostly on the size of practically useful information that HealthLog can reason about. Table 3 shows the current default values.

Table 3. HealthLog device configuration (default values)

Type	Symbol name	Value
Int	MAX_MSG	8
Int	MAX_MSG_SIZE	1024 bytes

3.3. Payloads / message Format

The messages that are sent by hypervisor to HealthLog via this channel are in binary format, following the structure shown in Table 4.

Table 4. Message format

Type	Symbolic name	Value: Description
uint16_t	HT_ERROR_TYPE	int: HT_SOFT_ERROR _KERNEL int: HT_SOFT_ERROR _USER Unique HealthLogError identifiers
uint16_t	HT_SOFTWARE_EVENT_ID	int: SE_CRASH int: SE_WARN int: SE_ILLEGAL_INSTR int: SE_CPU_STALL int: SE_OTHER Unique software event identifiers
uint16_t	RAW_DATA_SIZE	The actual size of additional data that may follow for each specific type of reported error. The limit is MAX_MSG_SIZE - 3*sizeof(uint16_t). The current default value is 0, but it can be easily changed to support extended information exchanges.
char *	RAW_DATA	The additional type-specific data.

4. System Software Interface between Hypervisor and Predictor

This section defines the communication interface and protocol between hypervisor and Predictor. Predictor is responsible for providing hypervisor with online system configurations based on hypervisor requests. These requests can be done when hypervisor seeks opportunities to change the profile of the system to be either more performance- or energy-efficient driven or when hypervisor needs extra confirmation whether the system under some circumstances stays into the zone of safe operating margins.

4.1. Overview of Hypervisor-Predictor API

The interface is designed to support a strictly sequential interaction: (i) hypervisor sends/writes the request, (ii) Predictor waits until a request is available, and then receives/reads the request, and finally (iii) Predictor sends/writes the response. In order to proceed with the next interaction, the previous one must be fully completed. In other words, hypervisor cannot issue the next request before having received the reply for the previous request. Any other usage may lead to undefined behavior.

To facilitate the communication, hypervisor employs *sysfs* capabilities of Linux Kernel. Specifically, *sysfs* is a pseudo file system that exposes various kernel subsystems to the user space through virtual files.

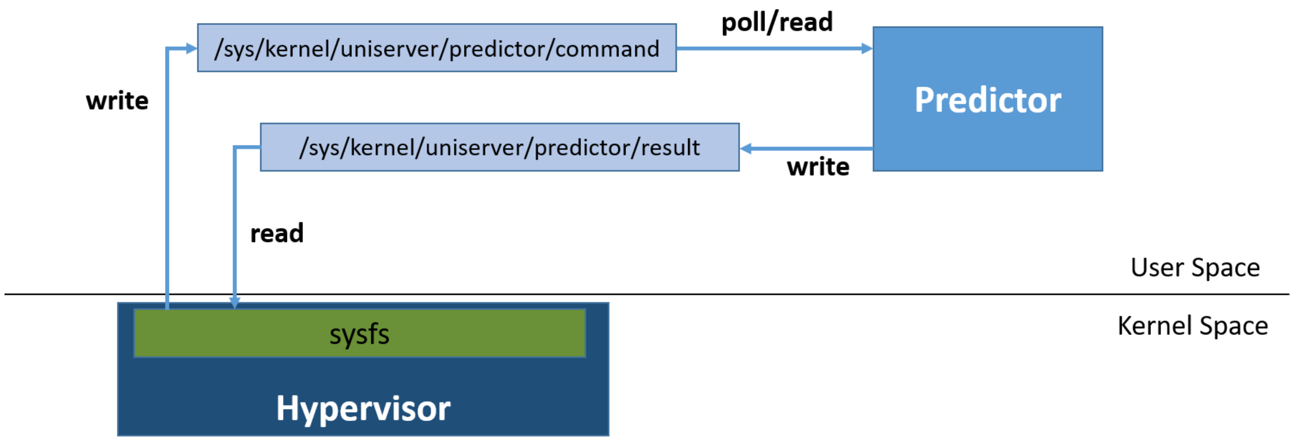
/sys/kernel/uniserver/predictor/command
poll/read
write
Predictor
/sys/kernel/uniserver/predictor/result
write
read
User Space
Kernel Space
sysfs
Hypervisor

Figure 4: Hypervisor – Predictor communication overview

As shown in Figure 4, there are two dedicated virtual files for communication between hypervisor and Predictor. The first virtual file (*/sys/kernel/uniserver/predictor/command*) acts as a trigger which informs Predictor through the poll system call that there are available requests from hypervisor. Afterwards Predictor is responsible for reading this file and servicing the hypervisor requests. The second virtual file (*/sys/kernel/uniserver/predictor/result*) is the buffer where Predictor writes the corresponding response/results for the last request received. A detailed presentation of the formatting and the payloads for each request is given in the sequel.

4.2. IPC Objects and Primitives

Table 5 summarizes the IPC objects and primitives/operations used for communication between hypervisor and Predictor. Below, we describe each primitive in more detail with respective payloads.

Table 5. Primitives of the Predictor-hypervisor API.

Primitive	Brief Description
Poll /sys/kernel/uniserver/predictor/command	When poll succeeds this means there are pending requests by hypervisor for Predictor.
Read /sys/kernel/uniserver/predictor/command	After a successful poll, the Predictor needs to read the file so it can get the request details from hypervisor
Write /sys/kernel/uniserver/predictor/result	The Predictor must write the results of a hypervisor request in this file.

4.3. Payloads / message Format

Hypervisor requests information and Predictor replies are sent as ASCII-formatted messages (strings). If hypervisor decides to retrain the system through stress tests, it notifies the Predictor by issuing a special one-way notification message.

The individual values for each message field are delimited by a single space character. In Tables Table 6, Table 7, Table 8, Table 9 and Table 10, we report the intended/interpreted values of each message field.

Table 6. Hypervisor request: Predictor systems settings

Name	Value Type	Optional	Comment
Request Type	Int	No	1
PSDC	Int	No	Fix decimal points: 2
PAppCrash	Int	No	Fix decimal points: 2
PSysCrash	Int	No	Fix decimal points: 2
Cores Frequency*	int[]	No	Multiple values vector, in kHz
DRAM Frequency	Int	No	In kHz
SoC Frequency	Int	No	In kHz

*The size of this array/sequence is equal to the number of cores available in the system. It is assumed that both hypervisor and Predictor are aware and use the same/correct value for this.

Table 7. Predictor response to system settings

Name	Value Type	Optional	Comment
Core Voltage	Int	No	In mV
Soc Voltage	Int	No	In mV
DRAM Voltage*	int[]	No	In mV
DRAM Refresh Rate	Int	No	In us

* The size of this array/sequence is equal to the number of DIMMs available in the system. It is assumed that both hypervisor and Predictor are aware and use the same/correct value for this.

Table 8. Hypervisor request: Is systems settings safe?

Name	Value Type	Optional	Comment
Request Type	Int	No	2
Cores Frequency	int[]	No	Multiple values vector, in KHz
DRAM Frequency	Int	No	In KHz
SoC Frequency	Int	No	In KHz

Table 9. Predictor response to safe or not

Name	Value Type	Optional	Comment
Answer	Boolean	No	1 or 0

Table 10. Hypervisor request: retrain the system with stress tests

Name	Value Type	Optional	Comment
Query Type	Int	No	3

The Predictor does not respond to this special request (this is a one-way notification).

5. Conclusion

The UniServer project attempts to reduce hardware safety margins by utilizing representative stress cases, constant hardware monitoring and predictive mechanisms within commercial servers. The complete system stack approach includes a modified error-resilient hypervisor and a cloud resource management software all being ported on a state of-the-art ARMv8 based microserver.

In this report, we present the overall system software stack of the UniServer ecosystem and interfaces across the different layers. We discuss in detail the implemented API for facilitating the communication of the required information between different layers, i.e. Libvirt, OpenStack, Predictor and HealthLog daemons. Finally, we introduce reliability levels controlled by OpenStack to relax specific hardware settings on a specific node which makes it possible to gain power savings.

The introduced API can facilitate the utilization of the UniServer modules on any state of the art server facilitating the deployment on classical centralized and de-centralized datacenters or individual nodes.

6. 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] 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/.

[5] 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*.

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

[7] libvirt, "implementing a new API in libvirt," [Online]. Available: http://libvirt.org/api_extension.html.

[8] 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*.

[9] OpenStack, "Open Stack Nova documentation," [Online]. Available: https://docs.openstack.org/nova/pike/.

[END OF DOCUMENT]