
uniserver

# D6.4 – OS Support for Standalone Micro-Server Deployments

| | |
|---|---|
| Contract number | 688540 |
| Project website | http://www.UniServer2020.eu |
| Contractual deadline | Project Month 27 (M27): 30th April 2018 |
| Actual Delivery Date | 1st June 2018 |
| Dissemination level | Public |
| Report Version | 1.0 |
| Main Authors | Panos Koutsovasilis (UTH), Christos Kalogirou (UTH), Christos Antonopoulos (UTH), Srikumar Venugopal (IBM), Konstantinos Tovletoglou (QUB), Lev Mukhanov (QUB), Christian Pinto (IBM) |
| Contributors | Evangelia Malami (UTH), Panagiotis Vlastaridis (UTH), Georgios Karakonstantis (QUB), Spyros Lalis (UTH) |
| Reviewers | Srikumar Venugopal (IBM), Georgios Karakonstantis (QUB), Nikolaos Bellas (UTH), Arnau Prat (SPA) |
| Keywords | edge micro-server, CPU power efficiency, DRAM power efficiency, resilience, protection |

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.

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 |
|---|---|
| 1.0 | Final version prepared for submission with all inputs integrated and review comments addressed |

# Table of Contents

# INDEX OF FIGURES

# INDEX OF TABLES

# Executive Summary

This document describes the functionality implemented to enhance the resilience of standalone edge micro-servers when operating at extended margins. As a first step, we outline error mitigation techniques (both conventional and those designed and implemented in the context of UniServer) which are applied in cloud datacenters and discuss why most of them are not applicable in standalone micro-server deployments at the edge of the network. Due to the significantly limited protection arsenal in standalone setups, configuration of all hardware components (CPUs, memories) at extended margins needs to be more conservative, compared with the respective configuration when the micro-server is part of a datacenter. Moreover, the UniServer system software stack needs to provide mechanisms that can be used to proactively increase the resilience of the system.

For CPU components, we implement a software module which operates at user-level as a system daemon and applies configurations at extended margins in a conservative, educated manner. The applied undervolting levels are based on the findings of offline characterization and/or online monitoring of the interaction of software with the underlying hardware. Voltage levels are conservatively adjusted either proactively, or as a reaction to observed erratic behavior. The same software module can additionally be used to introspect the health status of end-user services running on the node. Moreover, in order to protect the OS/Hypervisor, we provide functionality for the execution of critical OS/Hypervisor functionality on cores configured at reliable, nominal operating points. Finally, we discuss the design and implementation of an external hardware module that acts as a watchdog (and reboots the micro-server in case of crash) and provides administrators with information on the health of the node and end-user services.

For DRAM components, we implement support of NUMA domains for the ARMv8 architecture. The latter is used as the basis to implement the mechanisms required to manage heterogeneous reliability memory and expose it to system- and application-software. Heterogeneous reliability memory (i.e. memory where parts of the physical address space are accommodated by DRAMs configured at nominal settings and are thus reliable, and parts by DRAMs configured at extended margins and are thus more power efficient yet less reliable) is necessary in standalone deployments, as it allows programmers to protect data structures of the OS/Hypervisor and selectively protect critical data of applications from corruption. The implementation of heterogeneous reliability memory on the XGene 2 requires the de-activation of memory interleaving. In order to outweigh the performance overhead of a non-interleaved memory system we experiment with software interleaving. Finally, we evaluate the applicability and effectiveness of our heterogeneous memory framework in terms of performance, average and maximum power consumption, and energy efficiency.

# 1. Introduction

A cloud datacenter typically consists of thousands of nodes that are interconnected with a high-speed network. This complex hardware system is controlled by a software infrastructure, which is responsible for monitoring the status of the system, managing the available resources and optimizing operation according to different metrics, both high-level – such as quality of service (QoS), reduction of the total cost of ownership and increase of profit margin – and low-level – such as throughput, performance, reliability, and power efficiency.

Performance, energy efficiency and reliability are already first order concerns in cloud computing. UniServer aspires to disrupt the existing power efficiency / reliability / performance tradeoff by selectively allowing hardware to operate at extended margins, thus limiting the inherent Voltage / frequency guardbanding introduced by hardware manufacturers to enhance reliability at the expense of power / performance efficiency. Therefore, cloud management software is extended with mechanisms and policies to: a) reduce the power consumption by managing efficiently the resources and exploiting both conventional and unconventional (operation at extended margins) power reduction techniques; and b) overcome and minimize the effects of any potential failures, by exploiting protection mechanisms and policies that try to prevent such failures, or – if this is not possible – to at least reduce their impact.

In the context of UniServer, we extend and use OpenStack [1] to manage our cloud setup. OpenStack uses an intermediate software layer (Libvirt [2]) to communicate with the hypervisor of the underlying nodes. Both the original and the UniServer version of OpenStack offer a wide range of protection mechanisms, such as migration and data replication, that can deal with failures. Moreover, UniServer OpenStack can collect detailed information about the health status of the nodes and it can explicitly request node configuration at more power efficient / less reliable points through the extended version of Libvirt.

Beyond cloud datacenters, UniServer architectures of interest span fog deployments as well. Such deployments often include standalone micro-servers. Although, similar to cloud datacenters, the major target of improving power & performance efficiency without disrupting reliability or QoS holds in fog/edge micro-server environments, there are two major differences:

a) In standalone deployments each node is fully responsible for itself; there is no centralized software to monitor and control operation, and to act as a frontend that receives new workloads and assigns them to nodes, having the opportunity to command a re-configuration of the node in advance if necessary.. This requires extra intelligence on behalf of the system software at the node-level in order to identify the sweetspot in the tradeoff between energy efficiency and the reliability of the system, and to react to unexpected changes to the workload.
b) ) The arsenal of mechanisms for protection against failure and for recovery is significantly more limited. For example, techniques such as data replication and cold or warm standby of backup services are popular in cloud deployments to protect against node crashes. However, most of these techniques assume the availability of multiple independent nodes, which accommodate the replicas, standby backup services etc., which is not the case in standalone edge setups. Therefore, system software on standalone servers needs to be conservative when operating at extended margins, as manifested errors will lead – with high probability – to service disruption.

This deliverable discusses the design and implementation of the software infrastructure needed for operation at extended margins in standalone deployments. The respective infrastructure is implemented as a combination of in-hypervisor support and an external software module, operating as a system daemon at the user-level. This system software infrastructure is fully responsible for managing and configuring the node and implementing resilience mechanisms and policies. The daemon communicates with the hypervisor using the same extended interface exposed by Libvirt to OpenStack.

The rest of the document is organized as follows:

Section 2 motivates the necessity of additional functionality to enable operation at extended margins in standalone micro-server deployments. We briefly discuss mechanisms applied to protect services running on cloud infrastructure. Some of them were offered by the vanilla implementation of OpenStack, whereas others were introduced by the UniServer consortium. We find that most of those mechanisms are not applicable (or practical) in standalone micro-server deployments and discuss why this is the case.

In Section 3 we focus on CPU management. We introduce a voltage/frequency governor which takes into account runtime metrics and core utilization and uses a prediction model to dynamically set the CPU to a safe, “just right” extended voltage / frequency configuration. In this area of the configuration space we attain power gains without observably increasing the probability of errors. We discuss the safety vs energy efficiency tradeoff, and how it is steered towards additional safety in a standalone deployment. Another technique used to limit the susceptible to errors surface of OS/Hypervisor code is the migration of the respective functionality to cores configured in nominal settings. Finally, we discuss the design and implementation of an external watchdog for XGene 2. XGene 2 does not include a Baseboard Management Controller (BMC) that would allow out-of-band monitoring and management of a node. This functionality is necessary in remote standalone deployments, as it allows external monitoring of system health and automatically reboots the system if and whenever necessary.

In Section 4 we discuss the design and implementation of the infrastructure to support a memory system characterized by heterogeneity in terms of reliability and power efficiency. DRAMs serving different ranges of the physical address space can be configured to operate at different voltage levels and refresh rates. This heterogeneity is exposed to system and applications software. The design and implementation of heterogeneous memory support is a non-trivial undertaking, yet it is particularly important in stand-alone deployments. In cloud deployments, there are a multitude of data protection mechanisms. Therefore, the cloud management software may opt to configure the whole physical memory of the system to a less reliable operating point. However, in standalone environment this is not the case; it is necessary to provide reliable volatile (DRAM) storage for important system and application software data.

Finally, Section 5 concludes the document.

# 2. Motivation: Why are protection mechanisms for cloud deployments not applicable on standalone micro-servers

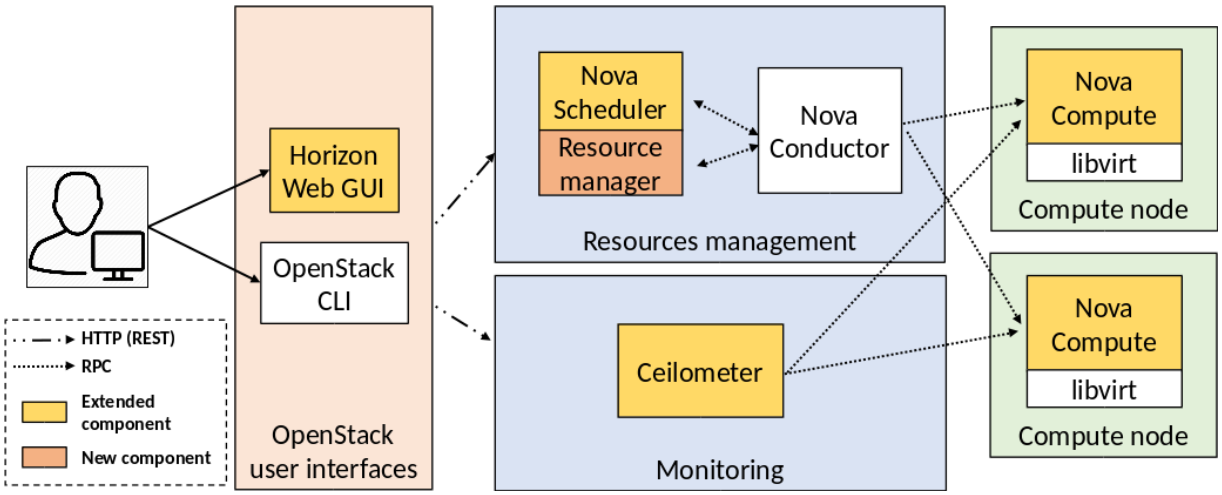


**Figure 1: OpenStack High level architecture highlighting new and extended components developed for UniServer**

OpenStack is an open source software for managing data center infrastructure that includes compute machines (servers), storage arrays, and networking equipment. The UniServer project has developed techniques to enable energy efficient, yet high performance micro-servers. An important technique explored by the UniServer project has been the exploitation of voltage states outside of the manufacturer-imposed guardbands that are more energy or performance efficient. The trade-off for running a server at these extended margins is to accept an increase in the probability of errors and failures, even if minimal.

Modifying OpenStack to support UniServer techniques has been a key focus for the project. Therefore, it is now possible for OpenStack to manage a data center equipped with UniServer-enabled servers so that efficiency gains can be realized at an aggregate level. The respective work has been described in deliverables D6.1, D6.2 and D6.3. Ongoing research in Work Package 6, which will be reported in deliverable D6.5, is exploring fault tolerance techniques so that the increase in the failure probability can be handled at the data center management level.

Most of these new OpenStack capabilities for managing server resources and mitigating the effects of failures assume that the cloud framework has multiple nodes under its control as normally expected in a cloud deployment. Therefore, they are not realistic or even not applicable in a standalone server scenario. However, the standalone scenario requires only the management of a single server instance and protecting the workload running on it. In the following sections, we will review the capabilities provided by OpenStack and its extensions developed by UniServer for meeting the requirements of data center management. We'll then contrast these requirements in the context of standalone mode and advise which of these capabilities meet the requirements of operating a UniServer machine in standalone mode.

## 2.1. Monitoring and Resource Management

The following new capabilities have been developed for OpenStack as part of UniServer (Figure 1):

1. Extending Nova, the compute service of OpenStack, to collect and provide information on the resource utilization of a Virtual Machine running on a compute node, the number of correctable and uncorrectable errors caught by that node and CPU/memory power consumption. This information is attained through Libvirt.
2. Extending the telemetry component of OpenStack, Ceilometer, to receive and digest the information collected by Nova. This allows the OpenStack compute controller to estimate the energy utilization of CPU under the current workload, as well as to assess if any nodes are at the risk of failure.
3. Extending Nova Scheduler and Compute Manager to set the operating point (voltage) of a particular node between nominal (within guardband) and extended (outside the guardband) modes. This allows

the compute manager to set a subset of datacenter resources to run in power efficient mode and thereby, save on energy costs.

4. Implementing scheduling algorithms in Nova to continuously map incoming VMs, according to their priority, to nodes running in either nominal or extended mode. The key tradeoff here is the energy saving produced for the entire datacenter versus the possibility of failure of workloads running on extended mode machines.

OpenStack is tuned to operate on clusters of servers and all decisions are made at the cluster level based on information aggregated from individual nodes. OpenStack observes behavior and makes workload assignment decisions at the granularity of nodes. It does not delve into intra-node details, such as controlling scheduling within the node, quantifying the sensitivity of individual hardware components within the node, or reaching operating point configuration decisions for operation at extended margins. The capabilities of aggregating information and scheduling workloads across nodes are not required in the standalone mode. However, the need to switch a node from nominal to extended mode and vice versa is relevant in the standalone scenario as well.

## 2.2. Migration

One of the main techniques available in the cloud to both facilitate resource management and protect the workload is the migration of virtual machines. This functionality is available in the stock distribution of OpenStack. When used for protection, this technique moves one VM from a faulty – or suspected to soon become faulty – node to another one that is functioning correctly. This workload protection is particularly useful for stateful applications that would not accept the failure of one instance or that have a costly recovery-from-reboot process.

The migration of a virtual machine involves a source node, the one currently running the VM to migrate, and a destination node to host the VM after its migration. Migration in its naïve implementation (also known as cold migration) involves stopping the virtual machine, copying its whole state to the destination node over the network and resuming execution. The state of a virtual machine is composed by the CPU state, the memory and block devices (e.g., disks), should the filesystem not be shared across nodes. However, the above approach often introduces long disruption of the service being migrated, as the VM is paused for the whole time needed to copy the state to the destination node. This is particularly observable when a VM has large memory or disk footprint. To overcome this problem, OpenStack provides the capability to *live migrate* VMs [3], i.e., migrating a virtual machine without interrupting its normal operation. The hypervisor is capable of marking the memory pages belonging to a virtual machine to track them during the copy. Memory pages are copied to the destination node while the source VM is still functional, and only when the majority of the memory pages is already copied is the VM paused for the actual migration. In most cases this approach stops the virtual machine for a negligible amount of time, needed to finalize the copy of the last few memory pages and move CPU state (that cannot be done while the VM is operational). The time to perform a live migration can vary according to the memory access (more specifically memory-write) pattern of the VM, as pages need to be re-copied to the destination if the application writes them during the migration. However, in general, live migration is sensibly faster than naïve, cold migration and introduces a smaller disruption of the service, if any at all [4].

VMs migration is not to be considered the Holy Grail for the protection of a workload, as it is effective only if failures can be predicted ahead of time, before a node starts malfunctioning and causes irreparable damage to the state of the workload. As a matter of fact, VM migration is mostly used in cloud production environments to move virtual machines out of nodes that are overloaded or are going under a maintenance cycle.

At the system level, the requirements for a successful VM migration are:

1) At least two compute nodes in the cluster to act as source and destination.

2) A destination node with enough resources (memory, CPU, disk) to host the virtual machine being migrated.

3) A network connection between source and destination nodes.

In OpenStack live migration is handled by Nova. The call of the service that executes the migrations takes two arguments, namely the ID of the VM and the destination host. It is also possible to dynamically select the destination host by omitting the second parameter of the call.

Migration, either cold or live, is not applicable in a completely standalone configuration. It may be possible in the case of standalone machines that are connected over a local or wide area network, but the high latency and constrained bandwidth in such scenarios may render migration too inefficient to be practically applicable. Moreover, two of the reasons for deploying micro-servers at the edge of the network is the large volume of data produced by the application at the edge, and the low latency requirements imposed by the application domain. Therefore, the locality requirements in those cases prohibit services from being migrated away from the edge node, even if this would be otherwise technically possible. Hence, workloads cannot be practically protected from failure by transferring them to other nodes in standalone mode.

Therefore, in the standalone mode, the most likely technique to be applied is to checkpoint the virtual machine during its execution. Checkpointing saves the current state of the VM to a file on the disk. After the node recovers from a failure, the VM can be restarted and its state restored by reading from this file, provided that the non-volatile storage / filesystem has survived the failure. Similarly to migration, both naïve, cold checkpointing and live checkpointing solutions are available.

## 2.3. Fault Tolerance for Management Components

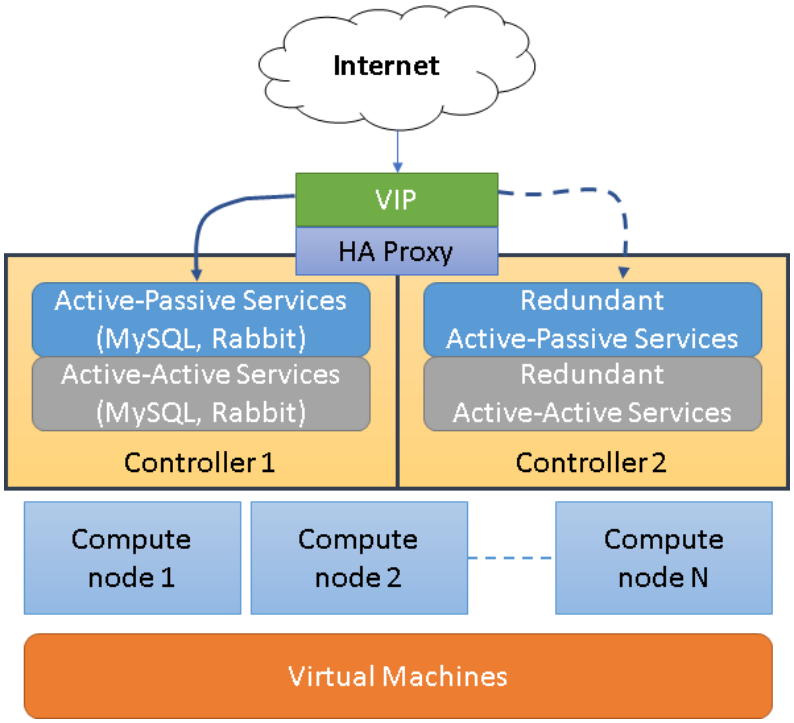


**Figure 2: OpenStack High Availability redundant deployment.**

Highly available systems aim to solve the problems of a) a system becoming unavailable, and b) the loss of data in the event of failure. In this section, we discuss techniques for ensuring high availability of the data center management components so that the user is able to submit and manage her workload executing on the nodes despite of failures of individual components. Techniques for avoiding loss of data through replication will be discussed in the next subsection.

A common approach for ensuring high availability is to ensure redundancy (Figure 2). For example, the presence of multiple instances of the compute controller will ensure that VMs will be scheduled even if one of the controllers crashes. OpenStack components are organized as Web services that expose a REST API for communication with the outside world. These services are mainly stateless, with the bulk of the state residing in the persistent data stored in the database and the message queue backend. Therefore, it is possible to replicate each service an arbitrary number of times, as long as all the instances of the service synchronize against the same persistent storage backends.

Redundant services can be organized in *active/passive* or *active/active* configurations. In both cases, a Virtual IP (VIP) is used to point to an active service endpoint. In the former, the VIP points to only one service. In case of failure the VIP endpoint is modified, so that requests are routed to the one of the passive copies, thereby activating it. In the latter, multiple instances of the same service are active, and the VIP points to a load balancer instance deployed to manage distribution of requests across them. There is no need to switch control in case of service failure.

In this context, it is important to consider the redundancy of the persistent storage backend. Currently, most commercial and open-source databases offer a mature clustered configuration, in which multiple instances of a database are synchronized using different mechanisms. Failure of one instance switches both the reading and writing to another replica of the database.

In the case of the standalone mode, redundancy of the control systems can be achieved by running multiple copies on the single machine. This technique guards from software faults, however it does not necessarily protect from hardware induced faults, as is the case in the context of UniServer. Therefore, it is debatable whether this improves the overall reliability of the system. It is more important to prevent catastrophic loss of data even in the case of the whole node going down.

## 2.4. Replication of Data

One of the critical requirements of managing data centers is to avoid catastrophic loss of data in the case of node failures. Loss of data occurs when there is no valid copy of a data item available on any storage media in the data center. A common technique to guard against this type of failure is to replicate data. OpenStack enables two main resources for data storage: volumes (Cinder) and an Object Store (Swift). In both cases there is the possibility to enable data replication and automatic failover.

OpenStack Cinder [5] is the OpenStack volumes manager. It enables the creation of replicated volumes to guarantee availability of data in case of disaster (e.g., disk failures, node failures, etc.). The user of OpenStack can create a replicated Cinder volume by specifying a number of backend devices that are used as replicas and kept in synch with the primary volume. Upon a failure OpenStack informs Cinder that a certain volume, associated to a VM, has failed and that one of the backend devices should now become the primary one. OpenStack Cinder supports multiple storage backends each of which should support replication and failover recovery in order to benefit from Cinder volumes replication. The configuration of replication on the backend side is vendor dependent and it is usually treated separately from Cinder configuration. An example storage backend with full support for replication is Ceph RBD that provides object replication capabilities by storing Block Storage volumes as Ceph RBD objects. Ceph RBD ensures that each replica of an object is stored on a different node, to protect data from both disk and node failures.

OpenStack Swift [6] is the default object store for OpenStack that similarly to Cinder provides seamless support for replication of data. Replication is handled by peer-to-peer replicator processes and happens at two levels: the database level, and the data objects level. The database keeps track of all object containers and objects metadata. Replicas of a database are compared using hashes. Any mismatch of the respective hashes triggers a synchronization of all replicas. Object data are replicated with PUSH semantics; each time an object is written locally, the same data is also written into the remote replicas. When a replicator process detects that a remote drive has failed, it chooses an alternate node to synchronize with. The replicator can also maintain desired levels of replication during disk failures.

In both cases, volumes and object storage, one requirement for replication to be possible is to replicate the data on multiple nodes to protect them from disk or full node failures. For some use-cases at least three nodes are required to implement robust replication that is resilient to multiple failures.

Thus, the possibility of replication in a standalone scenario is moot. Even if the standalone servers were connected in a network, latency and bandwidth constraints would make data replication costly, both in terms of time and in the actual resource usage. Moreover, access to a remote data store would probably not satisfy the data volume and latency constraints typically necessitating edge deployments.

## 2.5. Conclusions

Reliability is a major concern for both cloud infrastructure and standalone deployments. Faults may occur at any time on a compute node – even when it operates at nominal operating points – affecting its availability. This effect is more evident in scale-out deployments of compute infrastructure.

In cloud infrastructures the cloud management framework is responsible, among others, to monitor the health of all nodes and services, and to command the hypervisors on the underlying nodes for actions, in order to ensure stable operation for the VMs running on top of the infrastructure, within the quality parameters agreed with the end-users. On the other hand, in a standalone deployment the hypervisor of each node is more flexible, being responsible for all configuration and resource management decisions.

**Table 1: Applicability of protection mechanisms used in cloud deployments on a standalone system deployed at the edge.**

| Protection mechanism for cloud infrastructure | Practical / Applicable for Standalone Systems |
|---|---|
| Educated scheduling of workloads | **No** |
| Proactive health monitoring | **Yes** |
| Workload migration | **No** |
| Management components replication | **No** |
| Data replication | **No** |
| Checkpointing | **Yes (under conditions)** |

A set of protection mechanisms have been developed, both within and outside the context of UniServer, to facilitate stable operation of services provided on top of cloud infrastructure, even in the presence of errors. However, as summarized in Table 1, most of these protection mechanisms are not practical or even applicable at all for standalone systems. Given the limited arsenal of techniques for mitigating the effects of errors, system software on standalone micro-servers deployed at the edge needs to be more conservative when exploiting opportunities to operate at extended margins. Moreover, it needs to be engineered in a way that proactively minimizes the manifestation of errors. The following two sections discuss the respective techniques for managing the operation at extended margins of the two main hardware resources in the focus of UniServer, namely CPUs (Section 3) and DRAMs (Section 4).

# 3. CPU Management at Extended Margins in Standalone Deployments

In this deployment mode, the micro-server node is not coordinated by a cloud framework manager, and any policies and mechanisms for energy footprint minimization, protection and recovery against erratic hardware behaviour are orchestrated locally.

In order to minimize the footprint and complexity of Hypervisor code as much as possible, we have implemented the necessary policies for standalone mode in the context of a software module, which runs locally on the node, at the user-level, as a system daemon. The module utilizes, especially for system monitoring and collection of information, the mechanisms already implemented in Libvirt and discussed in detail in D5.3. The only difference is that the receiving endpoint of information is the daemon, rather than OpenStack. Configuration / protection commands from the system daemon are typically directed straight to the OS / Hypervisor, in order to minimize latency. Figure 3 outlines the architecture of the UniServer software stack in standalone deployments. Note that the cloud manager (OpenStack) is not available.

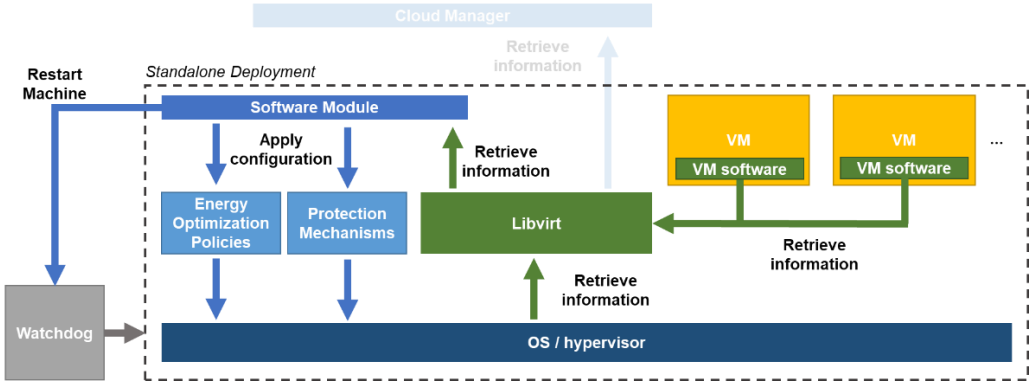


Figure 3: Standalone mode system software organization.

In standalone mode the Libvirt extended API also implements an additional virtual character device which is made available to each Virtual Machine (VM) hosted on the node. This character device is used for unidirectional communication between the VM and Libvirt. More specifically, VM administrators may install an optional additional software module to the VM, which acts as a forwarder of information concerning the health of services offered by the VM. The health checking methodology is service-specific, therefore it has to be implemented by the VM administrator / service provider. This functionality provides visibility of the health of the end-services to the UniServer system software stack and is particularly useful when operating at extended margins. An error introduced by erratic hardware behaviour may not produce observable effects to the health of VMs or the health of the Hypervisor, but rather to the health and functionality of services within the VMs. Such faults may occur in any hardware component i.e. in the cpus and in the memories.

## 3.1. CPU guardbands quantification and elimination

Aggressive CMOS technology scaling into lower nanometer geometries has led to variability of transistor characteristics resulting into increased failure rates in modern CPUs. Traditionally, techniques for dealing with transistor variability involve extra provisioning in logic and memory circuits in the form of increased voltage margins, reduced operating frequencies and error correction circuitry. Such guardbands are specified at design time by taking into account the implementation technology, power budget, the worst case timing paths, operating conditions and fabrication process variations. Guardbanding leads to significant power overheads, which is in conflict with one of the major challenges of semiconductor industry, namely limiting power dissipation. The average power cost of guardbands can be in the order of 35% yet most of the time these guardbands are excessive and translate to unnecessary overhead, as the worst-case combinations that were considered at design time may appear only rarely or even not at all during the life cycle of a given processor.

Prior work already presented in D3.3, identifies and quantifies the undervolting potential in different APM XGene 2 chips, using a diverse set of benchmarks which stress excessively the CPU microarchitecture. Figure 4 depicts an overview of the offline characterization process.

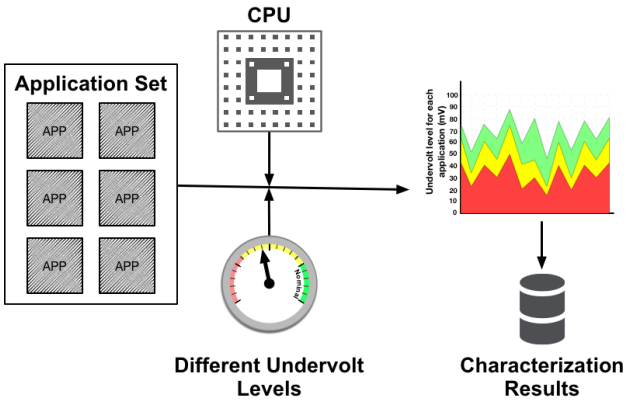


Figure 4: Characterization of the node CPU overview.

The characterization process identified substantial differences in different CPU parts (physical chips) and cores within each part. On top of that, different workloads that exercise different degrees of pressure to different resources on the CPU, are amenable to different undervolting levels. The observations of offline characterization can be exploited to drive both static and dynamic voltage management policies, as will be discussed in Sections 3.1.1 and 3.1.2.

### *3.1.1.* Static undervolting

In this mode our module, based on the offline characterization conducted on the CPU with representative workloads, always applies the lowest safe voltage (in extended margins). In other words, it applies the minimum undervolting level at which all applications that were examined, through the characterization phase, executed normally without any erratic behaviour (application crash, SDC or system crash). Figure 5 depicts an overview of this operation mode of the module.

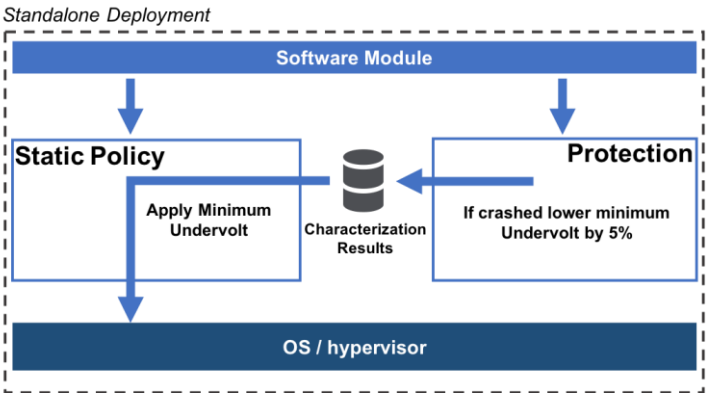


Figure 5: Static Mode overview.

Despite the fact, that this policy misses the potential energy savings that are presented at runtime, due to different workloads that exhibit different amenability to undervolting levels, it is the safest policy in terms of reliability. As the system operates under extended margins, resilience is inherently reduced and aggressive undervolting may lead to erratic behaviour.

Our module maintains a record in non-volatile storage of the applied undervolting level. Should the system exhibit erratic behaviour or even happens to crash, then this is detected (at the next boot in the case of a crash) and voltage margins are updated to more conservative levels. Moreover, the Hypervisor notifies the HealthLog (and implicitly the Predictor). This feedback and adjustment mechanism ensures that – even in the presence of extremely adverse conditions – the system will eventually – potentially after a series of restarts – identify a voltage level which will at least allow the workload to execute.

### *3.1.2.* Dynamic undervolting

In dynamic mode, our voltage management module leverages the amenability of workloads to different undervolting levels to maximize CPU power savings. In order to identify and associate dynamically changing workloads with safe undervolting levels, our module needs a way to make the respective estimations at run-time and apply them to the system.

#### 3.1.2.1 Modelling

Prior work, has indicated that models can correlate with adequate accuracy a safe undervolt level, in terms of nominal execution without any application crash, SDC or system crash, based on input from error detection and correction circuitry. In particular, heuristics presented in [7] and [8], dynamically reduce voltage margins while always preserving safe operation, based on the error correction ECC hardware built on modern processors such as the server-class Intel Itanium 9560. A key observation of those works is that as the operating voltage ($V_{dd}$) is lowered, ECC correctable errors appear before uncorrectable errors (SDCs and CPU crashes). The rate of ECC correctable errors is used as an indicator on how aggressively to readjust $V_{dd}$. Our approach targets different CPU architectures where, as characterization results indicate (deliverable D3.6), errors reported by the ECC mechanism appear very rarely. Even worse, undetected errors manifesting as SDCs and CPU crashes usually precede detected correctable errors. In our approach, we try to predict a safe supply voltage using a selected set of performance counters as estimators. Eventually, it is generic enough and can be applied to any processor that provides the ability to manipulate the supply voltage and to quantify resource pressure through performance counters.

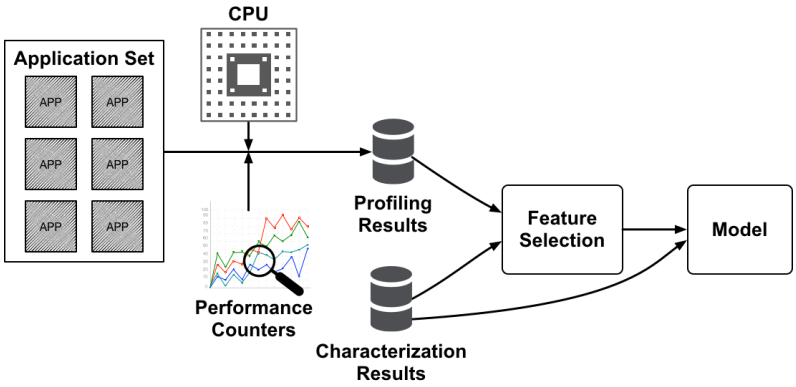


**Figure 6: Safe undervolting modelling using performance counters.**

In order to take into account application-specific behaviour, a prediction model has to be trained during the offline characterization phase. All applications that belong to the training set must be profiled using all available performance counters of the platform. At the next step, a subset of those counters is selected to serve as input to the model (feature selection). During this step we take into account the number and combinations of metrics that can be concurrently collected using performance counters on the target architecture. Then, the model is trained to infer safe undervolting levels given the respective features. Figure 6 summarizes this process. The details are, however, outside the scope of this deliverable and will be discussed, together with the respective experimental evaluation, in deliverable D5.5.

#### 3.1.2.2 Voltage control governor

The resulting model is employed by a voltage control governor, in the context of the user-level daemon. The governor is invoked periodically, observes the dynamic state of the system (and more specifically hardware / software interaction) by sampling the performance counters and sets the appropriate undervolting level. There is a tradeoff concerning governor invocation frequency. On the one hand, the more frequently the governor is invoked, the closer voltage can follow the dynamic changes of hardware pressure exercised by the workload. Beyond maximizing power efficiency, this also has the potential to result to higher reliability (especially in case the voltage needs to be rapidly adjusted towards nominal values). On the other hand, a high frequency of invocation results to higher overhead and higher interference between the governor and the workload. Moreover, there are inherent limitations when sampling performance counters at a system scope; the highest

sampling frequency that results to accurate values is 100 Hz. We have empirically observed that a governor invocation frequency of 10Hz is a good sweetspot, which limits interference, does not have adverse effects on reliability and limits the overhead of the governor to a mere 0.04%.

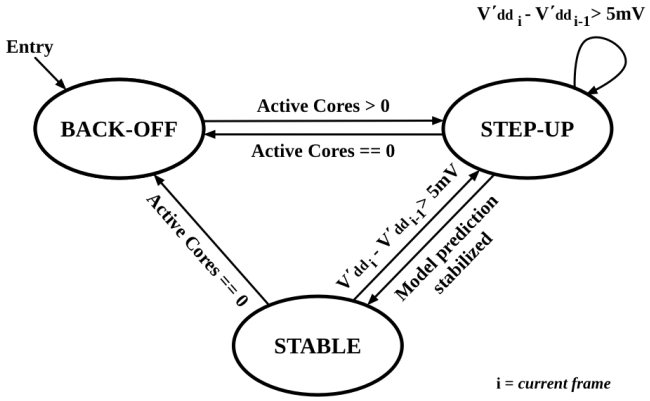


**Figure 7: Governor state-graph.**

The governor is implemented as a Finite State Machine (FSM), depicted in Figure 7. Among other metrics, it also monitors which cores are active, by observing the percentage of time the system was utilizing of each core. We consider a core as active when utilization goes above 70%. On the contrary, a core is classified as inactive when it drops below 50%. We avoid thresholds close to 100% or 0% as this would result to unnecessarily high transition sensitivity (e.g., a core would be considered as active when merely moving the mouse of a desktop).

The undervolting level selected for the next interval depends on the predictions of the model, the number of active cores, as well as on the current state of the governor. Below we describe the states and the logic of the governor:

- **Back-Off:** In this state, the measurements collected during the previous interval are not considered as representative for the workload during the next interval and, therefore, are not valid input for the model. This is the case, for example, when a core starts executing a new process/thread, or after a long idle period. In this state, the governor does not invoke the model and does not apply any undervolting. Instead, it applies nominal settings.
- **Step-Up:** This state provides a conservative, smooth transition between the *Back-Off* and the *Stable* states. When in this state, the governor invokes the model to perform predictions in a way similar to the *Stable* state. If, however, the model suggests a large reduction to the supply voltage, this is applied gradually, in smaller steps of 5mV. This state filters abrupt and potentially risky – in terms of reliability, should the behavior of the workload change again – voltage reductions. Voltage increase predictions, on the other hand, are treated as emergencies and are applied immediately in order to not compromise the reliability of the system.
- **Stable:** The governor collects the appropriate performance counter values for each core and invokes the model. Then, based on the model prediction, the governor applies the new undervolting level. Once again, if the model commands an abrupt decrease of voltage the governor transitions to the step-up state.

#### 3.1.2.3 Voltage safety margin

Beyond the aforementioned logic, the governor applies an additional safeguard against unreliable system operation. Over- or under-prediction is a common side-effect of many modeling approaches. In our case, over-predicting the safe undervolting level would result to unreliable operation. Therefore, as a last step, we introduce a small *Safety Margin* to the estimated permittable undervolting. The safety margin is set equal to the root mean square error (RMSE) between the value that is predicted for a set of validation benchmarks, and the safe undervolting level determined during the offline characterization for the respective applications in the validation set.

The safety margin controls the aggressiveness of our methodology. Using very small values would result to aggressive undervolting at the risk of reduced reliability, whereas too large safety margins would merely

decrease the energy gains. Beyond using RMSE, we could assign as safety margin the maximum error between the predicted values and validation data, but this would lead to an overly pessimistic model. In any case, the methodology for producing the "base" prediction model would remain the same. Including the safety margin reduces power efficiency, but enables safe operation.

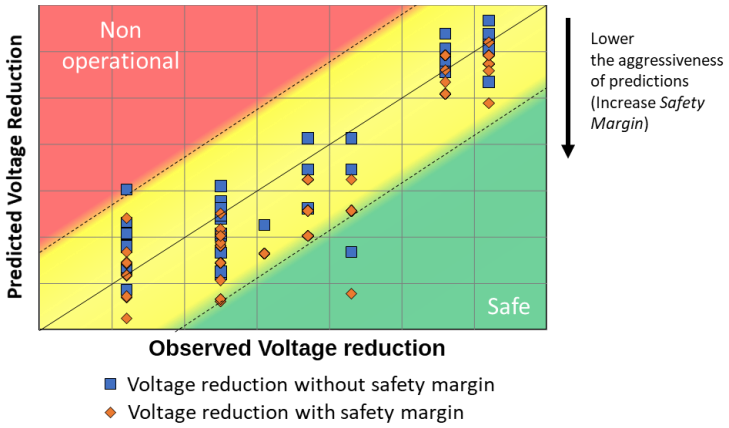


**Figure 8: Graphical overview of Safety Margin mechanism.**

Figure 8 encapsulates the behavior of the *Safety Margin* mechanism. The orange diamonds are the undervolt predictions with the *Safety Margins* mechanism implemented. Beyond proactively determining the safety margin as described earlier, the governor reactively increases the Safety Margin (by 5%) after any unexpected system crash (or other observed erratic behavior). The experimental evaluation of the effects of the voltage governor and the conservative capping of undervolting levels will be discussed in detail in D5.5. Initial results indicate that on Intel Xeon Skylake servers it is possible to achieve an average energy gain of 30% over the standard dynamic voltage and frequency scaling (DVFS) governor. The average voltage applied by the conservative voltage control governor – including the safety margin – is, on average, a mere 9.3mV higher than the minimum safe voltage (Vmin) identified for each application through offline characterization. With those conservative safety measures in place it was possible to run dynamically varying real world workloads (mining, linux kernel compilation, microarchitecture simulation etc.) continuously for 3 days without observing any erratic behavior.

While the dynamic mode exploits the correlation between safe undervolting levels and the resource pressure exercised on hardware, the implementation is still not identical with the respective mode when the targeted node operates under a cloud framework manager. In standalone deployments, workloads are introduced to the system without any warning, and the governor needs to react promptly to compensate to changes to resource pressure due to varying workloads. On the other hand, when operating under the control of a cloud manager, the arrival of a new workload is known to the cloud manager before the actual execution takes place. Therefore, the governor can have an advance warning and / or even opt to operate the system at nominal levels.

## 3.2. OS/Hypervisor functionality migration to reliable cores

Another proactive strategy to guard against errors that compromise the stability and availability of the whole standalone node is the execution of critical system software code on reliably configured cores only. As discussed in previous deliverables XGene 2 allows the configuration of the CPU to nominal / extended margins at the granularity of PMDs (pairs of cores). Therefore, one (or more) pairs of cores are maintained at nominal settings and system code is directed to them.

Deliverable D5.2 describes in detail the methodology for migrating system calls, page fault handler, interrupts, scheduler, etc. to reliable cores. The migration of OS/Hypervisor functionality to reliable cores requires the non-trivial undertaking of breaking the SMP[1] assumptions of Linux and KVM. Figure 9 illustrates the high-level

[1] Each core executes the OS/Hypervisor code on its own to serve system calls and page faults initiated by processes running on that core, to schedule processes on that core etc.

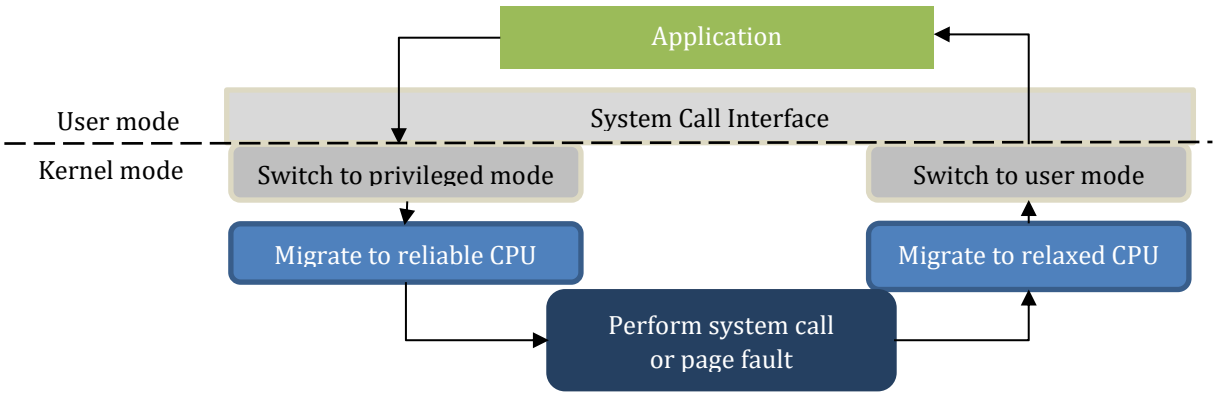


Figure 9. System calls migration (D5.2)

handling of system calls migration. Migration of system-code functionality to reliable cores significantly reduces the surface of critical code susceptible to errors due to being executed on potentially unreliable hardware.

## 3.3. Design and implementation of an external micro-server watchdog

A micro-server may inadvertently crash even when operating at nominal configuration. Operating at extended margins inevitably increases the risk of a node becoming unresponsive. A hardware watchdog is, therefore, required to detect such cases and restart the node. In standalone deployments, where the micro-server often is not readily accessible, such a mechanism is even more valuable. Many commercial servers include a baseboard management controller (BMC) which implements a watchdog. BMCs also typically implement virtual remote console functionality, as well as introspection capabilities to server hardware (through IPMI interfaces).

Unfortunatelly, XGene 2 boards do not come with a BMC add-on. Therefore, we had to design and implement this functionality. Figure 10 outlines the architecture of our approach. Our solution is based on a Raspberry Pi (RPi), combined with simple circuitry to restart the node. We selected RPi as it is a low cost platform, with low power footprint and rich connectivity options.

The circuitry consists of two optocouplers and one BJT transistor. The transistor acts as a switch that enables the optocouplers when the general purpose I/O (GPIO) pin of the RPi that is connected to the transistor base is high. When the optocouplers are enabled they short-circuit the reset pins of the motherboard (pins RST1 and RST2) with a 200 Ω resistor.

We opted to employ two optocouplers to achieve galvanic isolation between the RPi and the micro-server. This way, our circuit does not use a common ground and any anomalies in the power lines of either system do not affect the other system. Moreover, the circuit operates correctly, independently of the polarity of the reset pins of the motherboard.

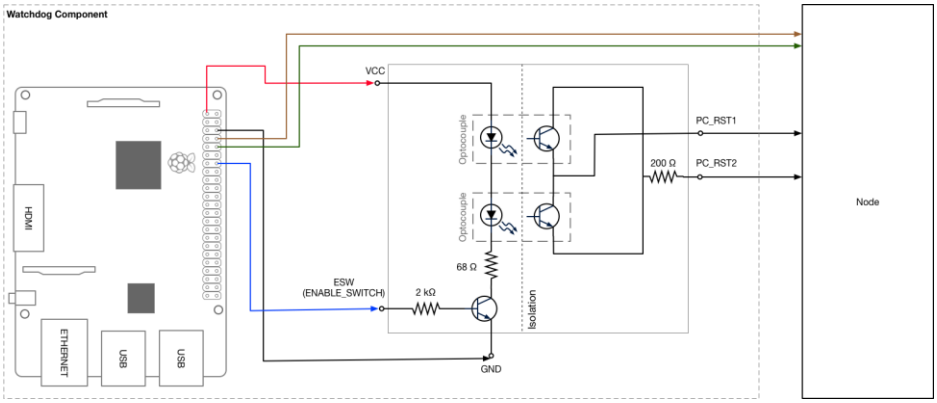


Figure 10: External component for micro-server control.

The RPi expects a periodic heartbeat which is generated by the user-level daemon and transmitted through the serial interface of XGene 2. Should the heartbeat not be received, the RPi signals the daemon to gracefully reboot the system and if this is not possible, it initiates a reset.

Beyond acting as a watchdog, the RPi is connected – through another opto-isolated serial interface – with the serial console of the micro-server. Therefore, the console is remotely accessible through the RPi Ethernet interface. Finally, beyond crashes, the RPi can collect – and forward – health information about the hypervisor, the VMs, and the services within the VMs through the software stack depicted in Figure 3.

# 4. Memory Management at Extended Margins in Standalone Deployments

As we discussed in prior deliverables within WP3, apart from the operation of CPUs under scaled supply voltages, we also explore operation of the DRAMs within each server under relaxed refresh rates and voltages. However, as we discussed in deliverable D3.3 any relaxed operating condition leads to a substantial increase of the manifested errors posing a threat to the non-disruptive system operation since it increases the probability of uncorrectable errors that may affect the OS. To this end, it is necessary to develop a mechanism that can provide elevated memory protection to critical OS and application software data, while allowing less critical application data (e.g. heap) to be stored within memory domain(s) whose energy-efficiency and reliability could be adjusted.

This is very important in the standalone deployment, as it is difficult to apply other protection mechanisms, like data replication. The basic idea of the proposed scheme lies on dividing the available address space into two or more separate memory domains, whose reliability could be controlled independently by adjusting the supply voltage and refresh rate depending on the criticality of the stored data. Having such a scheme in place we could effectively develop intelligent data allocation policies ensuring that critical OS and application software data are stored in a highly reliable memory domain with a reliable but power hungry setting (i.e. operating at nominal refresh rate and voltage) while the rest of the data will be stored in the other domain(s) whose operation could be relaxed to a less a reliable but lower power settings.

In previous year, as we discussed in the deliverable D3.3, we have implemented such a scheme on a dual-socket Intel server, in order to enable the characterization of DRAMs under extended range of refresh rates. In that case we had to implement such a scheme to separate the OS data since otherwise would be impossible to perform the DRAM characterization due to the frequent system crashes by errors affecting the OS. In our initial implementation we achieved to split the memory into 2 domains due to the availability of 2 different memory controllers and the support of Non-uniform memory access (NUMA) on the OS ported on the Intel server.

In the latest DRAM characterization deliverable D3.6 after the necessary changes on the UniServer firmware and software stack/OS we have implemented such a heterogeneous scheme on the ARMv8 based Tigershark (XGene 2) server. The implemented setup on the XGene2 was based on 4 Memory Controller Units (MCUs), spread in 2 Memory Controller Bridges (MCBs) and a dedicated processor, the Scalable Lightweight Intelligent Management processor (SLIMpro) used for controlling the memory operating parameters. However, the implemented scheme was different than the one on Intel since the Linux OS kernel on the XGene2 platform did not support NUMA. Therefore, the interface to system software and applications were not straightforward and compatible with the previous framework and required to be refined for minimizing the overheads and its efficiency.

However, such inefficiencies needed to be overcome for offering a better management of applications and data allocation to the UniServer stack for enabling more efficient data allocation policies. To this end, in this deliverable we discuss the practical limitations and the extension of our heterogeneous memory framework on the latest UniServer stack for overcoming any issues. For the first time we also evaluate the performance overheads incurred by the heterogeneous framework as well as the potential energy savings using various well known benchmarks in order to understand the costs of supporting such a scheme for increased system reliability/system even under relaxed operating points.

In the following subsections we discuss the extension of our heterogeneous memory framework on the latest UniServer stack and its evaluation.

## 4.1. Implementation

In our refined implementation of the framework on the latest software stack on the XGene 2, we use the notion and interface of NUMA domains in the Linux OS to treat different MCUs independently. NUMA domains are conventionally used on NUMA hardware, namely systems in which different processors observe different access latencies to parts of the physical address space. In our case, we employ the functionality of NUMA domains even though the access time on the four MCUs is the same for each of the 8 cores of the XGene 2.

The latest Linux kernel (used on XGene 2) already exposes an interface to create "fake" domains and assign cores and parts of the memory in each one of those. These interfaces had not been ported yet to the ARMv8 architecture, therefore we ported these capabilities and modified them to suit the purposes of the framework.

### *4.1.1.* Hardware Interleaving

One of the faced practical challenges lies on the memory interleaving mechanism that is used conventionally for improving the performance and avoiding frequent stalls due to a congested memory accesses. In fact, interleaving spreads memory accesses across different MCUs allowing to parallelize the use of the available MCUs. With such a performance oriented scheme active the realization of the heterogeneous memory scheme is impossible since irrespective of the data allocation made by the OS/hypervisor the memory controller would interleave the memory access spreading them around in the DIMMs.

Therefore, the first step in realizing our framework was to deactivate the hardware interleaving. This can be achieved at the initialization phase of the MCUs, during which we can define the level of interleaving, either between the four MCUs, or inside each MCB, or disable it totally.

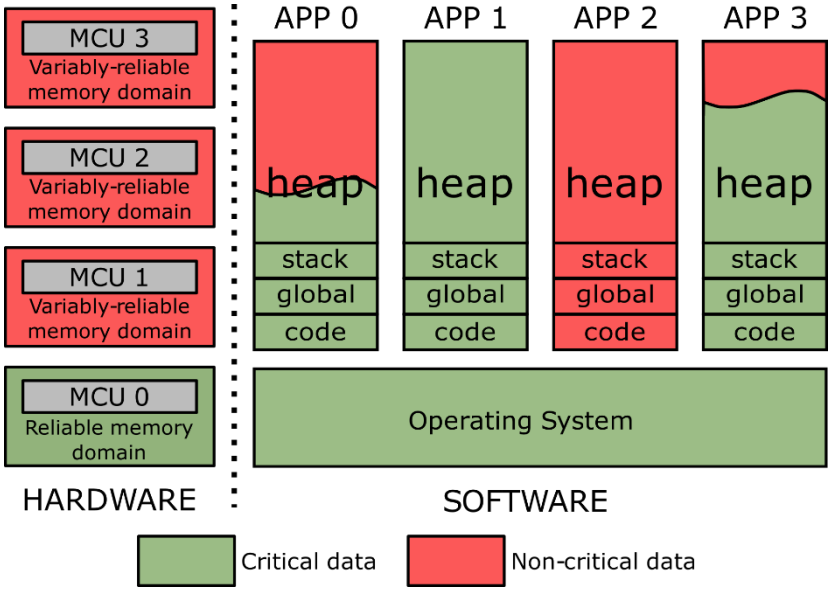


**Figure 11: Hardware setup for the variably-reliable memory and possible allocations of applications in the memory.**

At kernel boot time, we disable the hardware-based interleaving to achieve a separate address space for each MCU and we extract the address space of each MCU from the Advanced Configuration and Power Interface (ACPI) table, which corresponds to each DIMM. We use our interface to introduce 4 fake NUMA domains, each one corresponding to one MCU. Even though all NUMA memory domains have the same access time from each core, we assign the cores on the first NUMA domain that we deem as the Reliable Domain with nominal settings for the memory. The rest of the NUMA domains are configured as Variably-reliable Memory Domains, in which the memory parameters can be relaxed. By modifying the allocation scheme policy, we are forcing the Linux kernel and every application using the default allocation functions of the OS to allocate data with priority on reliable memory, as long as there is enough memory in this domain.

Applications can control the memory allocations for the whole application through the *numactl* command of the NUMA interface, which specifies the memory domain that the application will use. In Figure 11, APP 1 and APP 2 are examples of such an allocation; the whole application is assigned to either the Reliable Domain or the Variably-reliable Domain. Parameters can be passed to numactl, such as *--membind*, which define the NUMA memory domain that is going to be used.

### *4.1.2.* Software Interleaving

As we will show later in our results our experiments indicated that deactivating the hardware-based interleaving incurs high performance overheads in most cases. The result is expected as the interleaving allows consecutive accesses to be load balance across the MCUs. In case that a benchmark allocated memory less than the capacity of one MCU, the allocation will reside in one MCU and the bandwidth will be limited to the one of one MCU.

Therefore, we had to utilize a mechanism to limit such overheads and close the performance gap compared to a system with hardware-based interleaving

To this end, we have enabled the use of software interleaving between NUMA memory domains. This was achieved by utilizing the parameter *--interleave* of the numactl policy of Linux. By doing so, applications can exploit the memory bandwidth of multiple MCUs as in the case of the default hardware-based interleaved configuration. In our setup, we can extend the software-based interleaving among the 4 MCUs, however we limit the interleaving to the 3 variably-reliable domains, so that the reliable domain is not bloated with non-critical data.

### *4.1.3.* Programmer's Interface

In order to facilitate the adaption of our techniques for heterogeneous memory in commercials systems, we introduce an interface that enables the programmer to customize their applications for increased reliability for the critical data, to give the ability to allocate non-critical data on variably reliable to save energy and power, and ensure the proper operation of the system when other protective mechanism are sparse in the standalone deployment.

Programmer can also control the memory allocations at the granularity of individual allocations, if source code is available and can be modified. Typical dynamic allocation primitives, such as *malloc*, can be converted to the respective numa-aware ones, such as *numa_alloc_onnode* in which programmer can specify the NUMA domain that she opts to allocate the data on. With the knowledge of the criticality of different data-structures, programmer can place them in the appropriate memory domain.

We are forcing the Linux kernel and every application using the default allocation functions of the OS to allocate data with priority on the first MCU, as long as it is not full. The first MCU is set up with the nominal parameters for supply voltage and refresh rate.

The applications can explicitly allocate memory from the second domain MCB. We have developed an extension of malloc that uses the same scheme of allocation as the default Linux policy, however it prioritizes allocations of memory objects on the second MCB domain. The data structures of the OS have been extended to support allocations from the two domains.

For cases direct control on physical memory allocation is required, we have developed a routine that uses *mmap* interfaces and */dev/mem* to allocate memory on a specific DIMM explicitly and disable that memory for normal allocations. This explicit, low-level allocation mechanism mitigates the uncertainty of memory location caused by *malloc*. Therefore, we ensure that the same physical memory will be allocated for each experiment, thus yielding reproducible results.

## 4.2. Experimental Evaluation

### *4.2.1.* Workloads

For evaluating the heterogeneous memory framework, we chose a set of benchmarks, ranging from micro-benchmarks that are used typically in DRAM characterization, to real-world applications. Each of them stresses the cores and DRAMs in a different way and allows to study the influence of our modifications in different workloads. In particular we use:

1. Micro-benchmarks which identify deficiencies in utilization of memory bandwidth. STREAM is an example of a micro-benchmark used to quantify the bandwidth of DRAM.

2. SPEC CPU2006, a classical benchmark suite. We run 29 of the benchmarks with the "ref" data inputs, as a single instance in a core and as the "rate" version of the benchmarks by running an instance of the application in each core. This way we can evaluate the system on typical conditions of the SPEC benchmarks and also with increased pressure on memory and CPU.
3. NAS Parallel Benchmarks (NPB), a small set of programs designed to help evaluate the performance of parallel supercomputers. The benchmarks are derived from computational fluid dynamics applications and consist of 8 kernels. The version of NAS we use is parallelized with OpenMP and utilizes all cores of the system.

To evaluate the efficiency of the developed mechanism and quantify any performance overhead we used the metric that we evaluate is MPKI (Misses Per Kilo-Instructions). It reflects cache performance and the number of accesses that end up in the main memory.

### *4.2.2.* Performance Evaluation

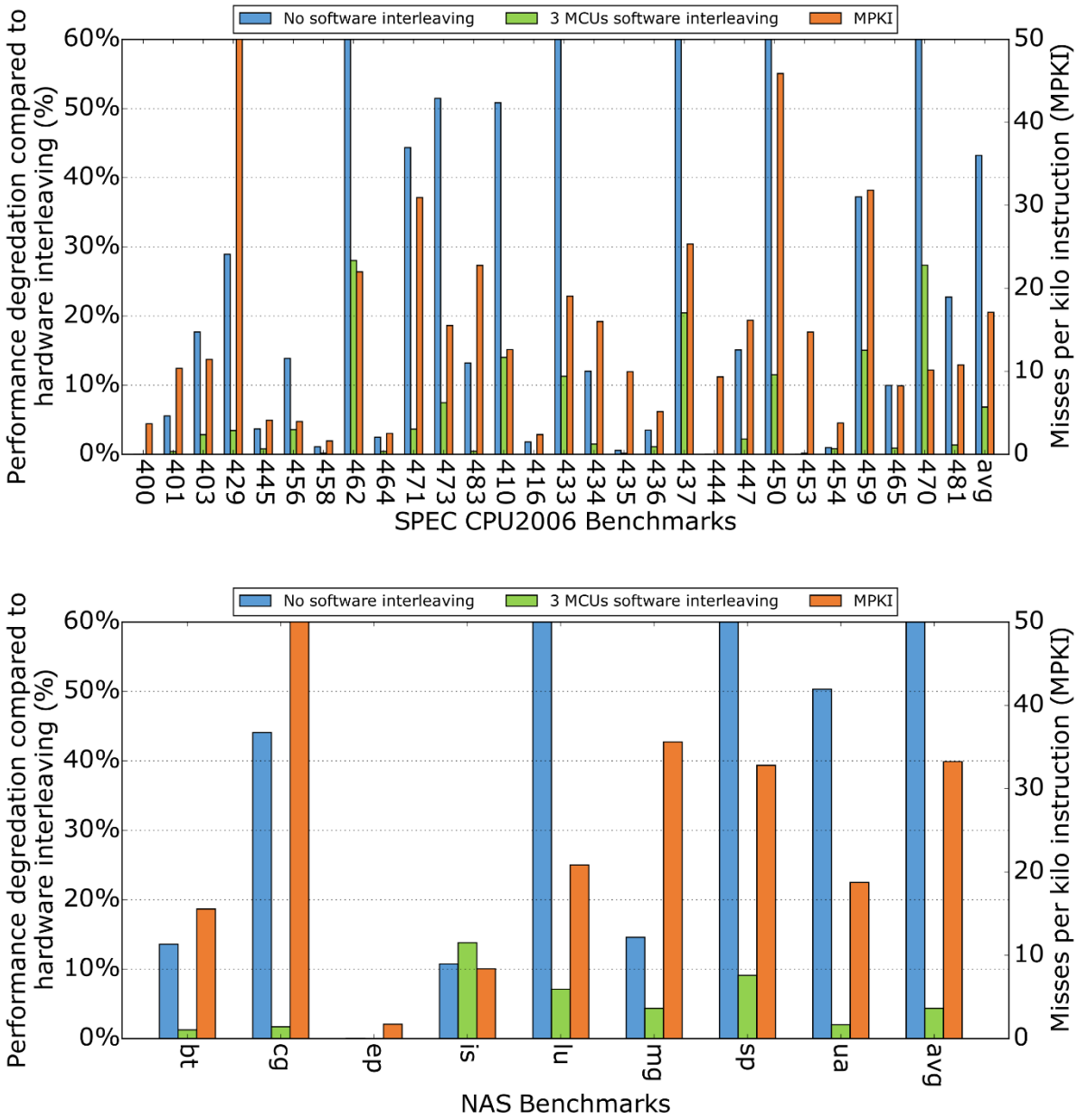


**Figure 12: Performance degradation compared to the nominal system employing hardware interleaving for: a) SPEC CPU2006 and b) NAS benchmarks.**

As we said above, by disabling interleaving of the memory, we induce a degradation of the exploitable bandwidth by applications, as accesses are not spread across the MCUs. We measure the memory bandwidth with the STREAM micro-benchmark with less than 8 GB allocation so that in the non-interleaved system the allocation is restricted in only one MCU. The bandwidth of the default system, that spreads the accesses across all 4 MCUs, is 22.1 GB/s which corresponds to 5.52 GB/s per MCU. When the hardware-based interleaving is disabled and data are allocated in one MCU, we saturate the bandwidth of one MCU at a transfer rate of 7.63 GB/s, which is higher than the default system's bandwidth of one MCU in the hardware-based interleaved. However, in reality when we use STREAM with more memory which extends to multiple memory domains the bandwidth does not increase more than 10.9 GB/s.

Figure 12 depicts performance degradation of benchmarks from the SPEC CPU2006 and NAS benchmark suites, compared to the hardware-interleaved memory system. For the isolated cases in which a slight performance increase (less than 1%) is observed, the culprit is system noise and OS jitter. On average, the non-interleaved implementation of heterogeneous memory framework decreases performance by 49.39%. This is expected, as memory bandwidth is not utilized properly.

Interestingly, we observe that the implementation based on software interleaving reduces the overhead to a mere 6% compared to hardware-based interleaving, even though each application is limited to 3 MCUs, compared to 4 in the nominal operating state of the hardware-based interleaved system.

The impact on the performance seems to be very workload dependent. In fact, we can observe that 9 out of the 36 applications exhibit under 5% degradation when they are executed using the non-interleaved scheme. On the other hand, *462.libquantum* and *470.lbm* exhibit a 2.11x and 2.28x slowdown respectively for the non-interleaved implementation and 28% and 27.3% for the software-based interleaved.

The worst-case scenario of performance degradation occurs in both the non-interleaved implementation and the software-based interleaved for the same benchmarks, namely *462.libquantum* and *470.lbm*. They suffer a 2.11x and 2.28x slowdown respectively for the non-interleaved implementation and 28% and 27.3% for the software-based interleaved. It is interesting to note that 25 out of 36 applications executed in the software-interleaved system inflict less than 5% overhead.

To further investigate the workload dependence of the performance degradation, we investigated how memory access pattern and intensity are affecting the performance degradation for each implementation, the non-interleaved and the software-based interleaved. We measure the MPKI to reflect how memory intensive an application is, as it is not affected by the execution time of the benchmark. Furthermore, MPKI remains similar across 3 setups (hardware interleaving, no interleaving, software interleaving), which is expected as it is affected by the size and the associativity of caches and the order of accesses of the benchmark, which both remain the same.

To further quantify the dependency between the afflicted performance overhead and MPKI, we use the Spearman's rank correlation coefficient [9], $r_s$, which reflects the monotonic relationship between those variables. The correlation coefficient is estimated as:

$$r_s = 1 - \frac{6 \times \sum d_i^2}{N \times (N^2 - 1)},$$

where $d_i$ is the difference between two ranks of each observation N is the number of samples.

Our results indicated that for the naive implementation the correlation is high with $r_s = 0.7$. In case of our framework the correlation coefficient is lower with $cr_s = 0.59$. This indicates that after the optimization, the incurred overhead is less correlated with the memory intensiveness of the application. This can be attributed to the fact that we have significantly less overhead, and part of this overhead is associated now with implementing software-based interleaving.

### *4.2.3.* Power and energy evaluation

To quantify the maximum expected power gains, we configured our framework with 3 MCUs at the least reliable settings. We can lower the supply voltage only on the DRAM DIMMs associated with one of the MCBs (MCB 1, so MCU 2 and 3) and relax the refresh rate on three of the MCUs (MCU 1, 2 and 3).

We configure SLIMpro to set the supply voltage to the minimum specified by the DDR datasheet, namely 1.425 V or 95% of the nominal, and increase refresh rate from the nominal 64 ms to 2.283 sec, which is the maximum allowed refresh rate on the XGene 2 server. Figure 13 (top) shows the average DRAM power consumption of the non-interleaved and the software-based interleaved system with 3 MCUs, compared to the hardware-based interleaved system. With this configuration of our framework, we can lower the average DRAM power consumption of the DIMMs from 9.3 W to 7.5 W, yielding and average gain of 19.9%. The highest DRAM power consumption, which is a metric of interest when considering a power capped system, is observed when executing the *470.lbm* benchmark. Operating DRAMs at extended margins reduces the maximum power consumption from 18.4 W to 13.3 W, or by 27.6%.

Taking into account the execution time overhead introduced by memory segmentation, we can calculate the difference of the energy footprint of applications executed with the nominal configuration vs the one exploiting extended margins. We also measure the power consumption of the CPU during each execution to take this into account as well. For those experiments CPU operates at nominal settings, in order to isolate power / energy effects of the memory system. We calculate the energy of the system based on the integral of the power over the duration of the benchmark.

Figure 13 (bottom) presents the results of the calculated energy footprint of applications on our system, normalized to that of the hardware-based interleaved system. The naive implementation introduces an energy overhead of 19.9%, while the utilization of the software interleaving implemented by our framework achieves a reduction in energy consumption of 8.8%.

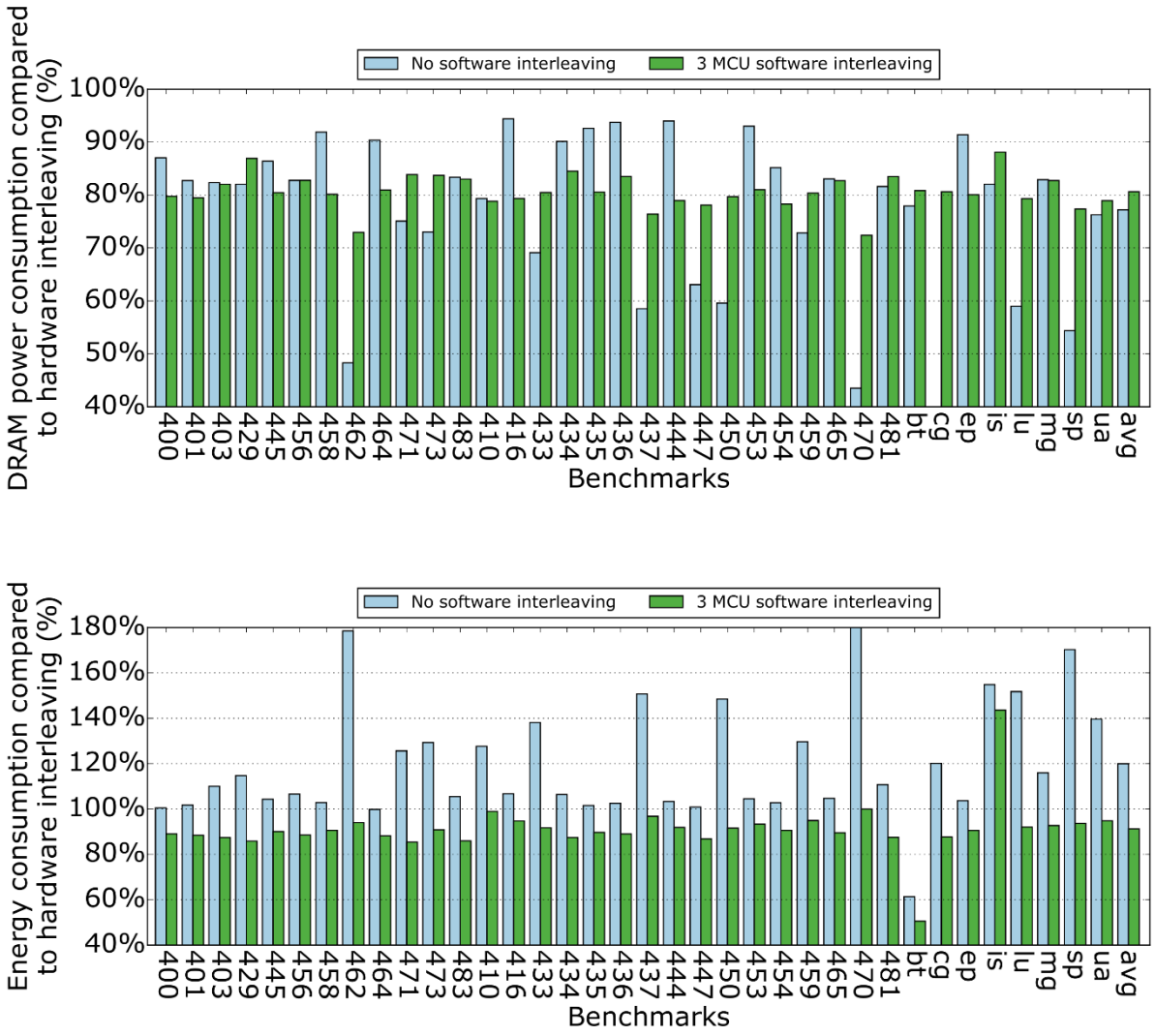


**Figure 13: Power and energy comparison between our framework and the hardware-interleaved system.**

Finally, by slicing the memory into different domains, we can employ additional power saving techniques when one of the memory controllers is not in use. We can progressively set memory controllers to deeper and deeper sleeping states, or even turn it off if there are no resident data, thereby drastically reducing the maximum power consumption of the system.

## 4.3. Alternative methods for increased memory reliability

Note that apart from the above scheme, the findings of the characterization and the proactive and reactive mechanisms discussed in D5.4 could also utilized for increasing the memory reliability and system availability to complement the heterogeneous scheme even under relaxed refresh rate and/or supply voltage.

For instance, the Linux governor presented in Section 3 can also monitor the DIMM temperature of the variably reliable domains and in case that it exceeds 70C, which was found to be a critical threshold beyond which the system crashes with extremely high probability then it could increase the refresh rate or voltage at a higher point in order to limit the errors.

Similarly, the checkpointing and restart mechanism described in the deliverable D5.4 could be used to further enhance the system dependability, in case that the heterogeneous scheme and any reactive mechanism would fail to maintain non-disruptive system operation.

# 5. Conclusions

In this document we introduced system software support for standalone micro-server deployments, to facilitate the exploitation of extended margins for the operation of CPUs and DRAMs, while at the same time minimizing the risk to compromise the stability of the system.

We observe that standalone systems enjoy flexibility in managing themselves, in contrast with cloud deployments, where the operating point of the node is often commanded by an external cloud management framework. At the same time, however, standalone micro-servers cannot benefit from protection mechanisms typically applicable on cloud infrastructure. Therefore, we find that system software needs to be more cautious and conservative when exploiting opportunities to operate hardware at extended margins. Moreover, it needs to provide mechanisms that allow software to selectively execute code and store data on nominally configured (and thus nominally reliable) CPU cores and DRAM regions respectively. Finally, the system (both at the hardware- and software-level) needs to be engineered in a way that allows, should all other safeguards fail, autonomic recovery from erratic operation.

In Sections 3 and 4 we discussed the design and implementation of the respective mechanisms and policies for the management of the operation of CPUs and DRAMs at extended margins in standalone systems. The proposed mechanisms and policies are characterized by an interesting tradeoff. Power gains by operating at extended margins are limited by the conservativeness necessary for operating in standalone mode without excessively risking quality of service. The respective energy gains are also limited by the overhead introduced by some of the proactive protection mechanisms. However, as the experimental evaluation presented for the heterogeneous reliability memory system proved, those overheads are limited by careful implementations of the respective mechanisms and cannot outweigh the power and energy gains unveiled by operating CPUs and DRAMs at extended margins.

# References


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

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

[3] Openstack, "OpenStack Docs: Migrate instances," [Online]. Available: https://docs.openstack.org/ocata/admin-guide/compute-live-migration-usage.html.

[4] W. Voorsluys, J. Broberg, S. Venugopal and R. Buyya, "Cost of virtual machine live migration in clouds: A performance evaluation," in *IEEE International Conference on Cloud Computing*, Heidelberg, Springer, 2009, pp. 254--265.

[5] Openstack, "OpenStack Docs: Replication," [Online]. Available: https://docs.openstack.org/cinder/pike/contributor/replication.html.

[6] Openstack, "OpenStack Docs: Replication," [Online]. Available: https://docs.openstack.org/swift/latest/admin/objectstorage-replication.html.

[7] A. Bacha and R. Teodorescu, "Using ECC Feedback to Guide Voltage Speculation in Low-Voltage Processors," in *In Proceedings of 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[8] A. Bacha and R. Teodorescu, "Dynamic Reduction of Voltage Margins by Leveraging On-chip ECC in Itanium II Processors," *SIGARCH Comput. Archit. News,* vol. 41, pp. 297-307, 6 2013.

[9] D. G. Bonett and T. A. Wright, "Sample size requirements for estimating pearson, kendall and spearman correlations," *Psychometrika,* vol. 65, no. 23, 2000.