Received June 3, 2021, accepted July 9, 2021, date of publication July 20, 2021, date of current version August 2, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3098644

Mitigating Virtualization Failures Through Migration to a Co-Located Hypervisor

FREDERICO CERVEIRA, (Member, IEEE), RAUL BARBOSA, AND HENRIQUE MADEIRA
CISUC, Department of Informatics Engineering, University of Coimbra, 3030-290 Coimbra, Portugal

Corresponding author: Frederico Cerveira (fmduarte@dei.uc.pt)

This work was supported in part by the Foundation for Science and Technology (FCT), I.P., under Ph.D. Grant SFRH/BD/130601/2017, in part by the scope of project under Grant CISUC-UID/CEC/00326/2020, in part by the European Social Fund, through the Regional Operational Program Centro 2020, in part by the Autonomic Service Operation (AESOP) Project under Grant P2020-31/SI/2017 and Grant 040004, in part by the FCT within project under Grant ECSEL/0018/2019, in part by the Electronic Components and Systems for European Leadership (ECSEL) Joint Undertaking (JU) under Grant 876852, and in part by the JU from the European Union's Horizon 2020 Research and Innovation Programme and Austria, Czech Republic, Germany, Ireland, Italy, Portugal, Spain, Sweden, and Turkey.

ABSTRACT Many organizations are moving their systems to the cloud, where providers consolidate multiple clients using virtualization, which creates challenges to business-critical applications. Research has shown that hypervisors fail, often causing common-mode failures that may abruptly disrupt dozens of virtual machines simultaneously. We hypothesize and empirically show that a significant percentage of virtual machines affected by a hypervisor failure are capable of continuing execution on a new hypervisor. Supported by this observation, we design a technique for recovering from hypervisor failures through efficient virtual machine migration to a co-located hypervisor, which allows virtual machines to continue executing with minimal downtime and which can be transparently applied to existing applications. We evaluate a proof-of-concept implementation using fault injection of hardware and software faults and show that it can recover, on average, 41-46% of all virtual machines, as well as having a mean virtual machine downtime of 3 seconds.

INDEX TERMS Cloud computing, dependability, fault injection, fault tolerance, virtualization.

I. INTRODUCTION

Cloud computing infrastructures provide elastic resources to organizations, enabling them to deploy scalable online applications and services while reducing the fixed costs of IT infrastructures [1]. Many organizations are already developing new applications by following a cloud-first strategy and most others are considering the move over the coming years. For organizations to shift to cloud computing, there is a need to assure that business-critical applications fulfill service-level objectives [2] such as availability and reliability [3].

Virtualization is one of the enabling technologies supporting cloud computing initiatives. Cloud providers rent their physical infrastructure to multiple tenants, using virtualization to execute up to hundreds of virtual machines (VMs) on a single, powerful physical machine [4]. Although this is a very cost-effective approach, it creates the risk of *common-mode failures* [5], which have been observed in previous research [6], [7] and create a significant threat to virtualized infrastructures – numerous VMs, potentially from different cloud tenants, lack failure independence and may simultaneously fail due to a single fault affecting a physical machine or the hypervisor.

This work aims to improve the availability of cloud computing deployments by maintaining service continuity of the VMs despite hypervisor failures. We hypothesize that at least some VMs may be left in a correct state after a hypervisor failure. This hypothesis derives from other works that have shown state corruption to be a relatively uncommon occurrence (*e.g.*, about 2.5% of failures caused by soft errors during hypercall execution [6]) and that recovery can be successful even without expensive mechanisms to handle corrupted state [8]. In these situations, the hypervisor crashes or hangs, preventing any execution to take place, and all VMs become unavailable (in spite of their internal state remaining correct). Successfully resuming a VM after a hypervisor failure, *e.g.*, due to a transient hardware fault or a software fault, has not been considered so far in literature because the hypervisor has unrestricted access to the hardware, including

The associate editor coordinating the review of this manuscript and approving it for publication was Moussa Boukhnifer.

every VM's memory space, and its failure may propagate to the VMs [6], [7].

Fault injection experiments presented in this paper show that our hypothesis holds and suggest that VMs can be recovered after hypervisor failures. Based upon this observation, we propose a generic and transparent fault tolerance technique to recover multiple VMs through migration between two hypervisors (more specifically, a failed hypervisor and a healthy hypervisor) co-residing on the same physical hardware. The technique, which we named 'Romulus', inserts a lightweight layer between the hardware and the hypervisors that monitors the state of the VMs, triggers recovery action on hypervisor failure and performs part of the VM migration process. This layer is an essential component that allows this technique to provide fault tolerance without external hardware resources (*e.g.*, Remus [9] requires two physical hosts), in a transparent and generic manner (*i.e.*, without modifications to the VMs or the applications) and with acceptable performance overhead.

A proof-of-concept implementation, which employs design principles that can be used to overcome the implementation challenges inherent to Romulus, was developed over Xen and evaluated using fault injection of bit-flips in CPU registers during hypervisor execution and software faults in hypervisor source code, as to emulate realistic hypervisor failures. The source code of the proof-of-concept has been made available publicly with the objective of fomenting research and discussion around this subject. The results show that Romulus is capable of recovering at least one VM after the large majority of failures, whereas all VMs in the system can be recovered successfully but not as often. Downtime is in the order of a few seconds, which is accomplished by simply resuming VM execution instead of restarting it.

The contributions of this work include:

1) A study on the correctness of the internal state of the VMs after a hypervisor failure. To the best of our knowledge this is the first study to evaluate whether the state of the VMs remains correct and can be recovered. Until this moment the assumption was that whenever the hypervisor failed, all VMs were lost as well;
2) A lean fault tolerance technique for tolerating hypervisor failures that allows VMs to transparently continue execution on a co-located hypervisor, thus reducing downtime, increasing availability and without needing external hardware;
3) An open-source proof-of-concept implementation of Romulus, along with design principles that can be used to develop other implementations of the technique, and an evaluation of its effectiveness using fault injection to emulate realistic hypervisor failures.

The remainder of the paper is organized as follows. Section II summarizes research in the topics closely connected to this work. Section III describes the proposed fault tolerance mechanism in a generic manner and Section IV provides the details of the proof-of-concept implementation. Section V details the experimental methodology used throughout this paper. Section VI contains the results obtained from the fault injection campaigns and the evaluation of the proof-of-concept. Section VII discusses the principal observations to take from the results, as well as the limitations of the approach, proof-of-concept and evaluation. Section VIII presents the conclusions and future work.

II. RELATED WORK

Cloud computing [10] has revolutionized the way that clients acquire and use computing resources, reducing upfront costs and avoiding payment for unused resources. It allows cloud providers to rent idle computing resources and enables workload consolidation, *i.e.*, sharing physical resources across multiple clients, in an isolated manner thanks to the development of mature virtualization support. In fact, studies have shown that consolidation in public cloud providers, despite far from maximized [11], still equates to an average of 10 VMs and a 95% percentile of 31 VMs being executed over the same hardware [4]. While economic and energy saving benefits derive from multiple clients sharing the same resources, new challenges appear, namely in terms of isolation between VMs, availability [12], reliability [13] and security [14], [15].

Failures are events that occur 'when the delivered service deviates from correct service' [5] and may be caused due to hardware, software and operator faults. Soft errors, *i.e.* temporary errors caused by transient hardware faults that happen due to ionizing radiation hitting a semiconductor device [16], [17], are specially prominent in cloud computing due to the tremendous amount of microprocessors [18] and the usage of energy-saving techniques, such as dynamic voltage frequency scaling (DVFS) [19], which have been shown to greatly increase the soft error rate [20], [21]. Furthermore, advancements in microprocessor manufacturing, which yield lower nodal capacitance and higher transistor density, cause an increase in the soft error rate [21], *i.e.*, the probability of occurrence of a soft error. Software faults are also a threat to the dependability of cloud computing and virtualized systems, particularly software faults in the components required for virtualization (*e.g.*, hypervisor, toolstack, privileged virtual machine) and cloud management [22].

Failures are particularly preoccupying for cloud computing systems because the impact of a single failure is magnified across the multiple clients hosted on a node. This observation is specially true when the fault affects the execution of the hypervisor (*i.e.*, the software component that supports the virtualization of VMs), which has been shown to cause mostly hangs that can affect multiple VMs and the less common case of data corruption [6], [7], [23]. However, the weight of the hypervisor on the overall availability and reliability of a node depends on the amount of CPU time spent in this component, which may vary significantly depending on the amount of VMs consolidated on the system, the type of workloads and details about the virtualization mode, but has been shown to range between 10% and 40% of a CPU time [7].

Various fault tolerance or recovery mechanisms against failures in cloud computing and virtualized systems have been proposed in the past, however high-availability operation has traditionally implied expensive setups that require more than one physical node, which sometimes are geographically far away [24], [25], and incur significant performance overhead during runtime and operation costs, both due to upfront and recurring costs (*e.g.*, energy consumption, bandwidth usage). Remus [9] relies on a secondary passive host that constantly receives updates with the most recent state of the VMs and which takes control when a failure is detected in the active host. Hence it is capable of tolerating transient and permanent hardware faults that cause a full-stop crash of the system without corruption of the VMs, at the expense of twice as much hardware resources and a performance hit due to the action of snapshotting a VM and sending its state over the network. COLO [26] employs the same concept of VM replication as Remus but uses an active-active and on-demand approach that monitors the external output produced by the VMs to trigger replication. As such, a VM only needs to be replicated when its output can be detected to differ, by comparison between both replicas. PLOVER [27] combines active-passive backup with state machine replication across three nodes, thus overcoming some limitations of Remus and avoiding expensive state transfer.

ReHype's [23] approach applies the concept of microreboot [28] (*i.e.*, rebooting of fine-grained components in order to renew possibly corrupted state and hence recover from a failure) to the hypervisor. It provides a fault tolerance mechanism against hypervisor failures that is attained by temporarily pausing the VMs while the hypervisor is rebooted and specific data structures are renewed with safe values. Results, obtained from a fault injection campaign of single bit-flips into registers during execution of the hypervisor, show decent VM recovery performance (over 90% probability of recovery of at least 1 VM in a 3 VM system, and around 70% chance of recovery of the 3 VMs) with no performance overhead. A posterior work [29] states a basic recovery latency of less than 3 seconds for a single VM, measured through ping timings, which is then reduced down to around 700 ms. A derivation of ReHype that is based on microreset is able to attain almost as good recovery rates with a recovery downtime close to 20ms [30].

RootHammer [31] aims to reduce the time required for a normal hypervisor reboot by maintaining the VM state in memory during this process and quickly resuming the VMs after the reboot has completed.

HyperFresh [32] presents a technique based on nested virtualization [33] and memory co-mapping for replacing a possibly corrupt and unstable hypervisor with a fresh hypervisor in as low as 100ms, which can be employed as a software rejuvenation [34] mechanism to recover from transient software failures caused by latent and non-deterministic software faults.

DualVisor [35] uses redundant VM execution and data structures on the same physical host as a technique for detecting and tolerating errors caused by hardware faults.

TinyChecker [36] uses nested virtualization to support monitoring of the communication between VMs and hypervisor and to duplicate key data structures, in an effort to detect and protect VMs from a misbehaving hypervisor.

III. FAULT TOLERANCE USING ROMULUS

Romulus is designed to tolerate the most frequent types of hypervisor failure [6], [7], which are common-mode failures that cause the entire virtualized system, including all of the VMs, to become unresponsive due to a crash or hang of the hypervisor. Its success depends on the failure being contained to the hypervisor and not propagating nor corrupting the VMs. Fortunately, this is the case with a large number of failures, as will be shown in this paper.

The technique consists of a novel form of VM migration where VMs are migrated from the failed hypervisor and resumed in a co-located and working hypervisor. The novelty of the migration resides in extracting the VM state from an unresponsive or uncooperative hypervisor in a transparent manner and resuming the VMs in another hypervisor. Minimizing migration duration is essential for increasing availability and Romulus accomplishes a low downtime, which is mostly spent performing state migration, because its VMs can resume operation immediately after migration and without requiring a costly reboot.

Romulus aims to be generic enough to be implemented for any applicable hardware architecture (*e.g.*, x86 or ARM, Intel or AMD CPUs), hypervisor or other software component being considered and requires the compliance with a set of requirements: *i*) a microvisor, *i.e.* a minimal hypervisor, must be added directly above the hardware for managing the failure detection and migration process; *ii*) the microvisor must support nested virtualization; *iii*) above the microvisor, two full-fledged hypervisors must be instantiated; *iv*) the microvisor must support virtual machine introspection (VMI) to enable the migration of VMs between hypervisors; *v*) the hardware platform must support hardware-assisted virtualization (*e.g.*, through Intel VT-x or AMD-V extensions); *vi*) the VMs to be recovered must be virtualized using hardware-assisted virtualization. Most of these requirements are already widely found in real deployments, which means that Romulus can easily be applied to these systems. For example, hardware-assisted virtualization is one of the most commonly used virtualization modes and current hardware has broad support for this virtualization mode. Furthermore, most hypervisors support virtual machine introspection and various libraries have been created to facilitate its usage (*e.g.*, libVMI [37]).

A. MICROVISOR

The insertion of a thin software layer, called *microvisor*, directly above the hardware is essential both to support the novel migration process and to host two hypervisors over the same physical hardware. The microvisor is a barebones and lean hypervisor that implements only the most essential functionalities, such as virtualizing a VM, scheduling CPU execution and managing the system memory. When compared to a traditional hypervisor, the microvisor does not need to implement functionalities such as providing a complex interface for management by the user (*e.g.*, through a toolstack), containing device drivers to interact with the hardware, implementing mechanisms for inter-VM communication or memory sharing (*e.g.*, grant-based memory sharing and event channels), and more. In fact, the microvisor delegates as many functionalities as it can to other parts of the system and strives to occupy the least amount of CPU time that is possible, both in order to reduce its overhead on the system and to reduce its exposure against transient hardware faults. Moreover, its limited set of features means that the microvisor needs only a relatively small number of lines of code, which in conjunction with the usage of classical software engineering techniques can lead to the creation of a robust piece of software.

The introduction of the microvisor replaces the single point of failure of a traditional virtualized system, which is the hypervisor and privileged VM, with the microvisor. Since the microvisor itself becomes the new point-of-failure, efforts should be taken to reduce the surface that is susceptible to faults. In practical terms, the microvisor should contain the least possible amount of code lines, as to reduce the likelihood of software faults, should execute the least amount of time, as to reduce the probability of being affected by a transient hardware fault, and, in general, should implement only the absolutely necessary functionalities. Furthermore, techniques such as formal verification of the microvisor are good candidates to ensure its correctness and have already been applied to hypervisors and operating systems in the past [38]–[40].

B. ARCHITECTURE AND NESTED VIRTUALIZATION

Above the microvisor, two full-fledged hypervisors (such as Xen or KVM) are spawned. One of the hypervisors will be actively operating and hosting VMs, while the remaining hypervisor will be idling or suspended without any VM running over it. This dual-hypervisor setup can be extended to include either a pool of idle hypervisors or a mechanism that destroys the active hypervisor after the migration process has been concluded and replenishes the system with a new idle hypervisor for future use.

Both hypervisors may either share the same implementation (*i.e.*, source code) or use different hypervisor implementations (*e.g.*, Xen and KVM). The first option is easier to configure and provides protection against transient hardware faults and certain software faults, such as mandelbugs and aging-related bugs, whereas the second option also provides coverage against other software faults (*e.g.*, some bohrbugs) through design diversity.

The existence of a new layer between the hardware and the hypervisors means that nested virtualization [33] is essential

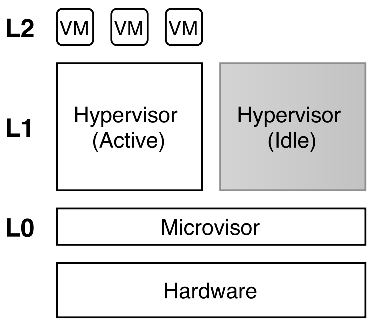


FIGURE 1. Layers of a setup that uses the proposed approach.

to the implementation of Romulus, whose architecture is divided into 3 layers of virtualization, as shown in Figure 1. The first layer (L0) is the microvisor, the second layer (L1) are the two hypervisors and the last layer (L2) are the VMs managed by the clients (IaaS) or the cloud provider-managed VMs providing a service (SaaS and PaaS).

C. LIFECYCLE

The lifecycle of the technique can be divided into 3 phases: preparation, monitoring and migration. The preparation phase is used to extract the static and semi-static state (*i.e.*, the state that does not change or rarely changes during a VM's lifecycle), specifically the configuration and emulation state, for each of the VMs that will be recovered when a hypervisor failure occurs. The monitoring phase takes up the majority of the lifetime and consists in two main actions: monitoring and storing VM state that may change dynamically, as is the case with some of the CPU state, and monitoring the health of the hypervisor and/or its VMs in order to detect when a recovery action must be performed. The approach used to monitor the hypervisor health and to trigger the recovery action when a failure is detected (*i.e.*, the triggering mechanism) is not the main focus of this article and any valid option may be combined with this failure recovery mechanism.

The first step in the migration phase is to pause the execution of the failed L1 hypervisor, in order to prevent corruption to its data structures or VMs. In the case that the hypervisor has already sent a shutdown signal (*e.g.*, as a response to a segmentation fault from hypervisor code), it is essential that the microvisor preserves its memory state (*i.e.*, it should not free the memory pages). Then the microvisor must employ virtual machine introspection to extract the missing VM's state from the hypervisor's memory, as detailed in Section III-D. This implies that the microvisor knows the offsets and sizes of the hypervisor's data structures, hence it requires a hypervisor-dependent configuration. When all of the required VM state has been obtained, the migration can take place. Firstly the memory of the VM is moved from the failed to the sane hypervisor. This operation is implemented by the microvisor without the need for memory copying, simply by rearranging its physical-to-machine table so that the memory pages which contain the state of the L2 VM and that were previously mapped by the failed L1 hypervisor now belong to the sane hypervisor.

Algorithm 1 Algorithm for Memory State Migration of One VM in an Intel System

```
Input: EPTP
1: $pml4 = *EPTP$
2: change_ownership(pml4, L1_A, L1_B)
3: for $l0 = 0$ to 511 do
4:     $pdpt = pml4[l0]$
5:     if (pdpt is valid) then
6:         change_ownership(pdpt, L1_A, L1_B)
7:         for $l1 = 0$ to 511 do
8:             $pd = pdpt[l1]$
9:             if (pd is valid) then
10:                change_ownership(pd, L1_A, L1_B)
11:                for $l2 = 0$ to 511 do
12:                    $pt = pd[l2]$
13:                    if (pt is valid) then
14:                        change_ownership(pt, L1_A, L1_B)
15:                    end if
16:                end for
17:            end if
18:        end for
19:    end if
20: end for
```

Algorithm 1 contains a pseudo-code implementation of this process for a system that uses Intel EPTs hardware extension, although a similar algorithm could be produced for other hardware extensions that provide memory virtualization. The symbol `EPTP` represents the location of the nested page table structure that contains the physical-to-machine mapping of a VM's pages. This multi-level data structure usually contains four levels (herein represented by the symbols `pml4`, `pdpt`, `pd` and `pt`, following Intel's nomenclature), where each level holds 512 entries that point to other pages. While the first three levels contain pointers to the next levels, the last level (*i.e.*, `pt`) contains pointers to the real memory pages. Given the EPTP as an input, the algorithm iterates over every valid (*i.e.*, allocated) page in the data structure and calls the `change_ownership` function to move the ownership of every page from the old hypervisor (represented by `L1_A`) to a new hypervisor (`L1_B`). Ultimately, changing page ownership as described is quicker than copying memory pages, which will accelerate the migration process.

After the microvisor exchanges the ownership of all of the memory pages from the failed hypervisor to the sane hypervisor, it should use the hypervisor's default suspension and resume mechanism for restoring the VM in the sane hypervisor. This step can be simplified by updating a template save file with the most recent known VM state (*e.g.*, configuration, CPU and emulation state) and using it for resuming the VM.

To complete the recovery of a VM, on the previously idle hypervisor's (now active) side, the capability for restoring a VM's memory state from a memory location containing the entry of the nested page table must be available. This restoration of a VM's memory differs from how most hypervisors' transfer memory state, which is by adding the memory page's contents to the save file and copying the contents into memory on VM restore. This method of restoring a VM's memory state is required in this situation due to the way the microvisor exchanges the ownership of memory between hypervisors (based on the entry pointer for the nested paging table), however it also brings a significant performance advantage by avoiding copying between memory and disk, which ultimately reduces the downtime during recovery.

After the recovery of all VMs is complete, the paused and failed hypervisor can be destroyed and its resources, including the memory pages that still belong to it, can be freed. At this point the lifecycle may restart, but not before a new idle hypervisor instance is spawned to serve as the sane hypervisor.

D. OPTIMIZED STATE EXTRACTION

One of the main elements of novelty in Romulus is the method for extracting VM state from a crashed or uncooperative hypervisor, which is an essential part of the process for tolerating and recovering VMs from a hypervisor failure. For this purpose, virtual machine introspection (VMI) [41], that is 'monitoring and analyzing the state of a virtual machine from the hypervisor level' [41], is performed by the microvisor, thus enabling extraction of the required state without hypervisor intervention as long as the internal structure is known *a priori*.

The state that is required for migration can be divided into:

1) *configuration state* – refers to details about the VM, such as how many memory pages, disks, CPUs, *etc.* it has;
2) *memory state* – refers to the memory pages assigned to a VM, which contain the data of the kernel and user space processes;
3) *CPU state* – refers to the register values for each of the CPUs used by the VM and to the data structures required by virtualization extensions (*e.g.*, VMCS and VMCB which are mandatory when using Intel's VMX extension);
4) *disk image* – contains the information stored by the VM in its physical storage;
5) *emulation state* – refers to the auxiliary state needed by the hypervisor to perform the virtualization and usually refers to the state of I/O devices (disks, network interfaces, *etc.*).

Different state requires specific methods for obtaining, treating and reusing it. Configuration state tends to be static and defined at VM boot time, hence it can easily be pre-obtained. Memory state is obtained and restored by taking advantage of the available memory virtualization extension (Intel EPT, AMD NPT, *etc.*) which keeps a multi-level paging structure in memory that describes the memory structure of a VM. CPU state, in particular the register state, is obtained by introspecting the L1 hypervisor structures that keep track of each VM's CPU state. However, different hypervisor implementations may keep track of different amounts of state, and part of the state may not be stored by the L1 hypervisor. For these situations, the microvisor must keep track of the missing state itself by trapping nested exits from the L2 VM and storing the required state. Disk state can be obtained and shared between L1 hypervisors using a range of well-established techniques, such as through a network file system, hence we will not dwell on this point. Emulation state, while not very dynamic, may change (for example when a previously disabled network interface becomes active) and is particularly difficult to obtain, as this state is usually stored in user-space processes (*e.g.*, QEMU) and hence harder to find and obtain. However, it can still be obtained from the L1 hypervisor's memory using VM introspection, or simply by relying on an outdated, but possibly correct, snapshot that is obtained after start up of a VM.

IV. PROOF-OF-CONCEPT

Romulus requires being able to transparently extract a VM and resume it in a different hypervisor, which is a task that has several inherent challenges. For example, how should one track the memory and CPU state of a L2 VM and keep it up-to-date or how should a hypervisor be modified to allow resuming a VM that was executing in another co-located hypervisor. This section describes how such challenges were dealt with in our proof-of-concept, hence providing a guide that can be used by others to support their own implementation of Romulus.

Our proof-of-concept implementation of Romulus was created over the source code of the universal Xen hypervisor [42], which was used as the basis for the microvisor. VMI was provided by libVMI v0.12.0 compiled with caching disabled and Intel VT-x hardware extensions are required. The code of the proof-of-concept implementation is available under an open-source license at https://github.com/ucx-code/romulus.

With respect to the amount of modifications applied to Xen 4.11.1, a total of ~1K lines of code were modified for the microvisor, ~500 lines were modified for the idle hypervisor, which we will designate as *L1B* for ease-of-use, and 23 lines were modified for the active hypervisor, which we will designate as *L1A*, although it would have been possible to use an unmodified Xen 4.11.1 as the extra code has solely debugging purposes (*e.g.*, it helps to find the offsets of the hypervisor's data structure).

Our implementation was developed to support the experimental campaigns performed in this paper. If the proposed technique was to be used in a production environment, the microvisor would ideally be developed from scratch, with special attention given to reducing its code size, focusing on a target architecture, refraining from implementing non-essential functionalities and, possibly, using defensive programming or formal verification techniques.

1) MICROVISOR

Xen was extended with two new hypercalls: *save_nvmcs* and *migrate*. The *save_nvmcs* hypercall obtains and triggers the monitoring of the CPU state that is not already being tracked by Xen. Specifically, this hypercall activates the execution of a branch added to the *nvmx_n2_vmexit_handler* function (which is called after the exit of a L2 VM) that extracts and keeps in memory the values related to the ES, FS, CS, GS, DS, SS, TR, GDTR, IDTR and LDTR registers. When this branch is activated a small performance penalty can be expected. Another approach was tested, which consisted in reading the values directly from the memory structure where they are stored (known as VMCS [43]), which is required by the virtualization hardware extension, but such is not advisable as this structure must be accessed by calling the *vmread* [43] instruction, and furthermore does not follow a well-defined or fixed schema (it may vary from CPU to CPU).

The *migrate* hypercall receives as input the EPTP of the L2 VM to be migrated and the memory location on the idle hypervisor to where the L2 VM's memory should be moved and performs the migration.

The toolstack was adapted to support correct calling of these hypercalls. A range of userspace utility applications was developed to: *i*) use VMI to obtain the EPTP and part of the CPU state of the L2 VM; *ii*) replace the contents of a base save file (created after the L2 VM has been spawned) with a more recent CPU state and the EPTP of the L2 VM on the idle hypervisor (after migration); *iii*) remove the page tables that Xen stores on the save file, in order to avoid higher migration time due to unnecessary information on the save file.

2) HYPERVISOR

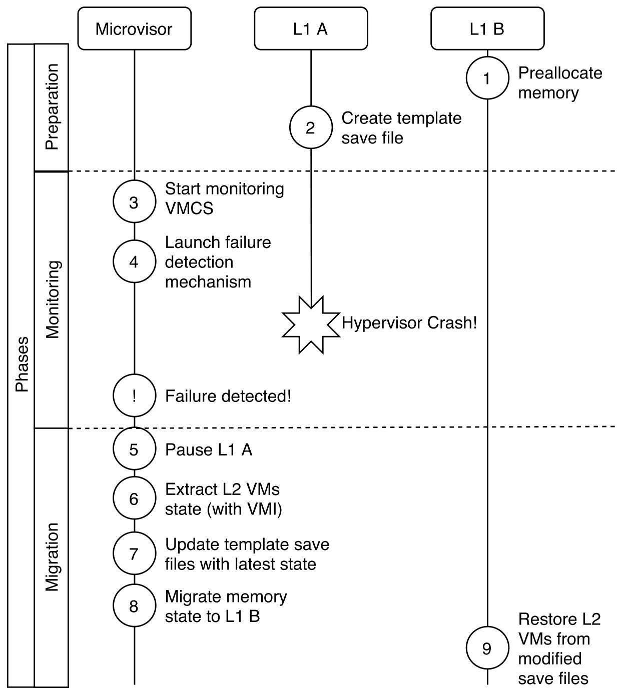
Modifications are only required for the L1 B hypervisor and these provide the capability to restore a VM from a save file that contains an EPTP. This extends over the normal functionality of Xen which expects the memory content of the VM to be embedded on the save file. Furthermore a simple hypercall and matching toolstack code was added to prepare the hypervisor to receive the memory of the L2 VMs from the active hypervisor. This can be accomplished by calling the *alloc_domheap_pages* function to allocate the free memory and reserve it for future use.

3) LIFECYCLE AND FLOW

Figure 2 presents the flow and actions taken during the lifecycle of a system that uses this proof-of-concept. The shown flow assumes that, at the start, all L1 and L2 VMs are up and running, and that the save files and IO state of the VMs is acessible to the microvisor and both L1 hypervisors. A total of 9 actions have been identified and grouped into the 3 phases that had been presented in Section III-C as constituting the lifecycle of the technique.

4) LIMITATIONS

Due to time constraints and since this is solely a proof-of-concept, there are several limitations, most of which are



Microvisor
L1 A
L1 B
Phases
Preparation
Monitoring
Migration
1 Preallocate memory
2 Create template save file
3 Start monitoring VMCS
4 Launch failure detection mechanism
Hypervisor Crash!
! Failure detected!
5 Pause L1 A
6 Extract L2 VMs state (with VMI)
7 Update template save files with latest state
8 Migrate memory state to L1 B
9 Restore L2 VMs from modified save files

FIGURE 2. Lifecycle and actions.

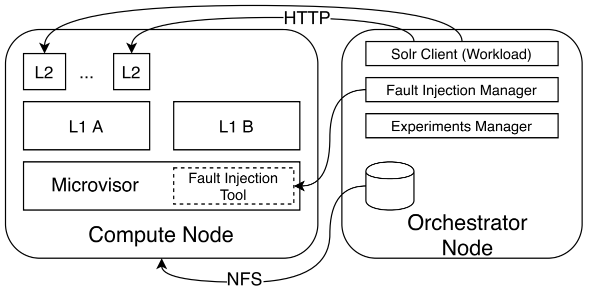


FIGURE 3. Experimental setup used for the experiments.

TABLE 1. Statistical analysis of both workload profiles.

		Full load			Light load		
		Max	Avg	Med	Max	Avg	Med
CPU	Total (%)	100	88	100	100	21	14
	User (%)	83	30	30	89	9	6
	Kernel (%)	90	57	66	68	8	4
	IO Wait (%)	28	0.3	0	48	4.3	0
Disk	Read (TPS)	99	13.6	9.6	104	7	0
	Read (Mbps)	65	9.7	5.6	73.8	5	0
	Write (TPS)	68	1.3	0	38.6	1	0
	Write (Kbps)	415	20.5	0	280	12	0
Net.	Sent (seg./s)	766	365	423	625	33	8
	Recv. (seg./s)	682	360	420	376	30	12

relatively easy to fix and do not represent an inherent limitation of Romulus. One limitation is that recovery is only supported for L2 VMs that have just a single vCPU, do not use hyperpaging (*i.e.*, page sizes bigger than 4Kb), use an Intel EPT hardware virtualization mode (HVM in Xen), do not use Xen's PV-on-HVM drivers (*e.g.*, by passing 'nopv' parameter to recent Linux kernels). Furthermore, LAPIC, APIC, MCE, XSAVE and X2APIC should not be used in the L2 VMs and hyperpaging must be disabled on both L1 hypervisors, but may be enabled in the microvisor.

V. EXPERIMENTAL METHODOLOGY

Experimental evaluation using fault injection was performed to validate the assumptions and contributions of this paper, including the proof-of-concept implementation. A consistent set of workload, physical setup and tools were used for the evaluation, which are described in this section.

A. PHYSICAL SETUP

The setup used for the experiments is comprised of two different physical systems, as depicted in Figure 3. One of the systems has the role of 'Compute Node' (*i.e.*, hosting the VMs) and the other is used as the orchestrator, client and disk image provider for the VMs. The compute node is equipped with two Intel Xeon Silver 4114, each with ten physical cores, 32 Gb of RAM and a network interface capable of 1 GbE. The orchestrator machine is equipped with a single Intel Xeon E5620 with four physical cores, 12 Gb of RAM and a 1 GbE network interface.

The disk images used by the VMs that are running on the compute node are stored and provided by the orchestrator machine through NFS [44]. This is a common setup found in the cloud (although using more advanced technologies [45]) where there is one or more nodes dedicated to storing and providing disk resources.

In the experiments where fault injection was used to emulate transient hardware faults both hypervisors used Xen 4.11.1, whereas when software faults were injected L1A used Xen 4.12.3 and L1B used Xen 4.11.1. The usage of different hypervisor versions reflects a scenario where Romulus is used to tolerate software faults of transient nature (*i.e.*, mandelbugs and aging-related bugs, which are only activated when specific conditions regarding the system state are found) through state rejuvenation and software faults of permanent nature (*i.e.*, bohrbugs) through design diversity. If the same hypervisor version had been used instead, Romulus would only have been able to cover software faults of transient nature.

B. WORKLOAD AND PROFILES

The workload used during our experimental evaluation emulates a Solr server [46] that provides access to a part (11 Gb) of Wikipedia's index [47]. This is a CPU and IO-heavy workload that also exercises memory. Only search operations are performed during the course of the workload.

Two different profiles were used in our experiments: a *full load* profile, which consists in 25 clients performing a request once every 50 milliseconds and represents a worst-case scenario where the VM is overloaded, and a *light load* profile, which represents a more common scenario [11], [18], [48]–[50] where the system seldom reaches full resource usage and consists of 1 client performing a request every second. Figure 4 and Table 1 provide insight into the resource usage of both workload profiles on a VM with 3 000 Mb of RAM.

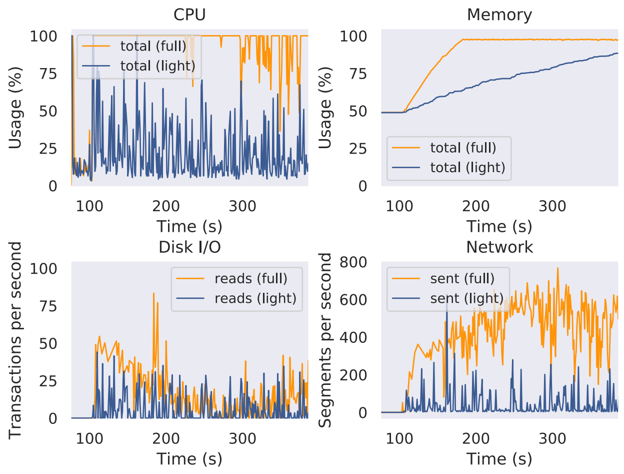


FIGURE 4. Resource usage of the workload and its profiles.

C. FAULT INJECTION

To emulate failures of the hypervisor, fault injection of transient hardware faults in CPU registers and software faults in hypervisor code was employed. The representativeness and accuracy of the fault injection process and fault model was ensured by following the best practices in the field, as is detailed below.

To emulate transient hardware faults, a fault injection tool was coded and integrated into the microvisor code. It operates by performing single bit-flips [51] on the register values of a VM during a context switch and is capable of targeting the RIP, RSP, RBP, RAX, RBX, RCX, RDX, and R8 to R15 registers and producing a log with the pre- and post-injection register values. It is also capable of targeting specific processes, including all of the hypervisor code, by filtering injections according to the virtual memory address pointed to by the RIP on the moment of context switch. Since it depends on a context switch it might incur some latency between the order to perform an injection and its execution, but since we aim to emulate random occurrences of faults we believe that a small delay does not represent a drawback. Furthermore it requires that vCPU pinning (*i.e.*, associating a physical CPU with a vCPU) be avoided, as this technique eliminates context switching.

For injecting software faults in hypervisor code, an existing fault injection tool was used to generate patch files containing realistic software faults according to the fault model defined by Durães *et al.* [52]. These patch files are applied to the source code of the L1A hypervisor one patch per each run during the fault injection campaigns. In order to speed up the evaluation, an analysis of the lines of the hypervisor source code that are covered during the workload was performed and all faults that do not affect those lines were filtered out.

A requirement we set for our experiments was that the software fault should only be activated while the workload (*i.e.*, Solr) is being executed, thus after any preparatory steps (*e.g.*, creating base save files) have been completed. This was accomplished by encompassing the software fault inside an *if* whose activation depends on a variable (a similar practice has been used in other works [53], [54]) that is manipulated by the microvisor using VMI. When the value of the variable is false, the non-faulty code is executed. Moreover, for tracking purposes the exact moment when the fault is first activated is stored, as well as the number of iterations that occurred before and after the fault was activated.

Faults were only injected in lines of the L1A hypervisor, which runs Xen 4.12.3, that do not exist in L1B hypervisor, which uses Xen 4.11.1, as to emulate recovery from a software fault that was introduced in a more recent version of Xen and does not exist in older versions. A comparison between the source code of both versions was made which showed that about 8% of lines of source code (comments were disregarded) differed between the two versions, including lines of source code in files that implement essential functionality, thus suggesting that migrating VMs between different versions of the same hypervisor, even if both versions are relatively recent, can provide some coverage against software faults through design diversity.

When performing fault injection of hardware faults, twelve registers were targeted (RIP, RSP, RBP, RAX, RBX, RCX, RDX, R8-R15) which represent all the registers that can be targeted using this fault injection tool. For software fault injection, the fault model includes 10 operators out of the 13 operators defined in Durães *et al.* fault model [52] and used 2 operators belonging to an extended fault model [55] (namely, WLEC and WALR). In summary, the used operators were WVAV, WPFV, WLEC, WAEP, MVAV, MVAE, MLAC, MIFS, MIEB, MIA, MFC and WALR, which are shown and briefly explained in Table 2.

TABLE 2. Software fault model operators.

Operators	Description
MFC	Missing function call
MIA	Missing `if` construct around statements
MIEB	Missing `if` construct plus statements plus `else` before statements
MIFS	Missing `if` construct an surrounded statements
MLAC	Missing `and` sub-expr. in logical expression used in branch condition
MVAE	Missing variable assignment with an expression
MVAV	Missing variable assignment with a value
WAEP	Wrong arithmetic expression in parameters of function call
WPFV	Wrong variable used in parameter of function call
WVAV	Wrong value assigned to a variable
WALR	Wrong algorithm – code was misplaced
WLEC	Wrong logical expression used as branch condition

Software faults were injected in 4 different source files, namely *arch/x86/hvm/vmx/vmx.c*, *arch/x86/hvm/vmx/vmcs.c*, *arch/x86/msr.c* and *arch/x86/mm.c*, which were chosen due to being the files that had the highest amount of changed lines between Xen 4.11.1 and Xen 4.12.3 that were exercised by the workload. These files contain functionality that deals with memory virtualization, hardware-assisted virtualization and model-specific register (MSR) emulation.

D. TRIGGERING MECHANISM FOR THE RECOVERY ACTION

The evaluated fault tolerance technique has to be paired with a mechanism capable of detecting when a hypervisor or a

VM has failed as to commence the recovery process. Multiple options are available for this key element, from more generic to application specific ones. The goal in selecting the best alternative for the recovery mechanism resides in reducing the downtime incurred from the detection interval and increasing the precision and sensitivity of the mechanism (*i.e.*, reducing the false positives and false negatives). For our experiments we opted to use a straightforward mechanism which consists in a constant stream of ping requests which will trigger recovery after a certain amount of consecutive pings has timed out. This proved sufficient for our requirements, which were to detect occasions where the hypervisor had a crash or hang failure which caused all the other VMs to fail (common-mode failure), however it might not be capable of detecting more complex failures where silent data corruption has occurred. Furthermore, it represents a baseline error mechanism, which means that better error mechanisms may possibly result in even better performance of Romulus than what is presented in this paper.

E. CORRECTNESS VERIFICATION

The workload client is prepared to verify the correctness of the received responses according to the query that was searched. This is accomplished by having a pre-prepared list of all possible search queries and the expected response, and then comparing all performed queries and respective response against the oracle, during workload execution. This step is essential in ensuring that failure recovery can be accomplished without corrupting the VM state in a manner that leads to incorrect output being sent to the service's clients.

F. RECOVERY ASSESSMENT

Apart from ensuring the correctness of the responses, there is the need to assess whether each VM recovered successfully from a failure of the hypervisor. For this purpose two different 'points-of-view' are considered:

1) Service point-of-view – Recovery is measured according to whether the service (Solr) continued to execute correctly after the hypervisor failure. This is the traditional point-of-view, which corresponds to how the service clients experience unavailability;
2) Operating system point-of-view – A VM is considered recovered if its operating system continues to operate correctly, even if the service stops working. The state of the operating system is verified by performing a SSH check and executing a small number of simple commands at the end of each experiment. We consider this point-of-view because, although the service may not be successfully recovered, the operating system may continue to work correctly and specific methods can be designed to take advantage of this.

VI. RESULTS

The results are divided into two parts: the first part evaluates the hypothesis that sustains the presented technique (*i.e.*, after a hypervisor failure, VMs may remain uncorrupted and can continue execution on a new hypervisor) and the second part evaluates the performance and effectiveness of the proof-of-concept. In terms of metrics, the experiments measure recovery effectiveness (*i.e*, the probability of a VM being successfully recovered after hypervisor failure), migration time (*i.e.*, the time taken in the process of migrating VMs between hypervisors), downtime (*i.e.*, how long the VMs become unavailable while recovery is taking place) and run-time overhead. To effectively measure the various metrics, various experiment campaigns with slightly different parameters were used (some of the varied parameters include the number of VMs being executed, VM memory size or the workload profile). In such cases, the leading text will clearly indicate the parameter values used in that specific experiment. Nevertheless, various control variables remain constant across experiments. For example, the fault models (*e.g.*, CPU registers that suffered fault injection).

A. HOW ARE VMs AFFECTED WHEN THE HYPERVISOR FAILS?

When a hypervisor fails, it may fail without affecting the state of its VMs or it may gain an erratic behaviour and eventually corrupt its VMs [6]. If recovery is to be attained the basic principle that the VM state has to remain sane must be upheld. To evaluate this hypothesis, an experimental campaign was setup where a single VM was migrated to a new hypervisor after its hypervisor failed, as to assess whether it continues correct execution.

Fault injection in CPU registers during hypervisor execution between 15 and 40 seconds after the start of the workload was used to generate hypervisor failures. The timing values were picked as to allow the VM to warmup while providing enough time for it to recover and resume responding to Solr requests before the workload finishes. Both profiles of the workload (light and full load) were evaluated during 200 seconds. The recovery process is triggered by a process that constantly performs ping checks to a VM and starts the recovery process sensibly 8 seconds after the last successful response.

A total of 339 failures using the full load profile and 663 failures using the light load profile were obtained, the results of which are displayed in Table 3. Over the course of all experiments a total of 55 678 requests were performed and no occurrence of incorrect response data was detected after a successful VM migration to a new hypervisor. Although this does not eliminate the possibility that a VM may be successfully recovered despite its state being partially corrupt due to the fault in the hypervisor having propagated, which may then lead to silent data corruption occurring inside the VM, such is not a common occurrence. Moreover, the lower the detection interval before triggering recovery action, the higher is the chance of preventing errors in the hypervisor from propagating to the VMs.

A subset of all the runs, amounting to 102 failures using the light load and 155 using the full load profile, was extended to

TABLE 3. Recovery probability (1 VM).

		Workload profile	
		Light load	Full load
PoV	Solr	189 (29%)	54 (16%)
	O.S.	40 (39%)	64 (41%)

include an operating system check through SSH at the end of the run, in order to detect situations where the VM may be responsive but the Solr process has failed. The results indicate that in many situations, specially when a heavy workload is used, the operating system of the VM is able to recover but the application under use (Solr) does not, as it is killed by the operating system. There was no run where Solr was classified as responsive but the operating system was unresponsive.

The light load profile resulted in a higher recovery percentage than when using the full load profile, which suggests that the usage level of a VM can affect its likelihood of recovery. This is not a significant problem for cloud computing systems since the majority of the dozens of VMs consolidated on a physical system are idle or have a small amount of load during short periods of time [11], [48], [50] and the presented results can be considered to be a worst-case scenario for this setting, specially when referring to the full load profile.

B. PROOF-OF-CONCEPT EVALUATION

The results from the experiments that evaluate the proof-of-concept focus on four key aspects: runtime performance overhead, duration of the recovery process, recovery effectiveness of the VMs after hypervisor failure and downtime.

1) PERFORMANCE OVERHEAD

As with any fault tolerance technique, particularly when applied to an area such as cloud computing where resources tend to be maximized as to increase business profit, the overhead of the solution can have a determining role on the decision to adopt it. We compared the performance, measured as the number of successfully answered requests over a fixed time period and their average response time, of a system that represents a traditional virtualized cloud computing system (*i.e.*, where no nested virtualization is used), a system that uses nested virtualization but does not contain the proposed fault tolerance technique, and a system with our implementation of Romulus. A total of 30 runs were performed for each setup and all the runs used a single VM with 3 000 Mb of RAM and the full load profile during 60 seconds. The L0 and L1 hypervisors, when applicable, were configured with 1 CPU and 12 000 Mb of RAM, as to represent a situation where consolidation is present. Table 4 presents the results for the light load and full load profiles, including a comparison of relative performance difference against the traditional virtualized system, which show that the proof-of-concept does not bring a measurable overhead on a system that already uses nested virtualization, however the addition of nested virtualization carries a significant overhead, which can reach up to 2.5x lower performance when a heavy workload is used.

TABLE 4. Performance overhead of different setups.

		Trad.	Nested Virt.	w/ PoC
Full	N. of requests	1234	487 (+153%)	488 (+153%)
	Avg. resp. time (s)	1.18	3.14 (-62%)	3.13 (-62%)
Light	N. of requests	96	79 (+22%)	79 (+22%)
	Avg. resp. time (s)	0.09	0.24 (-62%)	0.24 (-62%)

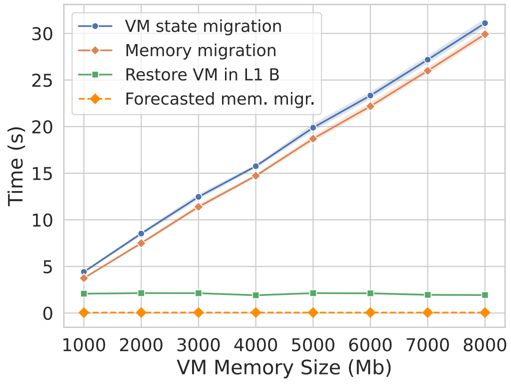


FIGURE 5. Duration of the recovery process per VM size.

2) RECOVERY PROCESS DURATION

A significant part of the experienced downtime is due to the time taken by the process of migrating all of the VMs. Since the duration of this process may vary with the size of a VM, namely its memory size, an analysis was performed where a single VM was migrated between hypervisors (without a hypervisor failure), 25 seconds after the workload, which used the light load profile, was started and using varying memory sizes, from 1 000 Mb to 8 000 Mb inclusive, at steps of 1 000 Mb, as well as executing 30 runs of each possible combination. Figure 5 presents the results.

The line with circle markers, labeled as *VM state migration*, represents all of the steps required for recovery apart from restoring the VM state at the L1B hypervisor, which is represented by the line with square markers, while the solid line with diamond markers, labeled as *Memory migration*, describes the time taken solely for the step of migrating the memory state. The total time required to complete the recovery process is not depicted but consists on the sum of the lines that have circle and square markers.

The algorithm of the recovery process has linear complexity, which grows proportionally to the number of pages that need to be migrated, as is empirically shown in Figure 5. However, there is a constant time element that is attributable to the step of restoring the VM in the new hypervisor, after the memory has been copied. It should be noted that although the number of pages has an impact in the migration time, it does not affect the recovery effectiveness of Romulus (which will be analyzed in the next section).

The dashed line represents the hypothesized time that the step of memory migration would have taken if the proof-of-concept implementation had support for hyperpaging (also known as superpaging). It was obtained by measuring how many pages would be used at the different memory levels when hyperpaging is enabled and extrapolating from the known data. This extrapolation transmits a very positive outlook for the improvements that hyperpaging can bring, since it significantly lowers the amount of pages that need to be migrated, thus reducing time spent migrating the memory state to values between 45 and 54 milliseconds. According to this data, if hyperpaging is considered we can expect the entire recovery process to take between 2 and 4 seconds.

3) RECOVERY EFFECTIVENESS

Romulus's objective is to tolerate common-mode hypervisor failures that affect multiple VMs at once, hence the most adequate metric to evaluate the effectiveness of the proof-of-concept implementation is the total number of recovered VMs and, indirectly, the probability of recovery of a VM.

For this study a system containing 4 VMs with 1 CPU and 900 Mb of memory each and executing the Solr workload with the light load profile was used. Fault injection of transient hardware faults in CPU registers and software faults in hypervisor code was once again used to produce realistic failure data. Injection took place between 200 and 210 seconds after the start of the workload and the recovery action was triggered sensibly 40 seconds after the last successful ping reply. This is a conservative timeout value as to avoid inadvertent triggering, but which may reduce the recovery effectiveness by providing the failed hypervisor more time to corrupt the VMs state. The L1 hypervisors were configured to have 6 CPUs, which means that the system had a consolidation ratio of 0.66 (or in other words, 4 L2 VMs executing over 6 CPUs).

A total of 774 hypervisor failures due to injected hardware faults and 117 failures due to software faults were collected. To obtain these failures, we had to inject over 2 000 hardware faults and over 400 software faults. A part of these faults never propagated into a failure and thus have been excluded from our analysis. In total, 8 months of continuous experiments were needed to obtain the results herein described.

Table 5(a) shows information about the registers that lead to failures, while Table 5(b) presents information about the operators that caused software failures. It should be noted that some CPU registers and operators did not cause any failures and have been omitted from these tables, nevertheless fault injection using these registers and operators was performed as usual.

Figure 6(a) and 6(b) show the histogram with the count of successfully recovered VMs discriminated by the type of fault, for the service and operating system point-of-view respectively.

Hypervisor failures caused by transient hardware faults translated to at least one VM being successfully recovered (between 96% and 91% of all failures) and an arithmetic mean of 1.82 recovered VMs (or 46% of all VMs). When a hypervisor fails due to a software fault, the mean becomes slightly lower at 1.62 recovered VMs (41% of all VMs), but the extremes increase (no VM is recoverable between 25% and 30% of the time, whereas all four VMs are recovered between 10% and 34% of the time). These observations suggest that transient hardware faults, due to their nature, are more likely to propagate to a smaller number of VMs, leaving the remaining VMs in a correct state and thus recoverable, whereas software faults are more likely to affect and corrupt either all VMs or none at all.

Comparing both of the point-of-views that were considered for classification of a recovery outcome, the service point-of-view, which is the most restrictive of the two, experiences lower recovery probability than the operating system point-of-view. Situations where the operating system was left operational but Solr was not are explained by the hypervisor failure having corrupted only state associated with Solr, which lead the operating system to kill its process after recovery. In these situations the addition of a simple application restart mechanism could greatly increase the recovery

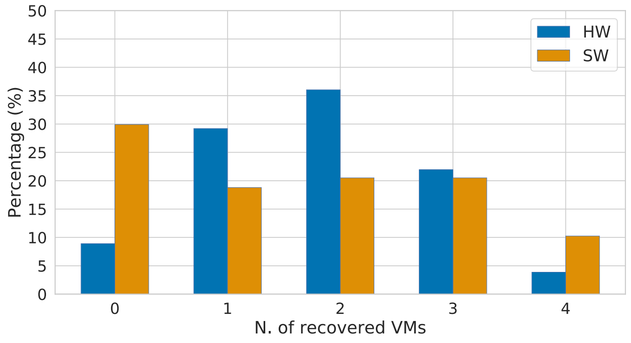
TABLE 5. Fault injection statistics.

Register	Failures	
RSP	144	(19%)
RIP	141	(18%)
RDX	100	(13%)
RBX	89	(11%)
RCX	85	(11%)
R13	80	(10%)
RBP	68	(9%)
R12	67	(9%)
Total	774	

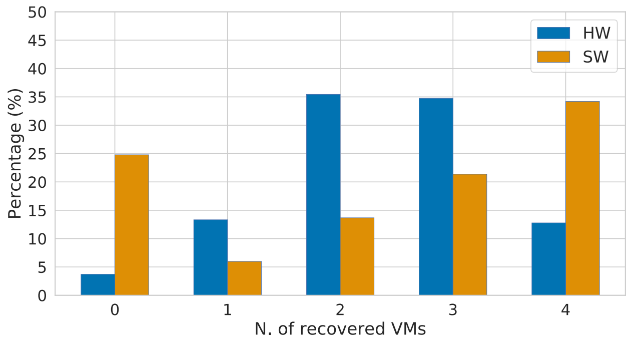
(a) HW faults.

Operator	Failures	
MIA	49	(42%)
MFC	31	(26%)
MIFS	21	(18%)
WLEC	3	(2%)
WPFV	3	(2%)
MVAE	2	(2%)
MVAV	2	(2%)
WAEP	2	(2%)
WVAV	2	(2%)
MLAC	1	(1%)
MIEB	1	(1%)
Total	117	

(b) SW faults.



(a) From the service (Solr) point-of-view.



(b) From the operating system point-of-view.

FIGURE 6. Total recovered VMs after a hypervisor failure.

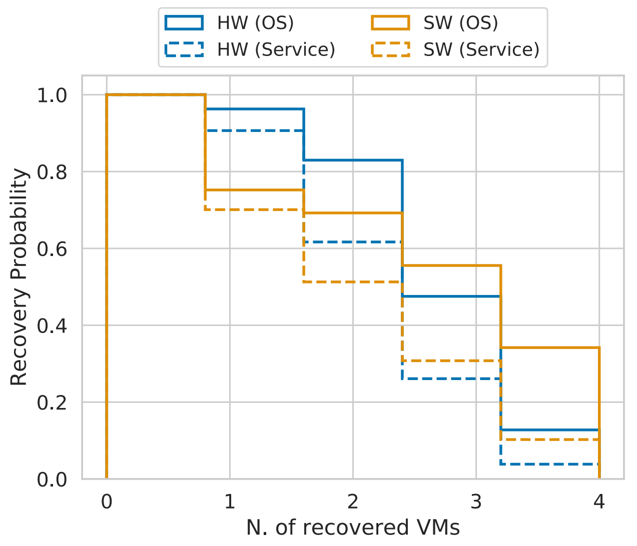


FIGURE 7. **Cumulative histogram of recovery probability.**

likelihood, although such would lose the transparency inherent to this fault tolerance technique. Figure 7 provides cumulative histograms that improve the comprehension of the data in the previous figures.

The operating system of all four VMs is recovered in 34% of all failures caused by software faults, but Solr is only recovered 10% of the time. The probability of all four VMs being recovered after a failure due to hardware faults is much lower, ranging between 13% for recovery of only the operating system and 4% for recovery of Solr. The recovery percentage increases if we consider all cases where at least 3 VMs were recovered: 56% of failures caused by software faults leave the operating system recoverable, but Solr can only be recovered 31% of the time, whereas if a failure is caused by a hardware fault, the operating system is recoverable 48% of the time and Solr 26% of the time. If we consider the operating system point-of-view, at least half of the VMs can be recovered in 83% and 69% of all failures, when caused by hardware and software faults respectively, or 62% and 51% of the time, if we use the service point-of-view. Finally, at least 1 VM is fully recoverable in 91% of all failures due to hardware faults and 75% of all failures due to software faults. Ultimately, even the recovery of one VM corresponds to a big improvement in comparison to a system that cannot tolerate hypervisor failures.

4) DOWNTIME

Although being able to successfully recover VMs is essential, the downtime incurred during recovery has an important role on the overall availability of the system and should be contemplated. Figure 8 shows the boxplot that contains the median and quartiles of the downtime for a single VM setup using both the full and light load profiles, as perceived through the clients of the Solr service (service downtime) and through the operating system logs (VM downtime), obtained using the SAR utility.

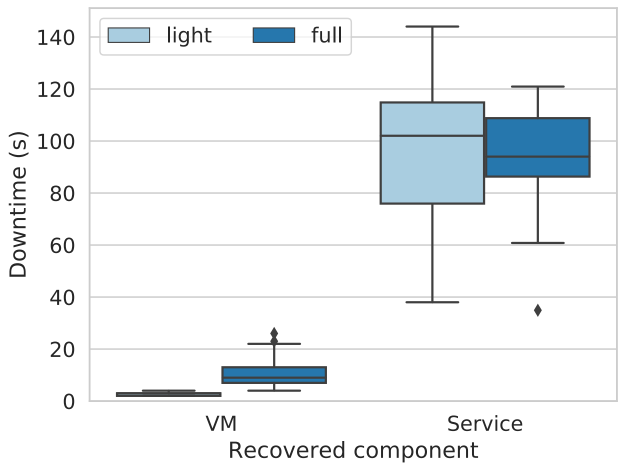


FIGURE 8. **Downtime in a single VM setup.**

The VM downtime reflects the period when the VM is operating but the network connection has not yet been restored, hence being considerably lower than the service downtime, ranging between 2 and 26 seconds with a mean of 3 seconds. The discrepancy between VM and service downtime is explained by a timeout in the network device driver used in the Linux kernel of the VMs that is only triggered after some seconds. Modifications to this device driver, or the usage of another driver, would equalize the VM and service downtimes. The 4 VM system that was used in Section VI-B3 yielded a service downtime ranging between 31 and 394 seconds with an average that varied between 254 and 275 seconds.

VII. DISCUSSION & LIMITATIONS

The results confirm that nearly half of the VMs in a system are left in a non-corrupted state after a hypervisor failure and thus can be recovered if the failed hypervisor is replaced by a working one, which is the basis for Romulus, the proposed technique which has been shown to tolerate hypervisor failures by evaluation of a proof-of-concept implementation. Despite occasions where not all VMs were successfully recovered, at least one VM was recoverable in the large majority ($\geqslant 75\%$) of cases, which, in itself, represents a big improvement over a traditional virtualized system where hypervisor failure always translates to the loss of every VM. Furthermore, given the limitations inherent to an implementation whose objective is to demonstrate a concept, the results represent a baseline of what can be accomplished, and implementations for use in production systems and optimized to improve recovery effectiveness and reduce downtime should be able to exceed the obtained results. Moreover, we suspect that systems with higher amount of consolidated VMs (public cloud providers average 10 VMs on the same node [4]) and higher CPU consolidation ratios (*i.e.*, more VMs running over less CPUs) will perform better and benefit more from the technique.

Fair comparisons against existing fault tolerance techniques for virtualized systems are difficult to perform due to

TABLE 6. Comparison of techniques for fault tolerance in virtualized systems.

Technique	Fault coverage	Recovery efficiency	Downtime	Runtime overhead	Nested virtual-ization?	Requires hardware redundancy?	Is publicly available?
Remus [9]	Permanent HW faults	100%	<100 ms	30-100%		✓	✓
ReHype [23], [56]	Transient HW faults & transient SW faults	60-70%	715-2895 ms	-			
HyperFresh [32]	Transient SW faults	-	100-1000 ms	-	✓		
Romulus	Transient HW faults & SW faults	48%-56%	∼17 000 ms	<1%	✓		✓

the lack of publicly available code and the fact that different setups were used for the evaluations found in the respective papers, including different amount of VMs, different memory size of each VM, different workloads and different fault injection techniques or fault models. For example, the fault model used for the evaluation can have a significant impact on the reported performance (*e.g.*, the recovery percentage), as fault models that cause more abrupt failures are less likely to silently corrupt the system and thus their failures are more easily tolerated. In the case of Romulus, the fault model accurately emulates software faults, including those that linger in the system during many seconds before recovery takes place.

Nevertheless, Table 6 presents an attempt at comparing Romulus against its three most direct competitors. It should be noted that the downtime values presented for Romulus refer to average values obtained for a VM with 4 Gb of memory size and that we considered recovery to be successful whenever 3 VMs (out of the 4 VMs) are recovered. These choices were taken as to more closely reflect the experimental setups used in the papers of the other techniques. Finally, assigning too much importance to performance comparisons may be counter-productive, since the values described in literature reflect the performance of the evaluated implementations and often there is a range of optimizations that can be introduced to further improve the results, but which are outside the scope of an academic work.

Our technique is the only, as far as we know, that has been proven capable of tolerating both failures due to transient hardware faults and software faults. It is also one of the few techniques that have an implementation published under an open-source license. In our opinion, the biggest threat to the adoption of this technique in production systems is not the less-than-perfect VM recovery percentage, but rather the runtime performance overhead, and respective cost, that is associated to nested virtualization. Nevertheless, if we compare the overhead of our technique against other alternatives, our techniques possesses one strong point: no external hardware is required. For example, Remus [9], which provides tolerance against permanent hardware faults, requires a secondary host and incurs a performance overhead ranging between 30%-100% depending on the configuration and workload. Another point to consider is that nested virtualization is gaining adoption in cloud computing, driven by user demand, as well as being a supporting technology for other techniques found in the literature, such as HyperFresh [32] and TinyChecker [36]. As such, adding Romulus to a system that already uses nested virtualization carries almost no overhead.

In terms of limitations, the technique only supports VMs that use hardware-assisted virtualization, systems that have virtualization extensions and requires a microvisor that supports nested virtualization and VMI. The proof-of-concept, due to its nature, has several limitations (which are detailed in Section IV-4) but these are technical limitations and in no way correspond to limitations of the technique. The experimental evaluation is limited in terms of generalizability. Cloud computing supports many different workloads, but only one (a Solr service) was evaluated. Another limitation is the number of VMs running in the system, which is lower than the average found in public clouds. These limitations are not expected to invalidate the presented results, but future work will try to comprehend how different setups affect the performance of the technique.

VIII. CONCLUSION

Hypervisor failures are a threat to the availability of cloud computing systems because they may propagate to the dozens of VMs that are usually consolidated on the same physical hardware and cause service disruption. Thus far the assumption was that whenever the hypervisor failed, all VMs were lost. On the contrary, this paper empirically verifies the hypothesis that a significant percentage of VMs remain correct after hypervisor failure and could be resumed in another hypervisor.

Based on this observation, this paper presents a technique for tolerating hypervisor failures without requiring spare redundant hardware nor modifications to the virtual machines, which means that legacy applications executing in the cloud are natively supported. It performs efficient migration of the VM state from the failed hypervisor to a co-located hypervisor with the purpose of continuing VM execution after failure. This process has inherent challenges regarding how to transparently extract and resume a VM in a different co-located hypervisor, all of which were solved in a proof-of-concept implementation that demonstrates the viability of Romulus.

Experimental results using the proof-of-concept have shown that it is capable of recovering an average of 41-46% of the VMs in the system after a hypervisor failure, including those caused by transient hardware faults and software faults, while incurring a VM downtime in the range of a few seconds, which accounts in large part for the time taken to migrate the VM state between hypervisors. If the system over which Romulus is applied already uses nested virtualization, the overhead of the technique is almost non-existent, otherwise the introduction of nested virtualization, which is a requirement of Romulus, will bring a considerable amount of overhead. Nevertheless, the overhead is still in line or better than that of other high-availability techniques and future developments to nested virtualization should lower the overhead.

As future work, development of the proof-of-concept implementation will continue to introduce optimizations and remove limitations, such as adding support for hyper-paging and multi-CPU VMs, and experimental evaluations using setups that have higher VM consolidation and use other workloads will be conducted. Furthermore, we will consider relaxing some of the current constraints, such as providing a transparent technique that can be used with legacy systems, and investigate using instrumentation at the VM and application-level to significantly improve recovery effectiveness.

ACKNOWLEDGMENT

The authors, the VALU3SConsortium, and the ECSEL JU are not responsible for the use which might be made of the information contained in here.

REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010, doi: 10.1145/1721654.1721672.

[2] S. Bouchenak, G. Chockler, H. Chockler, G. Gheorghe, N. Santos, and A. Shraer, "Verifying cloud services: Present and future," *ACM SIGOPS Operating Syst. Rev.*, vol. 47, no. 2, pp. 6–19, Jul. 2013, doi: 10.1145/2506164.2506167.

[3] Q.-H. Zhu, H. Tang, J.-J. Huang, and Y. Hou, "Task scheduling for multi-cloud computing subject to security and reliability constraints," *IEEE/CAA J. Automatica Sinica*, vol. 8, no. 4, pp. 848–865, Apr. 2021.

[4] R. Birke, A. Podzimek, L. Y. Chen, and E. Smirni, "State-of-the-practice in data center virtualization: Toward a better understanding of VM usage," in *Proc. 43rd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2013, pp. 1–12.

[5] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Depend. Sec. Comput.*, vol. 1, no. 1, pp. 11–33, Jan./Mar. 2004.

[6] F. Cerveira, R. Barbosa, H. Madeira, and F. Araujo, "The effects of soft errors and mitigation strategies for virtualization servers," *IEEE Trans. Cloud Comput.*, early access, Feb. 11, 2020, doi: 10.1109/TCC.2020.2973146.

[7] X. Xu and H. H. Huang, "On soft error reliability of virtualization infrastructure," *IEEE Trans. Comput.*, vol. 65, no. 12, pp. 3727–3739, Dec. 2016.

[8] B. Fang, Q. Guan, N. Debardeleben, K. Pattabiraman, and M. Ripeanu, "LetGo: A lightweight continuous framework for HPC applications under failures," in *Proc. 26th Int. Symp. High-Perform. Parallel Distrib. Comput.*, Jun. 2017, pp. 117–130, doi: 10.1145/3078597.3078609.

[9] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proc. 5th USENIX Symp. Netw. Syst. Design Implement.*, San Francisco, CA, USA, 2008, pp. 161–174.

[10] P. Mell and T. Grance, "The NIST definition of cloud computing," NIST, Gaithersburg, MD, USA, Tech. Rep. SP 800-145, 2011.

[11] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, Feb. 2014, doi: 10.1145/2644865.2541941.

[12] P. T. Endo, M. Rodrigues, G. E. Gonçalves, J. Kelner, D. H. Sadok, and C. Curescu, "High availability in clouds: Systematic review and research challenges," *J. Cloud Comput.*, vol. 5, no. 1, p. 16, Oct. 2016, doi: 10.1186/s13677-016-0066-8.

[13] X. Xu and H. H. Huang, "Understanding reliability implication of hardware error in virtualization infrastructure," in *Proc. 10th Workshop Hot Topics Syst. Dependability (HotDep)*, 2014, pp. 1–6.

[14] H. Takabi, J. B. D. Joshi, and G.-J. Ahn, "Security and privacy challenges in cloud computing environments," *IEEE Secur. Privacy Mag.*, vol. 8, no. 6, pp. 24–31, Nov. 2010.

[15] K. Ren, C. Wang, and Q. Wang, "Security challenges for the public cloud," *IEEE Internet Comput.*, vol. 16, no. 1, pp. 69–73, Jan. 2012.

[16] J. W. McPherson, "Reliability challenges for 45 nm and beyond," in *Proc. 43rd Annu. Design Automat. Conf.*, New York, NY, USA, Jul. 2006, pp. 176–181.

[17] S. Borkar, "Design perspectives on 22 nm CMOS and beyond," in *Proc. 46th Annu. Design Automat. Conf. (DAC)*, New York, NY, USA, 2009, pp. 93–94.

[18] L. A. Barroso, J. Clidaras, and U. Hlzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 2nd ed. San Mateo, CA, USA: Morgan & Claypool Publishers, 2013.

[19] G. Magklis, G. Semeraro, D. H. Albonesi, S. G. Dropsho, S. Dwarkadas, and M. L. Scott, "Dynamic frequency and voltage scaling for a multiple-clock-domain microprocessor," *IEEE Micro*, vol. 23, no. 6, pp. 62–68, Nov./Dec. 2003.

[20] A. Dixit and A. Wood, "The impact of new technology on soft error rates," in *Proc. Int. Rel. Phys. Symp.*, Apr. 2011, pp. 5B.4.1–5B.4.7, doi: 10.1109/IRPS.2011.5784522.

[21] V. Chandra and R. Aitken, "Impact of technology and voltage scaling on the soft error susceptibility in nanoscale CMOS," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Syst.*, Oct. 2008, pp. 114–122.

[22] D. Cotroneo, L. De Simone, P. Liguori, R. Natella, and N. Bidokhti, "How bad can a bug get? An empirical analysis of software failures in the OpenStack cloud computing platform," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, New York, NY, USA, Aug. 2019, pp. 200–211, doi: 10.1145/3338906.3338916.

[23] M. Le and Y. Tamir, "ReHype: Enabling VM survival across hypervisor failures," in *Proc. 7th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ. (VEE)*, New York, NY, USA, 2011, pp. 63–74, doi: 10.1145/1952682.1952692.

[24] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, "Steward: Scaling Byzantine fault-tolerant replication to wide area networks," *IEEE Trans. Dependable Secure Comput.*, vol. 7, no. 1, pp. 80–93, Jan./Mar. 2010.

[25] M. Eischer and T. Distler, "Resilient cloud-based replication with low latency," 2020, *arXiv:2009.10043*. [Online]. Available: https://arxiv.org/abs/2009.10043

[26] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan, "Colo: Coarse-grained lock-stepping virtual machines for non-stop service," in *Proc. 4th Annu. Symp. Cloud Comput.*, New York, NY, USA, 2013, pp. 1–16, doi: 10.1145/2523616.2523630.

[27] C. Wang, X. Chen, W. Jia, B. Li, H. Qiu, S. Zhao, and H. Cui, "PLOVER: Fast, multi-core scalable virtual machine fault-tolerance," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Renton, WA, USA, Apr. 2018, pp. 483–489. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/wang

[28] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot—A technique for cheap recovery," in *Proc. 6th Conf. Symp. Operating Syst. Design Implement.*, Renton, WA, USA, vol. 6, 2004, p. 3.

[29] M. Le and Y. Tamir, "Resilient virtualized systems using ReHype," Dept. Comput. Sci., UCLA, Los Angeles, CA, USA, Tech. Rep. #140019, 2014.

[30] D. Zhou and Y. Tamir, "Fast hypervisor recovery without reboot," in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2018, pp. 115–126.

[31] K. Kourai and S. Chiba, "A fast rejuvenation technique for server consolidation with virtual machines," in *Proc. 37th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2007, pp. 245–255.

[32] H. Bagdi, R. Kugve, and K. Gopalan, "HyperFresh: Live refresh of hypervisors using nested virtualization," in *Proc. 8th Asia–Pacific Workshop Syst.*, New York, NY, USA, Sep. 2017, doi: 10.1145/3124680.3124734.

[33] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. HarEl, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation of nested virtualization," in *Proc. 9th USENIX Conf. Operating Syst. Design Implement.*, Renton, WA, USA, 2010, pp. 423–436.

[34] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *25th Int. Symp. Fault-Tolerant Comput. Dig. Papers*, 1995, pp. 381–390.

[35] X. Xu and H. H. Huang, "DualVisor: Redundant hypervisor execution for achieving hardware error resilience in datacenters," in *Proc. 15th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, May 2015, pp. 485–494.

[36] C. Tan, Y. Xia, H. Chen, and B. Zang, "Tinychecker: Transparent protection of VMs against hypervisor failures with nested virtualization," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw. Workshops (DSN)*, Jun. 2012, pp. 1–6.

[37] H. Xiong, Z. Liu, W. Xu, and S. Jiao, "Libvmi: A library for bridging the semantic gap between guest OS and VMM," in *Proc. IEEE 12th Int. Conf. Comput. Inf. Technol.*, Oct. 2012, pp. 549–556.

[38] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "Sel4: Formal verification of an OS kernel," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Princ.*, New York, NY, USA, 2009, pp. 207–220, doi: 10.1145/1629575.1629596.

[39] R. C. Bhushan and D. K. Yadav, "Verification of virtual machine architecture in a hypervisor through model checking," *Procedia Comput. Sci.*, vol. 167, pp. 67–74, Jan. 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050920306487

[40] S. Li, X. Li, R. Gu, J. Nieh, and J. Hui, "A secure and formally verified Linux KVM hypervisor," in *Proc. IEEE Symp. Secur. Privacy (SP)*, Los Alamitos, CA, USA, May 2021, pp. 839–856. [Online]. Available: https://www.computer.org/csdl/proceedings-article/sp/2021/893400a839/1t0x8ICrxwQ, doi: 10.1109/SP40001.2021.00049.

[41] J. Pfoh, C. Schneider, and C. Eckert, "A formal model for virtual machine introspection," in *Proc. 1st ACM Workshop Virtual Mach. Secur.*, 2009, pp. 1–10, doi: 10.1145/1655148.1655150.

[42] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003, doi: 10.1145/1165389.945462.

[43] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corp., Santa Clara, CA, USA, Sep. 2016.

[44] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, "Rfc3010: Nfs version 4 protocol," Internet Soc., Reston, VA, USA, Tech. Rep. RFC 3010, 2000.

[45] Q. Feng, J. Han, Y. Gao, and D. Meng, "Magicube: High reliability and low redundancy storage architecture for cloud computing," in *Proc. IEEE 7th Int. Conf. Netw., Architecture, Storage*, Jun. 2012, pp. 89–93.

[46] D. Smiley, E. Pugh, K. Parisa, and M. Mitchell, *Apache Solr Enterprise Search Server*. Birmingham, U.K.: Packt Publishing, 2015.

[47] Wikimedia. *Enwiki Dump Progress on 20200301*. Accessed: Apr. 20, 2020. [Online]. Available: https://dumps.wikimedia.org/enwiki/20200301/

[48] M. Carvalho, W. Cirne, F. Brasileiro, and J. Wilkes, "Long-term slos for reclaimed cloud computing resources," in *Proc. ACM Symp. Cloud Comput.*, New York, NY, USA, 2014, pp. 1–13, doi: 10.1145/2670979.2670999.

[49] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Towards understanding heterogeneous clouds at scale: Google trace analysis," Intel Sci. Technol. Center Cloud Comput., Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. ISTC-CC-TR-12-101, 2012, vol. 84.

[50] H. Liu, "A measurement study of server utilization in public clouds," in *Proc. IEEE 9th Int. Conf. Dependable, Autonomic Secure Comput.*, Dec. 2011, pp. 435–442.

[51] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, "One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors," in *Proc. 47th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2017, pp. 97–108.

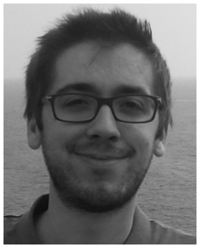
[52] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 849–867, Nov. 2006.

[53] E. V. D. Kouwe, C. Giuffriday, R. Ghituletez, and A. S. Tanenbaum, "A methodology to efficiently compare operating system stability," in *Proc. IEEE 16th Int. Symp. High Assurance Syst. Eng.*, Jan. 2015, pp. 93–100.

[54] D. Cotroneo, L. De Simone, P. Liguori, and R. Natella, "ProFIPy: Programmable software fault injection as-a-service," in *Proc. 50th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Los Alamitos, CA, USA, Jun. 2020, pp. 364–372, doi: 10.1109/dsn48063.2020.00052.

[55] R. Barbosa, F. Cerveira, L. Gonçalo, and H. Madeira, "Emulating representative software vulnerabilities using field data," *Computing*, vol. 101, no. 2, pp. 119–138, Feb. 2019, doi: 10.1007/s00607-018-0657-y.

[56] M. Le and Y. Tamir, "Resilient virtualized systems using ReHype," 2021, *arXiv:2101.09282*. [Online]. Available: https://arxiv.org/abs/2101.09282



FREDERICO CERVEIRA (Member, IEEE) is currently pursuing the Ph.D. degree with the University of Coimbra, Portugal. His Ph.D. topic deals with the evaluation and improvement of the dependability of cloud computing systems in the presence of hardware and software faults.

RAUL BARBOSA received the Ph.D. degree in computer engineering from the Chalmers University of Technology. He was an Adjunct Associate Teaching Professor with the Institute for Software Research, Carnegie Mellon University. He is currently an Assistant Professor with the University of Coimbra (UC). He collaborated with UC. He was the principal investigator at UC in diverse research projects. His research interests include reliable software and distributed systems, including principles for designing and evaluating computer systems that must ensure safety and availability. These topics are systematically addressed using formal approaches, such as model checking and experimental approaches, such as fault injection.

HENRIQUE MADEIRA is currently a Full Professor with the University of Coimbra, Coimbra, Portugal, where he has been involved in the research on dependable computing, since 1988. His research interests include experimental dependability evaluation, dependability and security benchmarking, fault injection, and error detection mechanisms.