Fine-Grained Fault Tolerance For Resilient pVM-based Virtual Machine Monitors

Djob Mvondo, Alain Tchana, Renaud Lachaize, Daniel Hagimont, Noel de Palma

Fine-Grained Fault Tolerance For Resilient pVM-based Virtual Machine Monitors

Djob Mvondo* Alain Tchana† Renaud Lachaize* Daniel Hagimont‡ Noël De Palma*

* *Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG* Grenoble, France

† *ENS Lyon, LIP* Lyon, France

‡ *University of Toulouse, IRIT* Toulouse, France

***Abstract*—Virtual machine monitors (VMMs) play a crucial role in the software stack of cloud computing platforms: their design and implementation have a major impact on performance, security and fault tolerance. In this paper, we focus on the latter aspect (fault tolerance), which has received less attention, although it is now a significant concern. Our work aims at improving the resilience of the "pVM-based" VMMs, a popular design pattern for virtualization platforms. In such a design, the VMM is split into two main components: a bare-metal hypervisor and a privileged guest virtual machine (pVM). We highlight that the pVM is the least robust component and that the existing fault-tolerance approaches provide limited resilience guarantees or prohibitive overheads. We present three design principles (*disaggregation*, *specialization*, and *pro-activity*), as well as optimized implementation techniques for building a resilient pVM without sacrificing end-user application performance. We validate our contribution on the mainstream Xen platform.**

I. Introduction

Virtualization is a major pillar of cloud computing infrastructures because it enables more flexible management and higher utilization of server physical resources. To this end, a virtual machine monitor (VMM) abstracts away a physical machine into a set of isolated guest virtual machines (VMs). This consolidation advantage comes with the risk of centralization. The crash of an OS instance in a bare-metal data center only impacts applications deployed on the corresponding physical server, whereas the crash of a VMM in a virtualized data center has a much larger "blast radius", resulting in the unavailability of the applications hosted in all the guest VMs managed by that VMM instance. Besides, given that the code size and the feature set of VMMs have grown over the years, the occurrences of such critical bugs are becoming more likely. For instance, the number of Xen source code lines has tripled from the first version 1.0 (in 2003) to the current version (4.12.1), as shown in Table I[1].

In the present paper, we highlight the fault-tolerance (FT) limitations of existing VMMs and propose techniques to mitigate them. More precisely, we focus on one of the most popular VMM software architectures, hereafter named *"pVM-based VMMs"*. In this architecture, which has some similarities with a microkernel OS design, the VMM is made of two components: the *hypervisor* and a *privileged VM* (*pVM*). The hypervisor is the low-level component that is mostly in charge of initializing the hardware and acting as a data plane, i.e., providing the logic needed to virtualize the underlying hardware platform, except I/O devices. The pVM acts as a control plane for the hypervisor, through a specific interface, and is involved in all VM management operations (creation, startup, suspension, migration ...). It also hosts I/O device drivers that are involved in all I/O operations performed by *user VMs* (i.e., regular VMs) on para-virtual devices. The pVM is typically based on a standard guest OS (e.g., Linux) hosting a set of control-plane daemons. This pVM-based design is popular and used in production-grade, mainstream virtualization platforms (for example, Xen, Microsoft Hyper-V and some versions of VMware ESX) for several important reasons, including the following ones: (i) it simplifies the development, debugging and customization of the control plane [1], (ii) it provides isolation boundaries to contain the impact of faults within the control plane or the I/O path [2], (iii) it offers flexibility for the choice of the OS hosting the control plane (which matters for considerations like code footprint, security features, and available drivers for physical devices) [3], (iv) it provides a data plane with a smaller attack surface than a full-blown operating system like Linux.

In such a pVM-based VMM design, the crash of the pVM leads to the following severe consequences: (1) the inability to connect to the physical server, (2) the inability to manage user VMs, and (3) the interruption of the networked applications running in user VMs. Besides, the faults within the pVM are more frequent than in the hypervisor itself because the code base of the former is very large (see Table I), thus likely to

	Xen Hypervisor	**Linux-based pVM**	$\frac{Linux}{Xen}$
#LOC in 2003	187,823	3.72 Million	19.81
#LOC in 2019	583,237	18.5 Million	31.76
$\frac{\#LOC in 2019}{\#LOC in 2003}$	3.11	4.98	1.6

TABLE I: Evolution of source code size for the Xen hypervisor and a Linux-based pVM. LOC stands for "Lines of Code".

We thank our shepherd, Elias Duarte, and the anonymous reviewers for their insightful comments. This work was funded by the "ScaleVisor" project of Agence Nationale de la Recherche, number ANR-18-CE25-0016, the "Studio virtuel" project of BPI and ERDF/FEDER, grant agreement number 16.010402.01, the "HYDDA" project of BPI Grant, and the "IDEX IRS" (COMUE UGA grant).

[1]For Linux, we only consider `x86` and `arm` in the `arch` folder. Also, only `net` and `block` are considered in `drivers` folder (other folders are not relevant for server machines).

be more bug-prone than the latter.

While the reliability of the pVM is of primary and growing importance, it has received relatively little attention from the research community. In TFD-Xen, Jo et al. [4] focused on the reliability of the network drivers inside the pVM of a Xen-based system. However, this work does not take into account the critical dependencies of these drivers with other pVM services (see §II). In the Xoar project, Colp et al. [5] proposed to periodically refresh each service of the Xen pVM using a micro-reboot mechanism. This approach, which was initially designed by its authors as a defense mechanism against (stepping-stone) security attacks, can also improve the resilience of a system against faults, by combining preventive rejuvenation and automatic restart of software components. However, periodic refreshes incur unacceptable performance degradation for I/O-sensitive user workloads. For example, on the TailBench benchmark suite [6], we observed a major degradation of the 95th-percentile latencies, up to 1300x (see §II-B). Due to all these limitations, the current solution adopted by data center operators is full server replication (VMM and VMs) at the physical level, like, for example, in VMware vSphere Fault Tolerance [7]. The main limitation of this approach is the fact that it doubles the number of servers in the data center.

In this paper, we assume that the hypervisor is reliable (e.g., thanks to state-of-the-art techniques [8]–[10]) and we propose a holistic and efficient design to improve *in-place* the resilience of a pVM against crashes and data corruption. We choose the Xen VMM [1], [11] as a case study for our prototype implementation, given that this is the most popular pVM-based virtualization platform[2]. In Xen's jargon, the pVM is called *"dom0"* (or *"Domain0"*). Our approach is built following three principles. The first principle is *disaggregation* (borrowed from Xoar [5]), meaning that each pVM service is launched in an isolated unikernel [12], thus avoiding the single point of failure nature of the vanilla pVM design. The second principle is *specialization*, meaning that each unikernel embeds a FT solution that is specifically chosen for the pVM service that it hosts. The third principle is *pro-activity*, meaning that each FT solution implements an active feedback loop to quickly detect and repair faults. The latter two principles are in opposition to the Xoar design, which systematically/unconditionally applies the same FT approach (refresh) to all the pVM components.

In respect to the *disaggregation* principle, we organize Xen's dom0 in four unikernel (uk) types namely *XenStore_uk*, *net_uk*, *disk_uk*, and *tool_uk*. XenStore_uk hosts XenStore, which is a database with a hierarchical namespace storing VM configurations and state information. net_uk hosts both the real and the para-virtualized network drivers. Its FT solution is based on the shadow driver approach [13]. disk_uk is similar to net_uk for storage devices, and tool_uk hosts VM management tools. XenStore is the only component that is subject to data corruption since it is the only one that is stateful. According to the *specialization* and the *pro-activity* principles, the FT solution of each component is as follows: XenStore_uk is replicated (for handling crashes) and it implements a sanity check solution for data corruption detection; net_uk and disk_uk implement a shadow driver FT approach; tool_uk redesigns the VM migration logic for improved resilience. Our solution also includes a global feedback loop implemented inside the hypervisor (which is assumed to be resilient) for managing cascading failures and total dom0 crashes. Cascading failures are related to the relationships between dom0 services. For instance, the failure of XenStore generally causes the failure of all the other components.

In summary, the paper makes the following three main contributions. First, we present for the first time a holistic FT solution for the pVM. Second, we implement a functioning prototype in Xen. The source code of our prototype is publicly available[3]. Third, we demonstrate the effectiveness of our solution. To this end, we first evaluate the FT solution of each pVM component individually. Then, we evaluate the global solution while injecting faults on several components at the same time. The evaluation results show that the impact of our solution on user VMs when they run performance-critical applications such as those from the TailBench suite is acceptable in comparison to state-of-the-art solutions (Xoar [5] and TFD-Xen [4]). For instance, we achieve a 12.7% increase for 95th-percentile tail latencies; in comparison, the increase caused by Xoar is of 12999%.

The rest of the article is organized as follows. Section II presents the background and the motivations. Section III presents the general overview of our solution and the fault model that we consider. Section IV presents the implementation of our solution for each service of Xen's pVM. Section V presents the evaluation results. Section VI discusses the related works. Section VII concludes the paper.

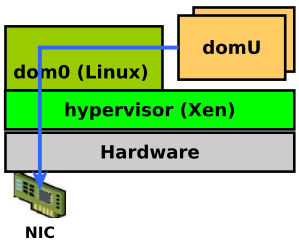

Fig. 1: Overall architecture of a Xen-based virtualization platform. The dom0 VM corresponds to the pVM.

II. Background and Motivation

A. Xen virtualization platform

Figure 1 presents the architecture of a physical server virualized with Xen. It comprises three component types: *domUs*

[2]Xen is used by major hyperscale cloud providers, such as AWS, Tencent, Alibaba Cloud, Oracle Cloud, and IBM Cloud. Xen is also supported by most software stacks used in private clouds of various scales like Citrix XenServer, Nutanix Acropolis, OpenStack and Apache Cloudstack.

[3]https://github.com/r-vmm/R-VMM

	VM manangement operations (impact the cloud provider)				Application operations (impact cloud users)		
	Start	Stop	Migrate	Update	Net I/O	Disk I/O	CPU/ Mem
Tools	A	A	A	A			
Net					A	S	
Disk						A	S
XS	A	A	A	A	S	S	S

TABLE II: Impact of the failure of the different dom0 services (`xl` tools, network/disk drivers, XenStore) on the VM management operations and on the applications (in user VMs). An *"A"* mark indicates that the failure *always* leads to unavailability while a *"S"* mark denotes correlated failures that occur *only in specific situations.*

(user VMs), Xen (the hypervisor), and *dom0* (the pVM). domUs are VMs booked and owned by datacenter users. The combination of Xen and dom0 forms the VMM. Xen runs directly atop the hardware, and is in charge of hardware initialization, resource allocation (except for I/O devices) and isolation between VMs. Besides, dom0 is a Linux system that hosts an important portion of the local virtualization system services, namely (i) the domU life-cycle administration tool (`xl`), (ii) XenStore, and (iii) I/O device drivers.

The `xl` tool stack [14] provides domU startup, shutdown, migration, checkpointing and dynamic resource adjustment (e.g., CPU hotplug). XenStore is a daemon implementing a metadata storage service shared between VMs, device drivers and Xen. It is meant for configuration and status information rather than for large data transfers. Each domain gets its own path in the store, which is somewhat similar in spirit to the Linux `procfs` subsystem. When values are changed in the store, the appropriate components are notified. Concerning I/0 devices, dom0 hosts their drivers and implements their multiplexing, as follows. Along with I/O drivers, dom0 embeds proxies (called *backend* drivers) that relay incoming events from the physical driver to a domU and outgoing requests from a domU to the physical driver. Each domU runs a pseudo-driver (called *frontend*) allowing to send/receive requests to/from the domU-assigned backend.

B. Motivations

The architecture presented in the previous subsection includes two points of failure: the Xen hypervisor and the dom0 pVM. This paper focuses on dom0 fault tolerance (FT). Table II summarizes the negative impact of the failure of dom0 with respect to each service that it provides. We can see that both cloud management operations (VM start, stop, migrate, update) and end user applications can be impacted by a dom0 failure. Concerning the former, they can no longer be invoked in case of dom0 failure. Regarding user applications, those which involve I/O devices become unreachable in case of dom0 failure. The table also shows that XenStore (XS) is the most critical dom0 service because its failure impacts all other services as well as user applications.

Failures within dom0 are likely to occur since it is based on Linux, whose code is known to contain bugs due to its monolithic design, large TCB (trusted computing base) and ever-increasing feature set. We analyzed *xen.markmail.org*, a Web site that aggregates messages from fourteen Xen related mailing lists since October 2003. At the time of writing this paper, we found 243 distinct message subjects including the terms *crash*, *hang*, *freeze*, *oops* and *panic*[4]. After manual inspection of each of the 243 messages, we discarded 82 of them because they were not talking about faults. 57% of the remaining messages were related to failures of dom0 components and 43% to the hypervisor. By zooming on dom0 faults, we observed that 66% were related to device drivers, 26% to the tool stack, and 8% to XenStore. From this analysis, two conclusions can be drawn: (1) cloud sysadmins report dom0 failures; (2) such failures are linked to all dom0 services.

To the best of our knowledge, the only existing exhaustive solution against dom0 failures (without resorting to physical server replication) is the one proposed in the Xoar project [5]. This approach was initially designed against security attacks, but also provides fault tolerance benefits. It has two main aspects. First, dom0 is disaggregated in several unikernels in order to confine each service failure. Second, each service is periodically restarted ("refreshed") using a fresh binary. The critical parameter in such an approach is the refresh frequency. On the one hand, if it is large (tens of seconds), then components that are impacted by a dom0 failure will experience this failure for a long time. On the other hand, if the refresh period is too short (e.g., one second) then failures are handled relatively quickly, but at the expense of significant performance degradation for the user applications. This dilemma has been partially acknowledged by the authors of Xoar in their paper [5]: in the case of a short refresh period (1 second), they measured a 3.5 degradation ratio for the throughput and latency of a Web server benchmark. We also assessed this limitation by running latency-sensitive applications from the TailBench suite [6] in a domU while varying the refresh period of its assigned network backend unikernel (the details of the testbed are provided in §V). Figure 2 reports for each benchmark the ratio of the mean and (95th and 99th percentile) tail latencies over the execution of the same benchmark without refresh. We can see that self refresh can incur a 5x-2000x degradation for the mean latency, 5x-1300x for the 95th percentile, and 5x-1200x for the 99th percentile. We also notice that the degradation remains significant even with a large refresh period (60 seconds). These values are too high, unacceptable for cloud users. This strengthens the need for a better approach for dom0 FT.

III. General overview

This section presents the basic idea behind our dom0 FT solution and the general fault model that we target.

A. Basic idea

Our solution, named PpVMM (Phoenix pVM-based VMM), is based on three main principles. The first principle is *disag-*

[4]We used the search string *"crash hang freeze oops panic -type=checkins"*. The option *"type=checkins"* excludes commit messages.

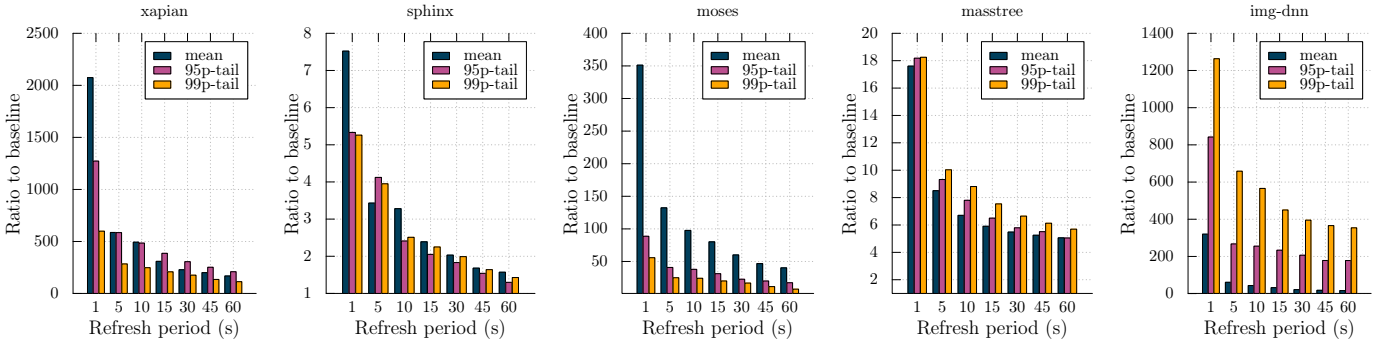

Fig. 2: Mean and tail latencies for TailBench applications when self-refresh is enabled for the (disaggregated) pVM components. The results (lower is better) are normalized w.r.t. a baseline without self-refresh for the same metrics.

gregation (borrowed from Xoar [5]), meaning that each dom0 service is launched in an isolated unikernel, thus avoiding the single point of failure nature of the vanilla centralized dom0 design. The second principle is *specialization*, meaning that each unikernel embeds a FT solution that is specifically chosen for the dom0 service that it hosts. The third principle is *proactivity*, meaning that each FT solution implements an active feedback loop to quickly detect and repair faults.

Driven by these three principles, we propose the general architecture of our FT dom0 in Figure 3. The latter is interpreted as follows. dom0 is disaggregated in four unikernels namely XenStore_uk, net_uk, disk_uk, and tool_uk. Some unikernels (e.g., device driver unikernels) are made of sub-components. We equip each unikernel and each sub-component with a feedback loop that includes fault detection (probes) and repair (actuators) agents. Both probes and actuators are implemented outside the target component.We associate our (local) dom0 FT solution with the (distributed) data center management system (e.g., OpenStack Nova) because the repair of some failures may require a global point of view. For instance, the failure of a VM creation request due to a lack of resources on the server may require to retry the request on another server. This decision can only be taken by the data center management system. Therefore, each time a failure occurs, a first step repair solution provided by our system is performed locally on the actual machine. Then, if necessary, a notification is sent to the data center management system.

A global feedback loop coordinates per-component feedback loops in order to handle concurrent failures. The latter requires a certain repair order. For instance, the failure of XenStore_uk is likely to cause the failure of other unikernels since XenStore acts as a storage backend for their configuration metadata. Therefore, XenStore_uk repair should be launched first, before the repair of the other unikernels. We implement the global feedback loop inside the hypervisor, which is the only component that we assume to be safe.

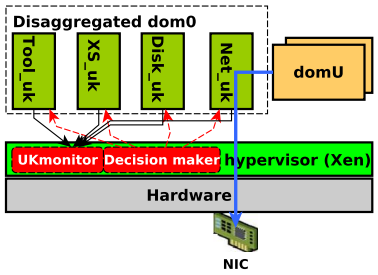

Fig. 3: Overall architecture of our FT pVM design.

B. General fault model

This section presents in a generic way the pVM (dom0) fault model that we target. Additional details are given in the next sections for each component. In the disaggregated architecture on which we build our FT solution, the dom0 components can be classified into two types: stateful (XenStore_uk) and stateless (net_uk, disk_uk, and tool_uk). We assume that all components may suffer from crash faults and that stateful components can also suffer from data corruption faults. Crash faults may happen in situations in which a component is abruptly terminated (e.g., due to invalid memory access) or hangs (e.g., due to a deadlock/livelock problem). These situations can make a component either unavailable, unreachable, or unresponsive when solicited. For stateful components, we are also interested in data corruption issues, that may stem from various causes (e.g., an inconsistency introduced by a software crash, a sporadic bug, or hardware "bit rot"). Furthermore, our fault model encompasses situations in which several components are simultaneously in a failed state (either due to correlated/cascading failures) or due to independent issues. Besides, our work assumes that the code and data within the hypervisor component (i.e., *Xen*) are reliable or, more reasonably, that potential reliability issues within the hypervisor are addressed with state-of-the-art fault tolerance techniques such as ReHype [8] (discussed in §VI). Our design

	net_uk	disk_uk	tool_uk	xenstore_uk
# Base LOCs	$193k$	$350k$	$270k$	$8k$
# Lines +	193	87	8	27

TABLE III: Lines of codes added to each unikernel codebase for fault tolerance.

requires small and localized modifications (318 LOCs) to the Xen hypervisor; we believe that they do not introduce significant weaknesses in terms of reliability.

IV. Implementation

To build our disaggregated dom0 architecture, we leverage the unikernels developed by the Xen project (Mini-OS and MirageOS). The motivation for these unikernels in the context of the Xen project is to contain the impact of faults in distinct pVM components. However, our contribution goes beyond the mere disaggregation of the pVM: we explain how to support fine-grained fault detection and recovery for each pVM component. This section presents the implementation details of our fault tolerance (FT) solution for each dom0 unikernel, except (due to lack of space) for disk_uk, which is relatively similar to net_uk. For each unikernel, we first present the specific fault model that we target followed by the FT solution (Table III presents a summary of the code size for each unikernel (existing code + modifications). Finally, the section presents the global feedback loop (which coordinates the recovery of multiple components) and discusses scheduling optimizations.

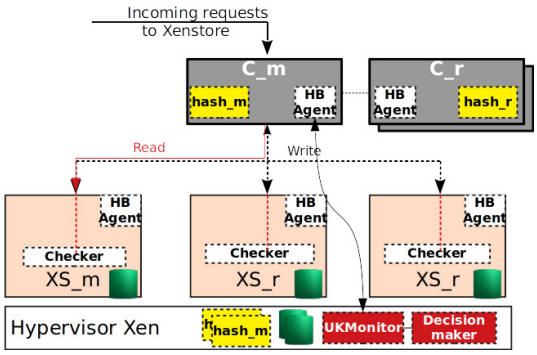

Fig. 4: Replication-based FT solution for XenStore.

A. XenStore_uk FT solution

XenStore is a critical metadata storage service, on which other dom0 services rely. XenStore_uk runs within the MirageOS unikernel [15].

Fault model. We consider two types of faults. The first type is *unavailability*, meaning that XenStore is unable to handle incoming requests, due to bugs (hangs/crashes). The second type is silent *data corruption*; such issues may be caused by bit flips, caused by defective hardware or possibly malicious VMs (e.g., RowHammer attacks [16], [17]).

FT solution. We use *state machine replication* and *sanity checks* to handle unavailability and data corruption respectively. The overall architecture is depicted in Figure 4. Note that the memory footprint of a XenStore database is typically very small (lower than 1MB for 40 VMs).

Unavailability. We organize XenStore into several replicas (e.g., three in our default setup). Each replica runs in a dedicated unikernel based on MirageOS [15]. The set of replicas is managed by a coordinator running in a dedicated unikernel. Notice that the coordinator (noted C_m) is also replicated (the replicas are noted C_r) for FT. C_m chooses a XenStore replica to play the role of the master (noted XS_m). Let us note the other XenStore replicas XS_r. C_m is the XenStore entry point for requests sent by XenStore clients. We enforce this by modifying the `xs_talkv` function of the XenStore client library, used by the other components. C_m forwards read requests only to XS_m, while write requests are broadcast to all replicas.

We implement this *state machine replication* strategy using the *etcd* coordination system [18] deployed in a MirageOS unikernel. We choose etcd because of its well-established robustness, and its relatively lightweight resource requirements (compared to other coordination systems such as ZooKeeper [19]). Also, etcd has built-in support for high availability through strongly-consistent replication based on the Raft consensus algorithm [20]. In the rest of this section we use the term etcd to refer to C_m and the C_r replicas.

We improve etcd to provide both failure detection and repair strategies, as follows. etcd is augmented with a heartbeat (HB) monitor for each XenStore replica. When a replica does not answer to a heartbeat, etcd pro-actively replaces the replica with a fresh version, whose state is obtained from another alive uncorrupted XenStore replica. This recovery process does not interrupt request handling by other replicas. In case of the unavailability of XS_m, etcd elects another master and forwards to it the in-progress requests that were assigned to the crashed master. C_m exchanges heartbeat messages with the hypervisor so that the latter can detect the simultaneous crashing of the former and the C_r replicas. In fact, the failure of one coordinator instance can be handled by the other instances without the intervention of the hypervisor. The latter intervenes only when all instances crash at the same time.

Besides, we have modified the communication mechanism used between etcd and the other components. Instead of leveraging its default communication interface based on the HTTP protocol, we rely on virtual IRQs and shared memory. The motivation is twofold. First, this reduces the communication overheads on the critical path. Second, the utilization of HTTP would involve net_uk in the failure handling path of XenStore, thus adding a dependence of the latter w.r.t. the former. This dependency would make it difficult to handle cascading failures since net_uk already relies on XenStore.

In order to provide a highly available metadata storage service, an alternative design could consist in using the etcd instances as a complete replacement for the XenStore instances. This approach would reduce the number of unikernel instances

and some communication steps between the pVM components. We have actually tried to implement this approach but the achieved performance was significantly poorer: we observed request latencies that were higher by up to several orders of magnitude (microseconds vs. milliseconds). Indeed, XenStore and etcd are datastores with a fairly similar data model but their implementations are optimized for different contexts (local machine interactions versus distributed systems). In addition, the design that we have chosen helps limiting the modifications to be made w.r.t. the implementation of the vanilla pVM components. In particular, this allows benefiting from new features and performance optimizations integrated into the vanilla XenStore codebase, e.g., regarding data branching and transactions.

Data corruption. This type of faults is handled by a *sanity check* approach implemented on all XenStore replicas, as described below. First, we make the following assumption: for a given logical piece of information that is replicated into several physical copies, we assume that there is at most one corrupted copy. Each etcd instance stores a hash of the content of the latest known uncorrupted XenStore database state. Besides, a sanity check agent (called *checker*) runs as a transparent proxy in each XenStore replica. Upon every write request sent to the XenStore service, each `checker` computes a new hash, which is forwarded to the etcd coordinator. If the hashes sent by all the replicas match, then this new value is used to update the hash stored by all the etcd instances. Upon the reception of a read request, the master XenStore replica computes a hash of its current database state and compares it against the hash sent by the coordinator. If they do not match, a distributed recovery protocol is run between the etcd coordinators to determine if the corrupted hash stems from the coordinator or the XenStore master replica. In the former case, the hash of the coordinator is replaced by the correct value. In the latter case, the XenStore replica is considered faulty and the etcd coordinator triggers the above-mentioned recovery process.

Total XenStore failure. In the worse case, all XenStore and/or etcd components can crash at the same time. In our solution, this situation is detected and handled by the hypervisor via the heartbeat mechanism mentioned above. The hypervisor relaunches the impacted component according to a dependency graph (see §IV-D). However, an additional issue is the need to retrieve the state of the XenStore database. In order to tolerate such a scenario without relying on the availability of the disk_uk (to retrieve a persistent copy of the database state), we rely on the hypervisor to store additional copies of the XenStore database and the corresponding hashes. More precisely, the hypervisor hosts an in-memory backup copy for the database and hash stored by each replica, and each replica is in charge of updating its backup copy.

B. net_uk FT solution

Based on the Mini-OS unikernel [21], the net_uk component embeds the NIC driver and, in accordance with the *split driver model*, it proxies incoming and outgoing network I/O requests to/from user VMs. To this end, net_uk also runs a virtual driver called *netback* that interacts with a pseudo NIC driver called *netfront* inside the user VM. The interactions between netback and netfront correspond to a bidirectional producer-consumer pattern and are implemented via a ring buffer of shared memory pages and virtual IRQs. Overall, net_uk can be seen as a composite component encapsulating the NIC driver and the netback.

Fault model. We are interested in mitigating the unavailability of net_uk. The latter can be caused by a crash of the NIC driver, of the netback or of the whole unikernel. We assume that a fault in the NIC driver or the netback does not corrupt the low-level data structures of the kernel. This is a viable assumption as we can run the NIC driver in an isolated environment similar to Nooks [22] or LXDs [23].

FT solution. Our approach aims at detecting failures at two levels: a coarse-grained level when the whole unikernel fails and a fine-grained level for the NIC driver and netback failures. Before presenting the details of our solution, we first provide a brief background on the design of the I/O path, using the reception of a new packet as an example.

Once the NIC reports the arrival of a packet, a hardware interrupt is raised and trapped inside the hypervisor. The latter forwards the interrupt (as a virtual interrupt) to net_uk. The handler of that virtual interrupt is then scheduled inside net_uk. In general, the interrupt handler is organized in two parts namely *top half* (TH) and *bottom half* (BH). The top half masks off interrupt generation on the NIC and generates a *softirq* whose handler is the bottom half. The latter registers a NAPI ("New API") function, aimed at polling for additional incoming network packets. The maximum number of packets that can be pooled using this mechanism is controlled via a *budget* and a *weight* parameter. Upon its completion, the bottom half unmasks interrupt generation by the NIC. Overall, this design allows limiting the overhead of network interrupts.

To handle NIC driver failures, we leverage the *shadow driver* approach introduced by Swift et al. [13]. The latter was proposed for bare-metal systems. We adapt it for a Xen virtualized environment as follows. The original *shadow driver* approach states that each (physical) driver to be made fault tolerant should be associated a with shadow driver, interposed between the former and the kernel. This way, a failure of the target driver can be masked by its shadow driver, which will mimic the former during the recovery period. The shadow driver can work in two modes: passive and active. In passive mode, it simply monitors the flow of incoming and completed requests between the kernel and the target driver. Upon failure of the target driver, the shadow driver switches to the active mode: it triggers the restart of the target driver (and intercepts the calls made to the kernel), and it buffers the incoming requests from the kernel to the target driver (which will be forwarded after the recovery process).

In our specific virtualized context, we do not create a shadow driver for each net_uk component. Instead, we con-

sider an improved version of the netback driver as the shadow driver for both itself and the NIC driver (see Fig. 5.b). In this way, we reduce the number of shadow drivers and, as a consequence, the net_uk code base (complexity). When a bottom half handler is scheduled, a signal is sent to the hypervisor, which records the corresponding timestamp t_o. Once the execution of the bottom half ends, another signal is sent to the hypervisor to notify completion (see Fig. 5.a). If no completion signal is received by the hypervisor after $t_o + t_{max}$, where t_{max} is the estimated bottom half maximum completion time, the hypervisor considers that the NIC driver has failed, and triggers the recovery of the driver (using existing techniques [13]), as shown in Fig. 5.b. The tuning of t_{max} depends on budget and weight values (see above) and is empirically determined. In our testbed, the values used for budget and weight are 300 and 64 respectively, and t_{max} is about $4s$.

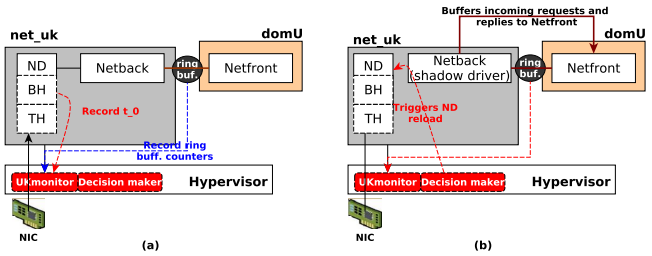

Fig. 5: net_uk, in which the shadow driver (netback) works either in *passive* (a) or in *active* (b) mode. In the former mode (no NIC driver failure), the hypervisor records the bottom half handler (BH) starting timestamp t_o and awaits a completion signal before $t_o + t_{max}$, otherwise triggers NIC driver (ND) reload. In active mode (ND failure has been detected), the netback buffers requests and acks the netfront upon ND recovery.

Regarding the failure of the netback, the hypervisor monitors the shared ring buffer producer and consumer counters between a netback and its corresponding frontend. If the netback's private ring counters remain stuck while the shared ring counters keep evolving, this lag is considered as a hint revealing the failure of the netback. Hence, the netback is reloaded (unregistered then registered). Meanwhile, its frontend's device attribute `otherend->state` value switches to `XenbusStateReconfigured` while the netback undergoes repair. Once the repair is complete, the latter value switches back to `XenbusStateConnected` and proceeds with the exchange of I/O requests with the netback.

Regarding the failure of the entire unikernel, we adopt the same approach as TFD-Xen [4]: the hypervisor monitors the sum of the counters of the shared ring buffer used by all netbacks and their corresponding netfront drivers to detect a lag between the producer and the consumer counter. However, this approach alone cannot detect net_uk hanging when it is not used by any user VM. Therefore, we combine it with a heartbeat mechanism, also controlled by the hypervisor. A reboot of the net_uk VM is triggered when any of the two above-described detection techniques raises an alarm.

C. tool_uk FT solution

The tool_uk unikernel embeds the Xen toolstack for VM administration tasks (creation, migration, etc.). We use XSM (Xen Security Modules [24]) to introduce a new *role* (*tooldom*) which has fewer privileges than the original monolithic dom0 but enough for administrative services. It runs in an enriched version of Mini-OS [21], a very lightweight unikernel, part of the Xen project.

Fault model. We strive to mitigate faults occurring during administrative operations. Apart from live migration (discussed below), the fault tolerance requirements for all the other administration tasks are already fully handled either locally, by the vanilla toolstack implementation, or globally, by the data center management system (e.g. Nova in OpenStack). In these cases, our solution provides nonetheless fast and reliable notifications regarding the failures of the local toolstack. We now describe the specific problem of resilient live migration. During the final phase of a live migration operation for a VM[5], the suspended state of the migrated VM is transferred to the destination host and upon reception on the latter, the VM is resumed. If a fault occurs during that phase, the migration process halts and leaves a corrupted state of the VM on the destination machine and a suspended VM on the sender machine.

FT solution. We consider that a failure has occurred during the migration process if the sender machine does not receive (within a timeout interval) the acknowledgement message from the destination machine, which validates the end of the operation. As other unikernels in our solution, faults resulting in the crash/hang of the entire tool_uk are detected with a heartbeat mechanism and trigger the restart of the tool_uk instance. In both cases (partial or complete failure of the component), the repair operation for the failed migration is quite simple and consists in (*i*) first discarding the state of the suspended VM on the destination machine, (*ii*) destroying the VM on the destination machine, and (*iii*) resuming the original VM instance on the sender machine.

D. Global feedback loop

Our solution includes a global feedback loop for handling concurrent failures of multiple pVM components (and potentially all of them). Such a situation may or may not be due to a cascading failure. To handle such a situation in the most efficient way, the hypervisor embeds a graph thats indicates the dependencies between the different unikernels, which are represented in Figure 6. When a unikernel fails, the hypervisor starts the recovery process only when all unikernels used by the former are known to be healthy and reachable.

[5]Xen adopts a pre-copy iterative strategy for live migration [25].

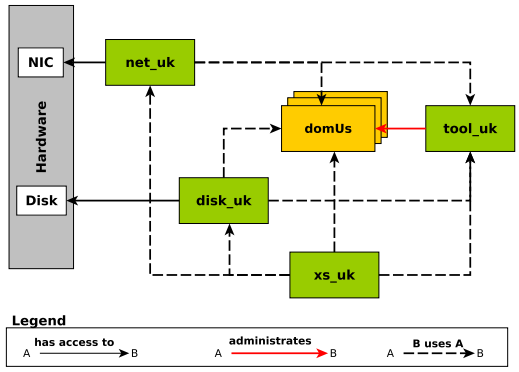

Fig. 6: Relationships between the different components of the disaggregated pVM.

E. Scheduling optimizations

The design that we have described so far, with the disaggregation of the pVM services into independent unikernel VMs and the usage of heartbeats to detect their failures, raises some challenges with respect to CPU scheduling. Indeed, it is non-trivial to ensure that these VMs are appropriately scheduled. On the one hand, due to the number of VMs resulting from the disaggregation, dedicating one (or several) distinct physical CPU core(s) to each unikernel VM would result in significant resource waste (overprovisioning). On the other hand, if such VMs are not scheduled frequently enough, they may not be able to send their heartbeats on time to the hypervisor (leading to false positives, and unneeded repair procedures), or, as a workaround, this may require to set longer timeouts (leading to slow detection of actual failures). In order to overcome the above-described issues, we slightly modify the CPU scheduler of the hypervisor. At creation time, each service VM is marked with a special flag and the hypervisor CPU scheduler guarantees that such VMs are frequently scheduled and sends a ping request to a unikernel VM before switching to it. Each service VM is granted a time slice of 5ms for heartbeat response. As an additional optimization, the scheduling algorithm is modified to skip the allocation of a CPU time slice to a unikernel VM if the latter has recently (in our setup, within the last 15ms) issued an "implicit" heartbeat (for example, in the case of the net_uk VM, a recent and successful interaction with the hypervisor for sending or receiving a packet is a form of implicit heartbeat). This avoids the cost of context switches to a unikernel VM solely for a ping-ack exchange when there are hints that this VM is alive.

V. Evaluation

This section presents the evaluation results of our prototype.

Evaluation methodology and goals. We evaluate both the robustness and the reactivity of our solution in fault situations. We first evaluate each dom0 service FT solution individually, meaning that a single failure (in a single component) is injected at a time in the system. Then, we consider the failure of several services at the same time. For each experiment, we consider a challenging situation in which both dom0 services and user VMs are highly solicited. Crash failures are emulated by killing the target component or unikernel. In order to simulate data corruption (in the case of XenStore_uk), we issue a request that overwrites a path (key-value pair) within the data store.

We are interested in the following metrics: (1) the overhead of our solution on the performance of dom0 services; (2) the overhead of our solution on the performance of user VMs; (3) the failure detection time; (4) the failure recovery time; (5) the impact of failures on dom0 services; (6) the impact of failures on user VMs. The overhead evaluation is performed on fault-free situations. We compare our solution with vanilla Xen 4.12.1 (which provides almost no fault tolerance guarantees against pVM failures), Xoar [5] (periodic refresh), and TFD-Xen [4] (which only handles net_uk failures). For a meaningful comparison, we re-implemented the two previous systems in the (more recent) Xen version that we use for our solution. For Xoar, we use a component refresh period of 1 second, and the different components are refreshed sequentially (not simultaneously) in order to avoid pathologic behaviors.

Benchmarks. User VMs run applications from the TailBench benchmark suite [6]. The latter is composed of 8 latency-sensitive (I/O) applications that span a wide range of latency requirements and domains and a harness that implements a robust and statistically-sound load-testing methodology. It performs enough runs to achieve 95% confidence intervals $\leq 3\%$ on all runs. We use the client-server mode. The client and the server VMs run on distinct physical machines. The server VM is launched on the system under test. We left out 3 applications from the TailBench suite, namely *Shore, Silo* and *Specjbb*. Indeed, the two former are optimized to run on machines with solid state drives (whereas our testbed machine is equipped with hard disk drives), and *Specjbb* cannot run in client-server mode. In addition, we also measure the request throughput sustained by the Apache HTTP server (running in a user VM) with an input workload generated by the AB (ApacheBench) benchmark [26] (using 10,000 requests and a concurrency level of 10).

Testbed. All experiments are carried out on a 48-core PowerEdge R185 machine with AMD Opteron 6344 processors and 64 GB of memory. This is a four-socket NUMA machine, with 2 NUMA nodes per socket, 6 cores and 8 GB memory per NUMA node.

The dom0 components use two dedicated sockets and the user VMs are run on the two other sockets. Providing dedicated resources to the pVM is in line with common practices used in production [27] in order to avoid interference. Besides, we choose to allocate a substantial amount of resources to the dom0 in order to evaluate more clearly the intrinsic overheads of our approach (rather than side effects of potential resource contention). We use Xen 4.10.0 and

the dom0 runs Ubuntu 12.04.5 LTS with Linux kernel 5.0.8. The NIC is a Broadcom Corporation NetXtreme II BCM5709 Gigabit Ethernet interface. The driver is bnx2. The machines are linked using a 1Gb/s Ethernet switch. Unless indicated otherwise, user VMs run Ubuntu 16.04 with Linux Kernel 5.0.8, configured with 16 virtual CPUs (vCPUs) and 16GB of memory. Concerning unikernels composing dom0, each is configured with 1 vCPU and 1 GB of memory (128MB for the XenStore instances). The real memory footprint during our evaluations is ≈500MB for every unikernel (≈100MB for each Xenstore instance). For fault-free runs, compared to vanilla Xen, we achieve 1-3% slowdown for I/O-intensive applications (disk or network). These results are similar to those reported with Xoar (original version [5] and our reimplementation): the intrinsic performance overhead of disaggregation is low.

A. XenStore_uks

Recall that, in the fault model that we consider, XenStore is subject to both unavailability and data corruption faults. XenStore is highly solicited and plays a critical role during VM administration tasks. We use VM creation operations to evaluate XenStore, because this type of operation is one of the most latency-sensitive and also involves Xenstore the most[6].

1) Robustness: We launch a VM creation operation and inject a crash failure into the master XenStore replica (recall that we use a total of 3 XenStore instances) during the phase where XenStore is the most solicited.We repeat the experiment ten times and we report mean values. The observed results are as follows.

We observe that some VM creations fail with both vanilla Xen and Xoar. The latter, after the refresh period, is not able to replay the VM creation request because it has not been recorded. Besides, Xoar takes 1 second to detect the failure. Its recovery time is 22ms (a reboot of XenStore_uk). In contrast, using our solution, all VM creation operations complete successfully. Our solution takes 1.54ms and 5.04ms to detect crashing and data corruption faults respectively. The recovery process for crashing and data corruption is 25.54ms (starting a new Xenstore_uk replica and synchronizing its database). The overall corresponding VM creation time is about 5.349s and 5.353s respectively for the two failure types, compared to 5.346s when no fault is injected.

2) Overhead: We sequentially execute ten VM creation operations (without faults). The mean VM creation time (until the full boot of the VM's kernel) for vanilla Xen, Xoar, and our solution (PpVMM) is respectively 4.445sec, 6.741sec, and 5.346sec. Our solution incurs about 20.27% (≈ $900ms$) overhead. This is due to the fact that a VM creation operation generates mostly write requests (89% of the requests are writes), which require synchronization between all XenStore replicas. Read requests do not require synchronization. Fig. 7 reports mean, 95th- and 99th-percentile latencies for read and write requests, confirming the above analysis. The overhead incurred by our solution is significantly lower than the overhead of Xoar, which is about 51.65% (≈ $2.3s$).

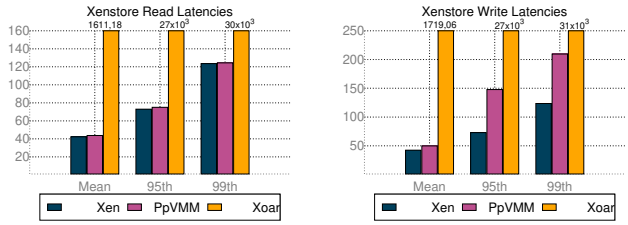

Fig. 7: Mean, 95th and 99th-percentile latencies of XenStore requests during 10 VM creation operations. The reported latencies are in μs.

B. net_uk

For these experiments, we run independently each TailBench application inside the user VM and we measure how it is impacted by crash failures.

1) Robustness: Recall that our solution enhances net_uk with several FT feedback loops in order to detect failures at different granularities: the unavailability of the subcomponents (NIC driver and netback) and the unavailability of the entire net_uk. Here, we only evaluate the robustness of our system facing NIC driver crashes because it allows us, through the same experiment, to compare fine-grained (FG) and coarse-grained (CG) FT solutions. We inject a fault in the NIC driver at the middle of the execution of the benchmark. Table IV and Table V present the results. We do not interpret Xoar results here (already discussed in §II-B). Besides, we do not show performance results for vanilla Xen because it is unable to achieve application completion in case of a net_uk failure.

We can see that the fine-grained solution allows quick detection compared to coarse-grained solutions (ours and TFD-Xen): up to a 3.6x difference for detection and 1.4x for repair times (compared to our coarse-grained approach). TFD-Xen is faster to recover because it relies on net_uk replicas linked to backup physical NICs: instead of recovering a failed net_uk unikernel, it switches from one net_uk to another and reconfigures the bindings with the running user VMs. However, TFD-Xen requires at least $N+1$ physical NICs and $N+1$ net_uks to survive N net_uk faults, which results in resource waste and limited resilience over long time intervals. Furthermore, our fine-grained solution avoids packet losses, thanks to the use of a shadow driver that buffers packets in case of failure. For instance, we measured $212,506$ buffered packets for the *sphinx* application. In contrast, the other solutions lead to broken TCP sessions (caused by packet losses) that occur during network reconfiguration (even for TFD-Xen, despite its short recovery time). Moreover, we can see that the fine-grained FT solution reduces the tail latency degradation compared to the coarse-grained solution. Considering the *sphinx* application for instance, the differences are respectively 24.88%, 12.88%, and 5.88% for the mean, 95th and 99th-percentile latencies.

[6]A VM creation request requires 53 XenStore requests, whereas VM destruction, VM migration and vCPU hotplug operations respectively require 47, 24, and 12 requests.

	DT (ms)	RT (s)	PL
FG FT	27.27	4.7	0
CG FT	98.2	6.9	425,866
TFD-Xen [4]	102.1	0.8	2379
Xoar [5]	52×10^3	6.9	1,870,921

TABLE IV: Robustness evaluation of different FT solutions for net_uk. The failed component is the NIC driver.
DT = Fault Detection Time; RT = Fault Recovery Time; PL = number of (outgoing) lost packets.

Regarding the throughput measurements with the AB benchmark, we observe the following results. TFD-Xen achieves the best performance with 45 requests/s. The FG solution is relatively close with 42 requests/s (7.14% gap). In contrast, the CG approach is significantly less efficient with 29 requests/s (55.17%) and Xoar is much worse with 9 requests/s (400%).

2) Overhead: The experiment is the same as previous without fault injection. Table VI presents the results. The overhead incurred by our solution is up to 12.4% for mean latencies, up to 17.3% for the 95th percentiles, and up 12.3% for the 99th percentiles. This overhead is due to periodic communication with the hypervisor to track the driver execution state (§IV-B). Notice that TFD-Xen [4] incurs overhead up to 2.88% for mean latencies, 17.87% for the 95th percentiles, and up to 13.77% for the 99th percentiles.The overhead incurred by Xoar is much higher, as already discussed in §II-B.

Regarding the throughput measurements with the AB benchmark, we observe the following results compared to the vanilla Xen baseline (123 requests/s). Both TFD-Xen and our solutions (FG and CG) exhibit a noticeable but acceptable overhead (13.31%, 12%, and 15% respectively), whereas Xoar incurs a more pronounced performance degradation (1130%[7]).

C. tool_uk

Contrary to other dom0 unikernels, tool_uk does not execute a task permanently.It only starts a task when invoked for performing a VM administration operation. The FT solution does not incur overhead when there are no failures. Therefore we only evaluate the robustness aspect. To this end, we consider the VM live migration operation because it is the most critical one. We run inside the user VM a Linux kernel compilation task and inject a failure during the second stage of the migration process, i.e., when a replica of the migrated VM has been launched on the destination machine, and the memory transfer is ongoing.

We observe that vanilla Xen and Xoar lead the physical machine to an inconsistent state: the migration stops but both the original VM (on the source machine) and its replica (on the destination machine) keep running. This situation leads to resource waste because the replica VM consumes resources. Using our solution, the replica VM is stopped upon failure detection. The detection time is 800ms.

[7]With a refresh period of 5s, Xoar still incurs a performance degradation of up to 697%, significantly worse than our approach. Detailed results for this setup are not reported due to lack of space.

D. Global failure

We also evaluate the robustness of our solution when all the pVM components crash at the same time. We execute the sphinx application from TailBench in a guest and we inject faults to crash all the components simultaneously. In this case, the hypervisor detects the global crash and restores all unikernels in the appropriate order (see §IV-D). The whole recovery of all unikernels takes 15.8s. Concerning application performance, we observe a downtime of 7.85s (corresponding to the time needed for XenStore_uk and net_uk to recover), but the application survives and finishes its execution correctly. We experience a huge degradation of tail latencies due to the long downtime but we allow full and transparent functional recovery of the user VM, unlike vanilla Xen, TFD-Xen, and with a much lower overhead than Xoar (esp. during failure-free execution phases).

E. Scheduling optimizations

We measure the benefits of our scheduling optimizations (§IV-E) in terms of reactivity and CPU time used by our unikernels. Regarding reactivity, we run the *sphinx* application in a guest VM, and we trigger the crash of the net_uk. On average, with the scheduling optimizations, we detect the crash after 141.8ms compared to 149.5ms without, i.e., a 5.15% decrease. Besides, on a fault-free run, compared to a standard scheduling policy, the usage of implicit heartbeats allows a 13% decrease of the CPU time consumed by the pVM components.

VI. Related work

pVM resilience. The projects most closely related to our work are Xoar [5] and TFD-Xen [4]. Given that they are described in detail and evaluated in the previous sections.

Beyond Xoar, a number of projects have investigated the benefits of disaggregating the VMM into multiple isolated components. Murray et al. [28] modified the original Xen platform design in order to move the domain builder (a security-sensitive module within the Xen toolstack running in the pVM) to a separate virtual machine. This work did not investigate fine-grained disaggregation nor fault tolerance. Fraser et al. [2] revisited the design of the Xen platform in order to support "driver domains", i.e., the possibility to isolate each physical device driver in a separate VM. Our contribution builds on this work but also considers fine-grained fault-tolerance mechanisms within driver domains, as well as disaggregation and robustness of other pVM components.

As part of their Xen-based "resilient virtualization infrastructure" (RVI) [29], [30], Le and Tamir briefly discussed how to improve the fault tolerance of the pVM and the driver domains (dVMs). The failure of a driver domain is detected by agents within the dVM kernel and the hypervisor, which triggers the microreboot of the dVM. The failures of the services hosted by the pVM (e.g., XenStore) are detected by an agent running within the pVM. Upon such a detection, the agent issues a hypercall to the hypervisor, and the latter triggers a crash of the whole pVM. Hence, any failure of

	sphinx			xapian			moses			masstree			img-dnn		
	mean	95th	99th	mean	95th	99th	mean	95th	99th	mean	95th	99th	mean	95th	99th
Xen	879.11	1696	1820.84	1.79	4.35	9.67	8.6	39.56	65.64	457.61	475.37	476.2	1.7	3.42	7.6
FG FT	1201.1	3100.12	3891.73	51.19	100.16	1700.31	73.21	473.21	1492.31	821.11	1891.1	2122.18	88.22	440.21	1310.88
CG FT	1500.3	3499.4	4120.91	89.9	154.9	2101.45	100.5	591.51	1833.09	1091.5	2099.1	2461.09	112.01	610.91	1503
TFD-Xen	1159.32	2908.89	3304.2	50.10	98.11	1396.21	70.31	450.22	1101.44	788.3	1381.12	1631.77	80.12	398.32	1116.81
Xoar	8100.4	11026.7	13590.3	5188.1	7238.9	8193.3	5120.4	5581.8	5909.3	10011.2	13444.5	140881.43	1491.9	4721.33	12390.4

TABLE V: Performance of TailBench applications during a net_uk failure (latencies in milliseconds). Lower is better. The failed component is the NIC driver. The first line ("Xen") corresponds to a fault-free baseline.

	sphinx			xapian			moses			masstree			img-dnn		
	mean	95th	99th	mean	95th	99th	mean	95th	99th	mean	95th	99th	mean	95th	99th
Xen	879.11	1696	1820.84	1.79	4.35	9.67	8.6	39.567	65.642	457.61	475.37	476.2	1.7	3.42	7.6
FG FT	901.99	1711.11	1977.15	2.11	5.56	10.61	10.1	40.78	72.44	460.2	491.58	494.3	1.92	4.2	8.21
CG FT	900.12	1792.04	1963.5	2.12	5.96	11.05	11.9	41.3	72.12	461.09	489.03	490.12	1.9	4.3	7.98
TFD-Xen	889.91	1701.43	1911.33	2.11	4.98	10.89	9.12	40.44	73.19	461.18	489.1	493.55	1.9	4.5	8.11
Xoar	6616.44	9026.7	9590.54	3713.98	5535.77	5791.712	3019.33	3507.88	3660.65	8054.53	8642.85	8695	543.9	2526	9603

TABLE VI: Performance of TailBench applications without net_uk failure (latencies in milliseconds). Lower is better.

a given component hosted by the pVM (e.g., XenStore or toolstack) leads to downtime and full recovery for all the other components. In order to tolerate failures of the XenStore, its state is replicated in a dVM, and XenStore and VM management operations are made transactional, through the use of a log stored in a dVM. No detail is provided on the mechanisms used to enforce consistency and availability despite potential concurrent failures of the pVM-hosted components and the backup state in the dVM. Besides, the evaluation of this approach is focused on its resilience against synthetic fault injection. The authors of this solution do not provide any detailed performance measurements (with or without failures), and the code of the prototype is not available. Our work has similarities with this approach but, (i) we apply fault tolerance techniques at a finer granularity, (ii) we explore the ramifications of the interdependencies between services, and (iii) we provide a detailed performance evaluation.

Hypervisor resilience. Some works have focused on improving the resilience of the hypervisor. ReHype [8], [30], [31] is a Xen-based system, which leverages microreboot [32], [33] techniques in order to recover from hypervisor failures, without stopping or resetting the state of the VMs (including the pVM and dVMs) or the underlying physical hardware. NiLyHype [9] is a variant of ReHype, which replaces the microreboot approach with an alternative component recovery technique named microreset (i.e., resetting a software component to a quiescent state that is likely to be valid rather than performing a full reboot) in order to improve the recovery latency. TinyChecker [10] uses nested virtualization techniques (more precisely, a small, trusted hypervisor running below a main, full-fledged hypervisor like Xen) in order to provide resilience against crashes and state corruption in the main hypervisor. Shi et al. [34] proposed a new modular, "deconstructed" design for Xen in order to thwart the most dangerous types of security attacks. Their work focuses on a redesign of the Xen hypervisor (not the dom0 pVM). All these works do not consider the services hosted in the pVM (or driver domains) and are mostly orthogonal to our work. Our contribution leverages these results (i.e., the fact that the hypervisor can be made highly resilient) in order to improve the fault tolerance of the pVM components.

The FTXen project [35] aims at hardening the Xen hypervisor layer so that it can withstand hardware failures on some "relaxed" (i.e., fast but unreliable) CPU cores. In contrast, our work considers the resilience of the pVM components on current hardware, with a fault model that is homogeneous/symmetric with respect to CPU cores.

Some recent projects aim at supporting live reboot and/or upgrade of VMMs without disrupting the VMs [36], [37]. These techniques are focused on code updates for improving the safety and security of the hypervisor component. Hence, they are orthogonal to our contribution. This trend also highlights the crucial importance of our goal: improving the resilience of the remaining components, i.e., the pVM services.

VII. Conclusion

VMMs remain a key building block for cloud computing, and many of them are based on a pVM-based design. We have highlighted that, in this design, the pVM component has become the main weakness in terms of fault tolerance (compared to the bare metal hypervisor component). Besides, the existing solutions only tackle a limited set of pVM services (device drivers) and/or require long failure detection/recovery times and significant performance overheads. To the best of our knowledge, our contribution is the first to propose and demonstrate empirically, a complete approach allowing to achieve both high resilience (against failures of different components and concurrent failures of interdependent services) and low overhead. Our approach currently relies on manual tuning of some important parameters (e.g., for failure detection and scheduling) but, we envision that recently published works could help manage them in a more automated and robust way [38]. Another area for future work is the tuning and optimization of resource allocation for disagreggrated pVM components, which could be extended from existing techniques proposed for a monolithic pVM design [27].

REFERENCES

[1] 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. [Online]. Available: http://tiny.cc/5xt4nz

[2] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "Safe Hardware Access with the Xen Virtual Machine Monitor," in *In Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004.

[3] S. Spector, "Why Xen?" 2009. [Online]. Available: http://www-archive.xenproject.org/files/Marketing/WhyXen.pdf

[4] H. Jo, H. Kim, J. Jang, J. Lee, and S. Maeng, "Transparent fault tolerance of device drivers for virtual machines," *IEEE Transactions on Computers*, vol. 59, no. 11, pp. 1466–1479, Nov 2010.

[5] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield, "Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 189–202. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043575

[6] H. Kasture and D. Sanchez, "TailBench: a benchmark suite and evaluation methodology for latency-critical applications," *IEEE International Symposium on Workload Characterization (IISWC)*, 2016. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7581261&isnumber=7581253

[7] D. J. Scales, M. Nelson, and G. Venkitachalam, "The Design of a Practical System for Fault-tolerant Virtual Machines," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 4, pp. 30–39, Dec. 2010. [Online]. Available: http://doi.acm.org/10.1145/1899928.1899932

[8] M. Le and Y. Tamir, "ReHype: Enabling VM Survival Across Hypervisor Failures," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '11. New York, NY, USA: ACM, 2011, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1952682.1952692

[9] D. Zhou and Y. Tamir, "Fast Hypervisor Recovery Without Reboot," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2018, pp. 115–126.

[10] C. Tan, Y. Xia, H. Chen, and B. Zang, "TinyChecker: Transparent protection of VMs against hypervisor failures with nested virtualization," in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*, June 2012, pp. 1–6.

[11] The Linux Foundation, "Xen Project." [Online]. Available: https://xenproject.org

[12] A. Madhavapeddy and D. J. Scott, "Unikernels: The Rise of the Virtual Library Operating System," *Commun. ACM*, vol. 57, no. 1, pp. 61–69, Jan. 2014. [Online]. Available: http://doi.acm.org/10.1145/2541883.2541895

[13] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering Device Drivers," *ACM Trans. Comput. Syst.*, vol. 24, no. 4, pp. 333–360, Nov. 2006. [Online]. Available: http://doi.acm.org/10.1145/1189256.1189257

[14] "Xl." [Online]. Available: https://wiki.xenproject.org/wiki/XL

[15] "Mirage OS." [Online]. Available: https://mirage.io

[16] O. Mutlu, "The RowHammer Problem and Other Issues We May Face As Memory Becomes Denser," in *Proceedings of the Conference on Design, Automation & Test in Europe*, ser. DATE '17. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2017, pp. 1116–1121. [Online]. Available: http://dl.acm.org/citation.cfm?id=3130379.3130643

[17] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks," in *S&P*, May 2019, best Practical Paper Award, Pwnie Award Nomination for Most Innovative Research. [Online]. Available: http://tiny.cc/trt4nz

[18] "etcd." [Online]. Available: https://etcd.io

[19] "Exploring performance of etcd, zookeeper and consul consistent key-value datastores." [Online]. Available: http://tiny.cc/8hu4nz

[20] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 305–319. [Online]. Available: http://tiny.cc/wku4nz

[21] "MiniOS." [Online]. Available: https://github.com/mirage/mini-os

[22] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers, "Nooks: An architecture for reliable device drivers," in *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, ser. EW 10. New York, NY, USA: ACM, 2002, pp. 102–107. [Online]. Available: http://doi.acm.org/10.1145/1133373.1133393

[23] V. Narayanan, A. Balasubramanian, C. Jacobsen, S. Spall, S. Bauer, M. Quigley, A. Hussain, A. Younis, J. Shen, M. Bhattacharyya, and A. Burtsev, "LXDs: Towards Isolation of Kernel Subsystems," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 269–284. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/narayanan

[24] "Xen Security Modules." [Online]. Available: http://tiny.cc/zdu4nz

[25] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation*, ser. NSDI'05. USA: USENIX Association, 2005, p. 273–286.

[26] "ApacheBench." [Online]. Available: http://tiny.cc/anu4nz

[27] D. Mvondo, B. Teabe, A. Tchana, D. Hagimont, and N. De Palma, "Closer: A New Design Principle for the Privileged Virtual Machine OS," in *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2019, pp. 49–60.

[28] D. G. Murray, G. Milos, and S. Hand, "Improving Xen Security Through Disaggregation," in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '08. New York, NY, USA: ACM, 2008, pp. 151–160. [Online]. Available: http://doi.acm.org/10.1145/1346256.1346278

[29] M. Le and T. Yuval, "Resilient Virtualized Systems Using ReHype," UCLA Computer Science Department, Tech. Rep. 140019, October 2014.

[30] M. Le, "Resilient Virtualized Systems," UCLA Computer Science Department, Ph.D. thesis 140007, March 2014.

[31] M. Le and Y. Tamir, "Applying Microreboot to System Software," in *2012 IEEE Sixth International Conference on Software Security and Reliability*, June 2012, pp. 11–20.

[32] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot — A Technique for Cheap Recovery," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251257

[33] A. Depoutovitch and M. Stumm, "Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 181–194. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755933

[34] L. Shi, Y. Wu, Y. Xia, N. Dautenhahn, H. Chen, B. Zang, and J. Li, "Deconstructing Xen," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017. [Online]. Available: http://tiny.cc/j3t4nz

[35] X. Jin, S. Park, T. Sheng, R. Chen, Z. Shan, and Y. Zhou, "FTXen: Making hypervisor resilient to hardware faults on relaxed cores," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 451–462.

[36] S. Doddamani, P. Sinha, H. Lu, T.-H. K. Cheng, H. H. Bagdi, and K. Gopalan, "Fast and Live Hypervisor Replacement," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE 2019. New York, NY, USA: ACM, 2019, pp. 45–58. [Online]. Available: http://doi.acm.org/10.1145/3313808.3313821

[37] X. Zhang, X. Zheng, Z. Wang, Q. Li, J. Fu, Y. Zhang, and Y. Shen, "Fast and Scalable VMM Live Upgrade in Large Cloud Infrastructure, booktitle = Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems," ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 93–105. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304034

[38] S. Wang, C. Li, H. Hoffmann, S. Lu, W. Sentosa, and A. I. Kistijantoro, "Understanding and auto-adjusting performance-sensitive configurations," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 154–168. [Online]. Available: https://doi.org/10.1145/3173162.3173206